CS2112—Fall 2012

Assignment 2

Ciphers and Encryption

Due: September 18, 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 widely used RSA public-key encryption algorithm, which you use multiple times a day. You are tasked with creating a command-line application that can be used to generate, save, and use the ciphers you build. You should implement the system in a way that uses inheritance to share code between different ciphers.

0 Changes

- 9/7 – removed reference to the `Cipher.java` interface.
- 9/10 – changed the frequency analyzer specification to return a Caesar cipher.
- 9/12 – fixed `ChunkReader.nextChunk` signature.
- 9/13 – better guidance on bit length for prime factors p and q.
- 9/16 – clarified how to convert bytes to characters.

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 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.

1.3 Documentation

For this homework we ask that you document all of your methods with Javadoc style comments. How to write Javadocs will be covered in lab, and the course staff will be able to answer any questions during office hours.

1.4 Provided interfaces

On homework one there were many questions about changing the provided interfaces, so here the guidelines. If we provided a method you must implement it, even if you don't use it. You also may not change the return type, argument types, or number of arguments for any provided method. You may rename the arguments. You may also add `throws` declarations to the methods if you feel it makes sound design sense to do so. You may (and are encouraged to) add as many additional methods as you need.

2 Classical ciphers

2.1 Overview

A cipher is a way to protect a message by changing its letters or characters so that 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.

2.1.1 The Caesar cipher

A Caesar cipher is a monoalphabetic cipher that functions by mapping an alphabet to a 'shifted' version of itself. For example, if we use a cipher where each letter is shifted to the right by one (A → B, B → C, . . . , Z → A), we would encode the message CAT as DBU. We would say that this is a Caesar cipher with a shift parameter of 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, including negative numbers. A shift of -1 would map A → Z, B → A, etc.

2.1.2 Simple monoalphabetic ciphers

The Caesar cipher is a particularly simple example of a monoalphabetic *substitution cipher*, a cipher that replaces pieces of text by corresponding (and hopefully different) pieces of text. In the Caesar cipher from a single value (the shift parameter) you can determine the entire mapping. To make the cipher hard to break, This cipher will randomly generate a mapping such that knowing the mapping of any one letter pair gives no information about the remaining pairs.

Note that random number generators used in computing are almost never truly random, but rather, *pseudorandom*. They are produced by an algorithm called a *random number generator* whose output is difficult to distinguish from true random numbers. A *cryptographic* random number generator is a random number generator for which an adversary cannot have a practical algorithm that distinguishes its pseudorandomness from true randomness.

2.1.3 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, because even if the same letter appears many times in the plaintext it may appear in the cipher text as many different letters.

2.2 Implementation

Your task is to implement a Caesar cipher, a random substitution cipher, and a Vigenère cipher. Your code will be accessed using the `CipherFactory.java` class. We have provided a cipher interface in the two files `EncryptionCipher.java` and `DecryptionCipher.java`. In addition to implementing these interfaces, your ciphers should extend the abstract class in `AbstractCipher.java`. You may find it useful to define additional abstract classes, as you should aim for elegant program design that *minimizes the repetition of common code* across similar classes and methods. Often times the best program isn't the one with the most lines of code, but rather the program that accomplishes the task with the *fewest* lines.

2.2.1 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, specifically spaces, tabs, and newlines. All other characters should be discarded. For example the sentence "`I really like Cornell, don't you?`" would become the plaintext "`I REALLY LIKE CORNELL DONT YOU`" You may assume that when decrypting, the program will encounter only uppercase letters and whitespace.

2.2.2 Saving the cipher

To save a substitution cipher to a file, simply print the encrypted alphabet to the file, followed by a newline. For example, saving a Caesar cipher with shift parameter 1 would create a file that looks like `BCDEFGHIJKLMNOPQRSTUVWXYZA\n`.

[1] Not actually invented by Vigenère

3 Cipher cracking

3.1 Frequency analysis

Monoalphabetic ciphers are easiest 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 and decrypt the message.

3.1.1 Implementation

You should implement a tool to analyze the frequency of letters over multiple unencrypted texts in the target language, and then use this analysis to crack messages encrypted with a Caesar cipher. You should do so by completing the methods provided in `FrequencyAnalyzer.java`. Like the encrypter, the FrequencyAnalyzer should keep track of only uppercase English letters and handle other characters appropriately (convert or ignore). How you do this is up to you, but here is a hint: there are only 25 possible Caesar ciphers. Find the one that best explains the frequencies of the symbols seen in the ciphertext, under the assumption that the sample text provided contains letters with frequencies typical of the plaintext (e.g., the frequencies found in English-language text).

4 RSA encryption

RSA is probably the most widely used encryption schema in the world today. 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, and knowing the public key doesn't help figure out the private key. Public-key cryptography makes the secure Internet possible. Before public-key cryptography, keys had to be carefully exchanged between people who wanted to communicate, often by non-electronic means. Now RSA is routinely used to exchange keys without allowing anyone snooping on the channel to understand what has been communicated.

RSA is believed to be very secure, based on the widely held assumption that no one has an efficient algorithm for factoring large numbers; deriving the private key from the public key appears to be as hard as factoring.

4.1 The algorithm

4.1.1 Key generation

1. Choose two random and prime numbers p and q. These must be kept secret. The larger p and q are, the stronger the encryption will be.

2. Compute $n = p \cdot q$. This is the *modulus* used for encryption.

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 e is relatively prime to m. That is, the greatest common divisor of e and m is 1.
5. Compute $d = e^{-1} \bmod m$. That is, a value 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.[2]

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

4.1.2 Encryption

A plaintext message s is encrypted as ciphertext c via the following formula: $c = s^e \bmod n$. Note that it can be done with only the publicly known n and e.

4.1.3 Decryption

An encrypted message c is decrypted as plaintext s via the following formula: $s = c^d \bmod n$. Note that this *cannot* be done with just the public key. This works because $(s^e)^d = s \mod n$.

4.2 Implementation

4.2.1 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 `certainty` $= 20$. The numbers generated by this constructor are only 'probably' prime, but given a high enough `certainty`, this is good enough. You'll want to choose a bit length (the `bitlength` parameter) for p and q such that their product contains the right number of bits. Recall that the product of two n-digit numbers contains at most $2n$ digits.

4.2.2 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. The plaintext should be broken down into chunks of size 117 bytes by a class that implements `ChunkReader.java`. The bits from the sequence of plaintext bytes should then be used to construct the `BigInteger` object to which the RSA algorithm is applied, with the least significant bits of the number starting from the first byte in the chunk that is read from the file.

Since the input length is unlikely to be an even multiple of 117, it is necessary for each chunk to keep track of the actual number of bytes of data contained in the chunk. This is done by extending the chunk with a 118th byte containing the number of actual data bytes in the chunk (from 1 to

[2] Euler's theorem says for any x, m, n where m is the totient of n, and x is relatively prime to n, $x^m \equiv 1 \mod n$. Therefore if $ed = km + 1$ for some k, $x^{ed} = x^{km+1} = x \cdot x^{km} \equiv x \cdot 1 = x \mod n$.

117). There are always 118 total bytes in the data that is converted to a `BigInteger`: up to 117 data bytes, plus one extra byte to keep track of the chunk size. If fewer than 117 data bytes are available, padding bytes are inserted so that the size byte is still the 118th.

In general, encryption using the RSA algorithm can make the number larger. Therefore, the numbers generated by encryption are converted back into 128-byte arrays that are written to the output file. The length of encrypted output is always a multiple of 128. The difference between 118 and 128 leaves plenty of room for the number to grow when it is encrypted, so encryption never overflows the available space.

For stronger encryption, the padding bytes added to short chunks would be random bytes, protecting against a dictionary attack. However, your program will be easier to debug if you set all such bytes to zero, and we will be fine with such a solution.

When writing out an encryption result to a file, exactly the encrypted 128-byte chunks should be written. Hint: Use `OutputStream` directly rather than converting to a string and back.

Decryption
Decryption is simply the inverse of encryption. The input is read in chunks of size 128, which are converted to `BigIntegers` and run backward through the transformation. Byte 117 in the decrypted result then specifies how many bytes of data to extract from the array as the decrypted output.

Encrypting a file and then decrypting the result should give back exactly the original file. How encryption to a string works is less critical, but when converting sequences of bytes to characters, byte values 0–127 should be converted to characters 0–127, and byte values -128 to -1 should be converted to characters 128–255 correspondingly. The ISO-8859-1 character set should be helpful for this; see `java.nio.charset.Charset`. You can create a `Charset` using the `forName` method and then use that character set to do encoding and decoding of strings from byte arrays.

4.2.3 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.

4.3 Cracking RSA

While very secure when used correctly, RSA can be broken when small factors p and q are used. All one has to do is get p and q by factoring n. The decryption key can then be computed from the encryption key. 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 use Pollard's Monte-Carlo factorization. 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), and report the time it took in your `README.txt`. You should also extrapolate to estimate how long it would take to break 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
- Wikipedia article on RSA
- Abstract Class InputStream

5 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>
```

or

```
java -cp <CLASS_FILE_DIR> <MAIN_CLASS_NAME> <CIPHER_TYPE> <CIPHER_FUNCTION> <OUTPUT_O
```

Cipher type

There are three different 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 monoalphabetic substitution cipher should be randomly generated and used by this program.

`--classicalfa [-t <examples> | -c <encrypted>]`: A Caesar 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, max length 128 characters)

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

Next, 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

Finally the user may add as many output flags as they wish.

Output options

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

`--out <file>`:prints the result of applying the cipher (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.

5.1 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
  ```

5.2 Errors

Should anything go wrong during execution, including user-error (malformed requests, missing files), your program should not simply ’die’. Also no Java exception should ever be shown to the user. Instead your program should detect the error and find a sensible way to resolve or communicate the problem.

6 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. Possible extensions include but are not limited to the following:

- Additional commands, including more complex ciphers of your choice.
- Cryptanalysis for simple substitution ciphers, or other ciphers such as Vigenère.
- More secure storage of ciphers in files.
- Digraph (two character sequence) frequency analysis will yield more accurate automatic decryption than single-letter frequency analysis.

- Randomized RSA padding.

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.

7 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.*

Par for this assignment is about 1200 lines of code.