

CS 2112 Fall 2016

Assignment 2

Ciphers and Encryption

Due: **Wednesday, September 14**, 11:59PM

Design Overview due: Thursday, September 8, 11:59PM

In this assignment, you will build a system that provides multiple ways to encrypt and decrypt text. The first part of the assignment focuses on simple ciphers and cipher cracking, while the second part explores the widely used RSA public-key encryption algorithm. You are tasked with creating a command-line application that can generate, save, and use the ciphers you build. Your implementation of the system should use inheritance to share code between different ciphers.

0 Updates

- None yet – watch this space.

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 to 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 without explaining things that are obvious.

1.2 Partners

You must work alone for this assignment. Remember that the course staff is happy to help with any 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 assignment, we ask that you document all of your methods with Javadoc-style comments. We will cover how to write Javadoc-compliant comments in lab, and the course staff can help with this during office hours.

1.4 Provided interfaces

You must implement all methods provided, even if you do not use them. You may not change the signature and return type of any provided method. Formal parameters may be renamed, but this is

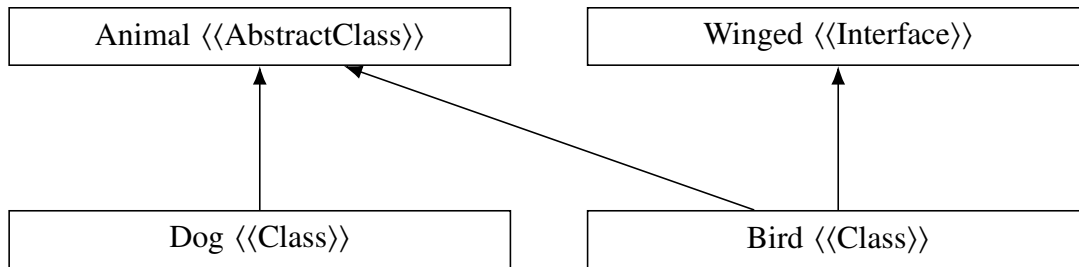


Figure 1: An example of a class diagram. Dog, which is a class, extends Animal, which is an abstract class. Bird, which is a class, extends Animal and implements Winged, which is an interface.

discouraged. You may also add `throws` declarations to methods if you believe they improve your design. You may (and are encouraged to) add as many additional classes and methods as you need.

1.5 Start early

You should start on this assignment as early as possible. It will require careful thought about your design. Trying to do it all at the last minute will almost certainly result in code that is both messy and broken.

2 Design overview document

Many components of this assignment share common functionality. For example, many ciphers need to read input, either from the command line or from files. Implementing this functionality once in a common place and allowing each cipher to use it by calling a method is better than having duplicated code for each cipher. A clean, readable program exploits inheritance by collecting common code from individual components into a single place.

To ensure that your design is reasonable for this assignment, we require that you submit a design overview document before the assignment is due. A design document should contain a diagram of the class hierarchy you are planning to implement, along with a short paragraph briefly justifying your design decisions. When designing the class hierarchy, try to eliminate redundant code by factoring out functionality shared across the implementations of multiple ciphers. Figure 1 shows an example of a class diagram that you can use as a guide.

Submit your design overview document as a PDF file named `a2_designOverview.pdf` to CMS. Scans of handwritten diagrams are acceptable.

3 Substitution ciphers

3.1 Overview

In cryptography, a *substitution cipher* attempts to obscure a message by replacing letters or sequences of letters by other letters or sequences of letters so as to make the transformed message

unreadable to anyone but the intended recipient. The original message is called the *plaintext*, and the transformed message is called the *ciphertext*.

Monoalphabetic substitution ciphers are the most rudimentary type of substitution cipher, using a fixed one-to-one correspondence between letters in the plaintext and letters in the ciphertext. The sender and receiver must know the correspondence in advance. The receiver can then apply the inverse substitution to decode the message.

3.1.1 The Caesar cipher

The Caesar cipher is a monoalphabetic cipher that maps an alphabet to a ‘shifted’ version of itself. For example, a cipher where each letter is shifted to the right by one ($A \rightarrow B, B \rightarrow C, \dots, Z \rightarrow A$) would encrypt the message CAT as DBU. This particular Caesar cipher has a *shift parameter* of 1. A shift parameter of 0 gives the *identity function* in which each letter is mapped to itself. Shift parameters are not limited to values 1–26; they can be any integer, including negative numbers. A shift of -1 maps $A \rightarrow Z, B \rightarrow A$, and so on. This method of encryption was used by Julius Caesar for military communications.

3.1.2 Random substitution ciphers

The Caesar cipher provides almost no security, because the shift parameter can be learned from knowing how a single symbol was encrypted, and this determines the entire mapping.

A monoalphabetic substitution cipher is harder to break if the mapping between plaintext and ciphertext symbols is random. In this case, knowing how any one symbol is mapped gives very little information about how other symbols are mapped.

Random number generators used in computing, such as those provided by the Java Random class, are typically not truly random but only *pseudorandom*. They are produced by an algorithm called a *pseudorandom number generator* (PRNG) whose output is difficult to distinguish from true random numbers. A *cryptographic* random number generator is a random number generator for which there are no known practical algorithms that can distinguish their pseudorandomness from true randomness.

3.1.3 The Vigenère cipher

Another way to strengthen the Caesar cipher is to use different substitutions on different letters. The Vigenère cipher¹ 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 key ABC yields DCWBNRH, because the shifts ABCABCA are applied to the plaintext. The Vigenère cipher makes frequency-based cryptanalysis more difficult, especially if the key is long, because even if the same letter appears many times in the plaintext, it may appear in the ciphertext as many different letters. This example is shown visually below where the top row is the word to be encrypted, the

¹The cipher was actually invented by an Italian cryptologist [Giovann Battista Bellaso](#) in 1553. [Blaise de Vigenère](#), a French cryptographer, created a different, stronger [autokey cipher](#) in 1586.

middle row is the repeated pattern of shifts, and the bottom row is the result of encrypting the top row with the repeated pattern.

C	A	T	A	L	O	G
A	B	C	A	B	C	A
D	C	W	B	N	R	H

3.2 Implementation

Your task is to implement the Caesar cipher, the random monoalphabetic substitution cipher, and the Vigenère cipher. Your cipher implementations will be accessed through class `CipherFactory`. The interfaces `EncryptionCipher` and `DecryptionCipher` define methods needed for encryption and decryption. In addition to implementing these interfaces, your ciphers should extend the abstract class `AbstractCipher`.

We will be looking for elegant program design that *minimizes the repetition of common code*. The best program is rarely one with the most lines of code, but rather one that accomplishes the task most simply and with the *least* code. You will find inheritance a valuable language feature for avoiding repetition. Abstract classes may further help achieve this goal.

3.2.1 Letter encoding

A consistent standard is important for representing characters during both the encryption and decryption. All letters should be converted to their uppercase equivalents. Whitespace—specifically, spaces, tabs, and newlines—should be maintained. 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 characters.

3.2.2 Saving the cipher

To save a Caesar or monoalphabetic 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 creates a file whose content is as follows:

```
1 BCDEFGHIJKLMNOPQRSTUVWXYZA
2
```

To save a Vigenère cipher, print the key to the file, followed by a newline. For example, a Vigenère cipher with the key “KEY” saves to a file that looks like

```
1 KEY
2
```

4 Cipher cracking using frequency analysis

Monoalphabetic ciphers are possible to break using frequency analysis. A cryptanalyst 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.

4.1 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 class `FrequencyAnalyzer`. Like the encrypter, `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 26 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 (i.e., the frequencies found in English-language text). If you are looking for large chunks of English text for testing you may find [Project Gutenberg](#) useful.

5 RSA encryption

RSA is one of the most widely used encryption schemes in the world today. It 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 does not help crack 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, because its security is 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.

5.1 The algorithm

5.1.1 Key generation

1. Choose two random and distinct prime numbers p and q . These must be kept secret. The larger p and q are, the stronger the encryption will be.
2. Compute their product $n = pq$. This will be the *modulus* used for encryption.
3. Compute $\varphi(n)$, the *totient* of n . It is the number of positive integers less than n that are relatively prime to it. For a product of primes p and q , the totient is easy to compute: $\varphi(n) = (p-1)(q-1)$. Notice that computing $\varphi(n)$ requires knowledge of p and q . If you would like to learn more about the totient function, see [Euler's Totient Function](#).

4. Choose an integer e such that $1 < e < \varphi(n)$ and e is relatively prime to $\varphi(n)$. That is, the greatest common divisor of e and $\varphi(n)$ is 1.
5. Compute the decryption key d as the multiplicative inverse of e modulo the totient, written as $e^{-1} \bmod \varphi(n)$. This is a value d such that $1 \equiv ed \bmod \varphi(n)$. Such a value d can be found using [Euclid's extended greatest-common-divisor algorithm](#).

The public key is the pair (n, e) and the private key is the pair (n, d) . To allow people to encode messages to you, you can advertise your public key, say on your webpage, keeping your private key secret.

5.1.2 Encryption

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

5.1.3 Decryption

An encrypted message c is decrypted as plaintext s via the following formula: $s = c^d \bmod n$. Note that this requires knowledge of the private key.

5.1.4 Why does this work?

If we encrypt and then decrypt a plaintext message s , we obtain a new message $s' = (s^e \bmod n)^d \bmod n$. For the cryptosystem to work, we must have $s = s'$. This will be true by *Euler's theorem*, which states that $s^{\varphi(n)} \equiv 1 \bmod n$ (provided s and n are relatively prime).

By the properties of modular arithmetic, we can pull the mod n to the outside:

$$s' = (s^e \bmod n)^d \bmod n = s^{ed} \bmod n.$$

The numbers e and d were chosen to be multiplicative inverses modulo $\varphi(n)$, which means that $ed = 1 + k\varphi(n)$ for some integer k , therefore

$$s^{ed} = s^{1+k\varphi(n)} = s \cdot (s^{\varphi(n)})^k.$$

By Euler's theorem, $s^{\varphi(n)} \equiv 1 \bmod n$, so $s^{ed} \equiv s \bmod n$, as desired.

For example, $\varphi(10) = \varphi(2 \cdot 5) = 1 \cdot 4 = 4$, and $3^4 = 81 \equiv 1 \equiv 2401 = 7^4 \bmod 10$. In fact, we can use 3 and 7 as e and d , since $3 \cdot 7 = 21 \equiv 1 \bmod \varphi(10)$. Let's try it out on the message 8. We encrypt it as $8^3 = 512 \equiv 2 \bmod 10$. Going back the other way, $2^7 = 128 \equiv 8 \bmod 10$. It works!

There is a minuscule chance that the message s will not be relatively prime to n , which will cause this procedure to fail. However the likelihood of this happening is 1 in $pq/(p+q-1)$, which is negligible for large primes p and q .

5.2 Implementation

5.2.1 Dealing with large numbers

RSA involves large numbers, so you should use 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 value of `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 to store in 128 bytes. Recall that the product of two n -digit numbers contains at most $2n$ digits. There are also tables of large primes available from [The Primes Pages](#).

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. The plaintext should be broken down into chunks of size 117 bytes by implementing interface `ChunkReader`. 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 as the first byte of each chunk**. Note that we do not need to convert case or special characters as we did with the substitution ciphers, as we are not working with characters but with their underlying numeric representations.

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 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 picture form, the chunks look something like this:



In general, encryption using the RSA algorithm can make the number larger. For this reason, the numbers generated by encryption are converted back into 128-byte arrays that are written to the output file, and 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 could be random bytes, protecting against dictionary attacks. However, your program will be easier to debug if you set all such bytes to zero, and we will accept such a solution.

When writing out the resulting ciphertext to a file, exactly the encrypted 128-byte chunks should be written, not the textual representation of the number. Hint: Use `OutputStream` directly rather than converting to a string and back.

We strongly recommend that you test converting text and files to and from chunks without any encryption before you try implementing the RSA algorithm itself. Doing this will help you catch bugs while the number of possible causes is still small.

Example Consider converting the plaintext string “CS2112” into an appropriately formatted array of bytes. Each character is associated with a unique number that is fixed and universal, as determined by the Unicode standard. These numbers are called *code points*. The characters in the string CS2112 correspond to the following code points, listed here in hexadecimal (0x indicates a hexadecimal, or base-16, numeral):

character	code point
C	0x43
S	0x53
2	0x32
1	0x31

The code point corresponding to a character is an abstract entity. For representation on a computer, the code points must be translated to an array of bytes by a *character encoding*. With the default ISO-8859-1 encoding, only code points 0x00–0xFF are supported. The string CS2112 will be encoded as the byte array {0x43, 0x53, 0x32, 0x31, 0x31, 0x32}. Here the translation from code points to bytes is direct, but this is not necessarily so for other character encodings. For example, with the UTF-8 encoding, characters above 0x7F will translate to multi-byte sequences. For this assignment, we will use the ISO-8859-1 encoding exclusively.

With 6 bytes of plaintext data, there will be $117 - 6 = 111$ bytes of padding, and the final byte will contain 6 to signify the size of the actual “payload.”

When interacting with data read from a file, the data should be handled directly as bytes, since the data may not make sense as characters under any character encoding. You should be able to encrypt .class files, for example.

Appendix A contains several examples showing expected results when encrypting and decrypting with RSA. We recommend using these examples to test your program.

Decryption Decryption is the inverse of encryption. The input is read in chunks of size 128, which are converted to `BigIntegers` and run backwards through the transformation. The 118th byte (that is, byte 117) in the decrypted result 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.

5.2.3 Saving the keys

The keys can be saved to files.

Public Key When a public key is stored to a file, the file should contain the string representation of n in decimal, followed by a newline, followed by the string representation of e in decimal, followed by a newline. That is, the file should look like this:


```
1 <n>
2 <e>
3
```

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 . Thus the file for a private key should look like this:

```
1 <n>
2 <e>
3 <d>
4
```

5.2.4 Encrypting and decrypting large files

Your implementation of the various ciphers should not crash when attempting to encrypt or decrypt input larger than the Java heap. For large input files, **you will not be able to bring the entire input into memory at once**, so do not assume you can.

5.3 Useful resources

- class [BigInteger](#)
- class [String](#)
- class [Byte](#)
- abstract class [InputStream](#)
- Wikipedia article on [RSA](#)
- Wikipedia article on [Factory design pattern](#)

For this assignment it may be helpful to know what is really stored in a file, especially if you are having trouble with reading or writing to a file. Viewing a file using a text editor is not reliable when you're dealing with binary data. Suppose you have a file `output.txt` and you want to see what's in it.

On Linux and Mac, either of these commands will show you what's there very clearly:

```
od -Ad -txC -tc output.txt
xxd output.txt
```

On Windows there are various “hex editors” that you can use. These editors display data in hexadecimal (base 16), in which the digits are 0–9, A–F. A single byte of file data ranges from 0 to FF (255). People seem to like HxD; hexedit is also okay. Hex editors also exist for Linux/Mac, of course. There is also a plug-in called EHEP for Eclipse that purports to provide hex editor support.

6 Command line invocation

The final goal of this assignment is to create a command-line interface to allow users to interact with the code that you write in this assignment. We have provided you with the structure to parse the

command line arguments, and it is your responsibility to implement the appropriate actions. You may choose to “fill in the blanks” in the code we provided, or to rewrite it as you please. However, to facilitate grading, please place your “main” method in the file we provided, `Main.java`.

Regardless of whether you choose to use our structure or your own, remember that we are looking for a design that *minimizes the repetition of common code*.

The user may provide commands of the following form via the console:

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

Cipher type There are four different cipher types that we have asked you to implement, and the flags for each of them are as follows.

- `--monosub <cipher_file>`: A monoalphabetic substitution cipher is loaded from the file specified.
- `--caesar <shift_param>`: A Caesar cipher with the given shift parameter is used for these operations.
- `--random`: A monoalphabetic substitution cipher is randomly generated and used by this program.
- `--crackedCaesar [-t <examples> | -c <encrypted>]`: A Caesar cipher is constructed using frequency analysis with the files flagged `-t` listed in examples as the unencrypted language and files tagged `-c` as the encrypted language. A user may provide any number of `-t` and `-c` flags, and in any order.
- `--vigenere <key>`: Creates a Vigenère cipher using the given keyword (given as a string, maximum length 128 characters)
- `--vigenereL <cipher_file>`: Loads a Vigenère cipher from the given file.
- `--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.

Cipher functions Next, at most one of the following options may also be specified by the user.

- `--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 Finally, the user may add as many output flags as they wish. 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`).

- `--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.

6.1 Examples

- Make a new Caesar cipher with shift parameter 15, apply it to the provided message, output the result to file `encr.txt`, and save the cipher to file `ca15`:

```
java -jar <your_jar> --caesar 15 --em 'ENCRypt Me!'
--out encr.txt --save ca15
```

- Load the cipher from file `ca15`, decrypt the message in file `encr.txt`, and print the result to the console:

```
java -jar <your_jar> --monosub 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> --crackedCaesar -t moby-dick.txt -c mystery.txt
-t frankenstein.txt -t macbeth.txt --df mystery.txt
--save brokenCiph --print
```

- Create an RSA encrypter, encrypt the given message, save the ciphertext 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 the resulting plaintext, and also save it to a file:

```
java -jar <your_jar> --rsaPr --df encr.txt --out decr.txt --print
```

6.2 Errors

Should anything go wrong during execution, including user error (malformed requests, missing files), your program should not simply die. For example, 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 (`System.out`). 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.

7 Advice

This assignment involves writing much more code than Assignment 1 and demands more careful design. Starting early is essential. Trying to do it all at the last minute is nearly certain to result in code that is both messy and broken. Par for this assignment is about 900 lines of code, assuming you design a reasonable class hierarchy that allows for effective code reuse.

It can also be a challenging debugging exercise if you make mistakes, especially for the RSA cipher. Think carefully about the code you are writing and convince yourself that it is correct. You will also need to test your code carefully. Try not only “normal” inputs but also corner cases. Use the examples provided in the appendix [A](#) to test your code.

7.1 Use assertions

Getting RSA to work can be the most challenging part of the assignment if you are not careful of design, testing, and debugging. Much of the problem comes from small issues when chunking bytes, padding, and converting to and from `BigInteger` objects. If a bug is propagated through the encryption process, it becomes infeasible to tell where the problem is.

We strongly recommend using assertions at each step to help pinpoint bugs. You can use assertions to confirm things like the length of your chunks, the fact the values do not change when converted to and from a `BigInteger` object without encryption, and other properties that you expect to be true. If there are more complicated things you would like to check, write extra methods to check them and call those methods from an assertion.

Enabling assertions By default, assertions are disabled, so that programmers can use computationally expensive assertions without hurting the performance of production code. To enable assertion checking, the program must be run with the `-ea` flag. This flag can be passed as a VM argument in the Run Configurations feature of Eclipse. Check out [this stackoverflow post](#) for help. The [Oracle guide on how to use assertions](#) may also be a helpful resource.

7.2 Build and test incrementally

When building large programs, it is helpful to test your code as you go. Continuous testing increases the chance that any new bugs that show up are the result of code you wrote recently. Ideally, at every point during development, you have some incomplete (but correct) code that offers a firm foundation for further work.

Think about how to develop your code in an order that allows you to test it as you go. Assertions that check preconditions and class invariants are helpful ways to test your code as you develop it.

It is also helpful to design your test cases ahead of time. A good set of test cases will make incremental testing much more effective, and will actually help you pinpoint the key issues your code has to deal with before you even write the code. Additionally, testing your RSA cipher with a relatively small modulus should be a helpful technique to pinpoint bugs in your program. For example, use $n = 55$, $e = 3$, and $d = 27$.

7.3 Read specs carefully

The specification for `BigInteger` has some subtle issues that may complicate your task. Read the specifications carefully, keeping the following issues in mind:

Endian The RSA algorithm specified here calls for the first byte from the file to represent the least significant bits of the number passed into the algorithm. Such a numeric representation is said to be *little-endian*. However, the relevant constructors for the `BigInteger` class expect a *big-endian* byte array in which the most significant byte comes first. These are opposite but equally valid conventions. Your code will need to deal with the difference correctly to earn full credit. Switching endianness can be achieved by reversing the byte array representing a number. In your program, this switch will need to occur whenever a `BigInteger` is created or when its bytes are retrieved during RSA encryption and decryption.

Sign The `BigInteger()` constructor expects a byte array in the *two's-complement representation*. This means that the most significant bit of the first byte represents the sign of the number. In fact, bytes themselves are in two's complement: using 8 bits, they represent numbers between -128 and 127 . A positive `BigInteger` may therefore need an extra zero byte in the most significant position to avoid having the number interpreted as negative. For example, the array $\{-128\}$ represents -128 . To represent positive 128 , we need the longer byte array $\{0, -128\}$. You are likely to encounter this issue both when constructing `BigInteger`s and when converting them back into byte arrays.

Character encoding Plaintext bytes outside the ASCII range of $[0, 127]$ will show up as negative values in the range $[-128, -1]$. For the purpose of representing the number to be encrypted, however, we are interpreting bytes as unsigned integers in the range $[0, 255]$, so -128 becomes 128 and -1 becomes 255 . The `BigInteger` constructor should take care of most of this behind the scenes, except for the most significant byte of the given array.

























You must use the ISO-8859-1 character set for this assignment; 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 to and from byte arrays. Equivalently, you can also pass the string "ISO-8859-1" as an argument to certain `String` methods and constructors to specify that you want to use this encoding into bytes.

7.4 Hex editors

For this assignment, you may find it useful to be able to view and edit bytes in hexadecimal form. This can be done using a hex editor such as HexFiend for Mac or HexEdit for Windows. You can also view an integer n in base 16 using `Integer.toString(n, 16)`.

8 HARMA

For full credit, you are not required to do anything more than what is specified so far, but for good **HARMA** you may add additional features. Possible extensions include but are not limited to the following:

- Randomized RSA padding.
HARMA:    
- **Cipher block chaining** for stronger RSA (instead of the current “electronic codebook” encryption mode).
HARMA:    
- Cryptanalysis for simple substitution ciphers, or other ciphers such as Vigenère, and a command to decrypt such messages.
HARMA:    
- Digraph (two-character sequence) frequency analysis to more accurate automatic decryption than single-letter frequency analysis.
HARMA:    
- More secure storage of ciphers in files.
HARMA:    
- Additional ciphers of your choice.
HARMA:    

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.

9 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 assignment with. It should also include descriptions of any extensions you implemented.

Do not include any files ending in `.class`.

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.*

A Full RSA Examples

We have provided you with three examples of RSA encryption, and we strongly recommend that you use these examples (as well as many more) to test your own code. All of the examples utilize an

RSA cipher constructed with the following values (also provided in the file `rsaExampleValues.txt`):

```
n = 1093733205710804998199890334068767166455920362828704765160762785
    7499483958630357076021298990643044443078830573207441251383385554
    0595306533636002574137640210725086799272942906615619662751138490
    7081299171319768555238149258992942399208388665959233718761302432
    26847351820673092753217938229621222197931912046784793
e = 157307166947463324293191
d = 5051344722702808401081282712266153060988904798305842766107188088
    5908654445646155458691942425898014706411932503960880426195998687
    5931918963365832849565604072385256625773752537810929752797947864
    1740341834159319848666223848546326674700490685819268998953869293
    3170796325885997807827620143533305258355518889959511
```

A.1 Example 1

Our first example is encrypting the plaintext message “Inheritance is cool!” using the RSA cipher constructed above.

After encrypting the plaintext using the RSA public key, the following is the expected ciphertext (in hex), which might not be readable otherwise.

```
b33b 9c60 2b50 d431 9f4a 91b3 1534 45af
4554 3962 442c f100 af94 5236 198b 7b6a
27b2 5ab5 0c21 0677 4dee 2740 c2f8 f4f2
4c66 aa59 8d8a 6a56 8a89 4bfd b644 3774
7367 047b 8faa 6171 11da 1287 e89e c891
ca03 a8e2 a5f2 6443 a49f 10a3 9cd4 27b9
2487 7f3b 5a38 2f7c 203f 642a 1c9a 8339
eb00 3735 5499 fd1d 4d41 f33c 9bdb 6e7b
```

If your RSA cipher is working correctly, but does *not* switch endianness during encryption, you would see the following output after encryption (in hex). If this is your output, make sure you are switching endianness correctly before creating a `BigInteger` object.

```
33c3 faab d960 c4c3 e9be 9515 9cec 6333
3b10 dd50 84c4 254c cbc8 329a bfe0 821d
139f 2911 0723 9765 3e96 b527 9a83 0ee4
8821 0e64 8ff4 e994 da2b 63b1 816d 364a
10df 6d4f b270 737c e103 b64a 2d2a 4412
0779 ca87 4442 68b4 80f2 2b8f d181 2571
5fee cad0 32d5 1db9 c189 aa98 8a58 8c1f
d7b7 fc39 8d7d 23cd 47f9 4aee 5bc7 fa69
```

Decrypting the correct ciphertext should yield the original message, “Inheritance is cool!”.

If you had the correct encrypted message, but you do *not* switch endianness during decryption, you would see this output after decryption: “!looc si ecnatirehnl”, the original message

backwards. If this is your output, make sure you are switching endianness correctly after reading bytes from a `BigInteger` object.

A.2 Example 2

Next, consider encrypting and decrypting the plaintext found in the file `rsaExample2.txt`. After encrypting this plaintext using the given RSA public key, the expected ciphertext (in hex) can be found in the file `rsaExample2Hex.txt`.

The given ciphertext should result in 6 chunks being created, and the last chunk should have size 75. If your RSA correctly encrypted and decrypted the previous example but is failing this example, ensure the correctness of your `ChunkReader` independent of the encryption process and pay special attention to the order in which you are performing operations on your data bytes. The order should be symmetric for the encryption and decryption steps.

A.3 Example 3

Consider encrypting and decrypting the plaintext message “I love CS2112” using the RSA cipher constructed above. After encrypting the plaintext using the above RSA public key, the following is the expected ciphertext (in hex), which might not be readable otherwise.

```
d6db 6e83 4048 c291 593f 6ea9 69ba 30c2
c6a1 d152 37f4 51e9 684a 32e2 ea07 a9e6
7d35 27bc 41a7 f58f ed4c 0ac4 afa5 f06b
3270 6c3b c730 0772 3c23 c795 a3a2 bfae
1ac8 5188 9cc8 319c fb37 b9d9 75b3 4e82
719e 6892 8171 fac8 2885 ce54 88dd 4d83
e663 6ee9 9dc5 618e 4728 aa69 ea0c ac1d
e89e 4d81 5b9c 9443 f399 42aa 9b8e d996
```

If your output differs from above after either encryption or decryption, but your output was correct for the previous two examples, something else is likely going wrong. In this case, try debugging with assert statements. Insert asserts for all your assumptions throughout your code—even ones that seem obvious or trivial!