CS 2112 Fall 2019
Assignment 2
Ciphers and Encryption

Due: Monday, September 23, 11:59PM
Design Document due: Monday, September 16, 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 alphabetic substitution ciphers, while the second part explores the widely used RSA public-key encryption algorithm. You will create 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.

1 Updates

- None yet; watch this space!

2 Instructions

2.1 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. Use brief but mnemonic variables names. Spacing and indentation should be consistent. Your code should include comments as necessary to convey the programmer's intent without belaboring the obvious.

2.2 Partners

You may choose to work alone or with a partner of your choice for this assignment. If you choose to work with a partner, one partner should invite the other partner to a group on CMSX. The other partner needs to accept the invitation before the assignment is submitted.

2.3 Getting help

The course staff is ready to help with any problems you run into. Use Piazza for questions or attend office hours with any course staff member for help. Office hours are on the web.

2.4 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.

2.5 Provided interfaces

You must implement all methods provided, even if you do not use them. **Important**: You may not change the signature (name, number and types of parameters) or the return type of any provided method. You may add `throws` declarations to methods if you believe they improve your design; however, make sure all thrown exceptions are caught and handled, and justify any such changes you make. You may (and are encouraged to) add as many additional classes and methods as you need.

2.6 Time management

This assignment is more involved than the last. Unlike the previous assignment, we have left the design of the program largely to you. Get started early, because it will require careful thought. Trying to do it all at the last minute will almost certainly result in disaster. In particular, it will be difficult to debug the RSA cipher, so it will be crucial to get your code as right as possible from the start. Don't expect to be able to incrementally debug the RSA cipher code into correctness.

3 Design document

Many components of this assignment share common functionality. For example, many ciphers need to read input 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 well designed program exploits inheritance by collecting common code shared by individual components in a single place.

To ensure that your design is reasonable, we require that you submit a design document before the assignment is due. The design document should at least contain a diagram of the type hierarchy you are planning to implement, along with a paragraph briefly justifying your design decisions. When designing the type hierarchy, try to think of how to eliminate redundant code by factoring out common functionality shared by multiple ciphers. Figure 1 shows an example of a type diagram in which the nodes represent classes or interfaces and the edges represent subtype relationships.

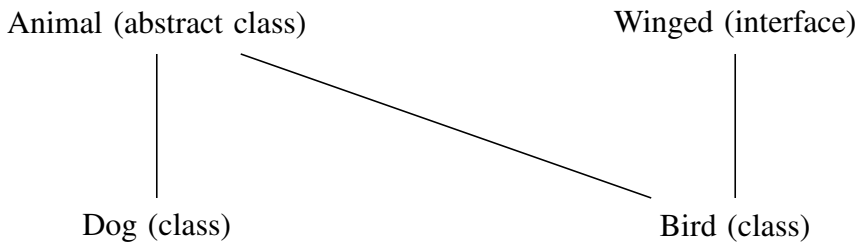


Figure 1: An example of a type diagram. `Dog`, which is a class, extends `Animal`, which is an abstract class. The class `Bird` extends `Animal` and implements the interface `Winged`.

Submit your design document as a PDF file named `A2DesignDocument.pdf` to CMS. Scans of handwritten diagrams are acceptable. The staff will attempt to give you quick feedback.

4 Substitution ciphers

4.1 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 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.

In this part of the assignment, you will implement two monoalphabetic substitution ciphers: a **Caesar cipher** and a **random substitution cipher**. You will also implement a simple polyalphabetic cipher, the **Vigenère cipher**.

4.1.1 The Caesar cipher

The Caesar cipher maps an alphabet to a shifted version of itself. For example, a cipher where each letter is shifted to the right by one (`A`→`B`, `B`→`C`, ..., `Z`→`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 through 26; they can be any integer, including negative numbers. A shift of −1 maps `A`→`Z`, `B`→`A`, `C`→`B`, etc. This method of encryption was used by Julius Caesar for military communications.

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.

4.1.2 Random substitution ciphers

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 class `java.util.Random`, are not truly random but only **pseudorandom**. They are produced by an algorithm called a **pseudorandom number generator** 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.

4.1.3 The Vigenère cipher

Another way to strengthen the Caesar cipher is to use different substitutions depending on the position of the letter in the text. The Vigenère cipher[1] was once considered to be unbreakable. Rather than using the same shift for all letters, a repeating pattern of shifts is used. 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.

The example below shows the Vigenère encryption of a quotation from Shakespeare using the key `ABC`. The top row is the plaintext, after converting to upper case and deleting all non-alphabetic, non-whitespace characters; the middle row is the repeated pattern of shifts; and the bottom row is the resulting ciphertext. Pay particular attention to how whitespace is handled.

```
TOMORROW AND TOMORROW AND TOMORROW CREEPS IN THIS PETTY PACE FROM DAY TO DAY
ABCABCAB CAB CABCABCA BCA BCABCABC ABCABC AB CABC ABCAB CABC ABCA BCA BC ABC
UQPPTUPY DOF WPORSTRX CQE VRNQUSQZ DTHFRV JP WIKV QGWUA SBEH GTRN FDZ VR ECB
```

4.2 Implementation

Your task is to implement the Caesar cipher, the random substitution cipher, and the Vigenère cipher. The interface `Cipher` defines the methods needed for encryption and decryption. Your ciphers should implement this interface and extend the abstract class `AbstractCipher`.

Your cipher implementations will be created and accessed through class `CipherFactory`. This is an example of the **Factory design pattern**. Design patterns are common patterns that arise so often in programming that they have been identified and given a name. They are good to know about because they can help you organize your thoughts about software design. We will learn more about different design patterns later in the course.

In this assignment, we are looking for elegant program design that **minimizes the repetition of common code**. The best program is one that accomplishes the task simply and efficiently with the least amount of code. You will find inheritance a valuable language feature for avoiding repetition. Abstract classes may further help to achieve this goal.

4.2.1 Letter encoding

A consistent standard is important for representing characters during both the encryption and decryption. All letters should be converted to their **lowercase** 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 when decrypting that the program will encounter only lowercase letters and whitespace characters.

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

4.2.2 Saving the cipher

You will also need to save your ciphers to a file at the user's request.

For a Caesar or random substitution cipher, print the word `MONO`, a newline, and the entire encrypted alphabet to the file, followed by a final newline. For example, saving a Caesar cipher with shift parameter 1 should create a file whose contents are as follows:

```
MONO
BCDEFGHIJKLMNOPQRSTUVWXYZA
```

To save a Vigenère cipher, print the word `VIGENERE`, a newline, and the key to the file, followed by a final newline. For example, a file containing a Vigenère cipher with the key "`KEY`" should read:

```
VIGENERE
KEY
```

Don't forget the newlines at the end.

5 RSA encryption

RSA[2] 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, but knowledge of the private key is required for decryption. 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 fact that there is no known efficient algorithm for factoring large numbers. Deriving a private key from the corresponding public key appears to be as hard as factoring.

5.0.1 Text and binary

There is one crucial difference between the substitution ciphers and RSA: with the substitution ciphers, the plaintext and the ciphertext are text, whereas with RSA, they are binary. The conversion between plaintext and ciphertext is done directly on the binary data and involves arithmetic on large numbers. Any text to be encoded using RSA must first be converted to binary data using some character encoding.

Recall that a **character encoding** determines how each character is represented in memory as a sequence of bytes. Please review the handout on Java I/O to remind yourself about character encodings and how they work. For more detail, see also this article and this article.

[2] Named for its inventors, Ronald Rivest, Adi Shamir, and Leonard Adelman.

The most common character encodings in use today are ISO-8859-1 (also known as Latin1), UTF-8, and UTF-16. The world seems to be slowly but surely converging on UTF-8 as a de facto standard, and that is what we will use in this assignment.

For the substitution ciphers, the character encoding does not matter, because the plaintext and ciphertext are both text, and the internal representation is irrelevant. But for RSA, it does matter, because different character encodings will give different ciphertexts. Thus to ensure uniformity and testability, we will agree on a fixed character encoding, which for this assignment will be UTF-8. You must specify this encoding explicitly for any conversion between characters and bytes you do in the context of RSA encoding or decoding.

5.1 The algorithm

We can describe the RSA algorithm at a high level in terms of **modular arithmetic**. If you are unsure what that is, please read this Wikipedia article.

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. This is the number of positive integers less than n that are relatively prime to n.[3] For a product of primes $n = pq$, 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 $\varphi(n)$.
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 $1 \equiv ed \bmod \varphi(n)$. The value of d can be computed from e and $\varphi(n)$ using the extended Euclidean 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, given in the form of a number, 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.

[3] We say that m is **relatively prime to** n, or that m and n are **relatively prime**, if the greatest common divisor (gcd) of m and n is 1; that is, if m and n have no nontrivial common factors.

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 chance this happens by accident is 1 in $pq/(p + q - 1)$, which is negligible for large primes p and q.

5.2 Implementation

5.2.1 Dealing with very large numbers

RSA involves very 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 with extremely high likelihood. Given a high enough value of `certainty`, this is good enough. You will want to choose a bit length (the `bitlength` parameter) for p and q such that the bit representation of the product $n = pq$ has no more than $1023 = 8 \cdot 128 - 1$ bits and no fewer than $1017 = 8 \cdot 127 + 1$ bits. That is, the product $n = pq$ must fall in the range $2^{1016} \le n \le 2^{1023} - 1$. The product of any two k-bit positive integers always has either $2k - 1$ or $2k$ bits.

The reason for these bounds is that you will be applying the RSA algorithm to numbers of 127 bytes ($8 \cdot 127$ bits), and n must be larger than that, so n must have at least $8 \cdot 127 + 1$ bits; and the result must fit in 128 bytes ($8 * 128$ bits), and the highest order bit, which is the sign bit, must be 0, so n must have at most $8 \cdot 128 - 1$ bits.

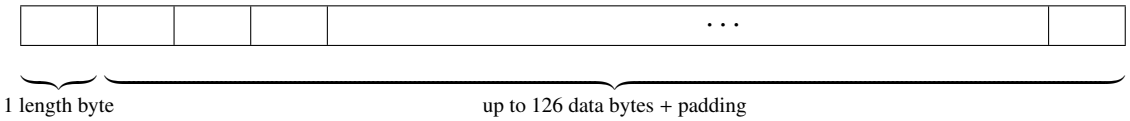
There are also tables of large primes available from The Primes Pages.

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 the `BigInteger` class. Rather, it is formatting the message so that it can be correctly encrypted and decrypted. The plaintext should be broken down into chunks of size 126 bytes by implementing the interface `ChunkReader`. Each chunk should then be used to construct the `BigInteger` object to which the RSA algorithm is applied. Note that there is no 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 byte representations.

Since the input length is unlikely to be an even multiple of 126, it is necessary for each chunk to keep track of the actual number of bytes of data contained in the chunk, so that it can be later decoded. This is done by adding a 127th byte at the front containing the number of actual data bytes in the chunk (from 1 to 126). If fewer than 126 data bytes are available, the data is padded out to 126 bytes with dummy bytes on the right. There are always 127 bytes in total that are converted to a `BigInteger`: up to 126 actual data bytes, the padding out to 126 if necessary, and one extra length byte containing the number of data bytes.

In picture form, the chunks look something like this:



For example, with 6 bytes of plaintext data, there will be $126 - 6 = 120$ bytes of padding on the right, and the first byte will contain 6 to signify the size of the actual payload.

For stronger encryption, the padding bytes added to short chunks could be random bytes, protecting against dictionary attacks. It doesn't matter what the padding bytes are, since the length byte tells us how many actual data bytes there are in the chunk. However, your program will be easier to debug if you set all such bytes to zero, and we will accept such a solution.

Fortuitously, because the length byte always contains a number less than 128, its sign bit (the leftmost or highest order bit) will always be 0, which means that the `BigInteger` encoding the 127-byte sequence will always be positive.

In general, encryption using the RSA algorithm can make the number larger or smaller, but it will never be larger than n because the arithmetic is modulo n. Thus the numbers generated by the encryption of each chunk will fit into 128 bytes. There is a slim chance that the encrypted chunk will be smaller than 128 bytes, in which case it will be necessary to pad with leading 0's. The resulting 128-byte arrays will be written to the output file. Thus the length of encrypted output will always be a multiple of 128 bytes.

When writing out the resulting ciphertext to a file, exactly the encrypted 128-byte chunks should be written, with no character conversion. Thus you should use `OutputStream` directly, perhaps wrapped in a `BufferedOutputStream` for efficiency. See the handout on Java I/O for examples.

We strongly suggest that you start with the code to read and write files using byte streams and to convert to and from chunks without any encryption. Get this working before you try implementing the RSA algorithm itself. This will help you catch bugs while the number of possible causes is still

small.

All data read from and written to files 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 some examples with the 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 `BigInteger`s and run backwards through the transformation. This should produce the chunk that was originally encoded along with its length byte. The length byte specifies how many bytes of data to extract from the remaining 126 bytes 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

RSA keys can be saved to files. When storing a key in a file, the file should contain the string representation of the public key e in decimal, followed by a newline, followed by the string representation of the private key d in decimal, followed by a newline, followed by the string representation of n in decimal, followed by a newline. That is, the file should look like this:

e
d
n

Don't forget the trailing newlines.

6 Command line invocation

In addition to the ciphers, you must create a command-line interface to allow users to interact with your cipher code. We have provided you with the structure to parse the command line arguments, but you must 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 keep the `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 should be able to provide commands of the following form via the console:

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

Cipher type We have asked you to implement four different cipher types, specified using the following flags:

- `--caesar <shift_param>`: Create a new Caesar cipher with the given integer shift parameter.

- `--random`: Create a new monoalphabetic substitution cipher with a randomly chosen permutation of the alphabet.
- `--vigenere <key>`: Create a new Vigenère cipher with the given keyword. The keyword is given as a string of maximum length 128 characters.
- `--rsa`: Create a new RSA cipher.
- `--monoLoad <cipher_file>`: Load a monoalphabetic substitution cipher (caesar or random) from the file specified.
- `--vigenereLoad <cipher_file>`: Load a Vigenère cipher from the file specified.
- `--rsaLoad <file>`: Create an RSA encrypter/decrypter from the public/private key pair stored in the file specified.

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

- `--em <message>`: Encrypt the given string using the specified cipher scheme.
- `--ef <file>`: Encrypt the contents of the specified file using the specified cipher scheme.
- `--dm <message>`: Decrypt the given string using the specified cipher scheme.
- `--df <file>`: Decrypt the contents of the specified file using the specified cipher scheme.

Output options Finally, the user may add as many of the following output flags as they wish.

- `--print`: Print the result of applying the cipher (if any) to the console.
- `--out <file>`: Print the result of applying the cipher (if any) to the specified file.
- `--save <file>`: Save the current cipher to the specified 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 a monoalphabetic substitution cipher from file `ca15`, use it to decrypt the message in file `encr.txt`, and print the result to the console:

```
java -jar <your_jar> --monoLoad ca15 --df encr.txt --print
```

- Create an RSA encrypter/decrypter, use it to encrypt the given message, save the ciphertext to a file `encr.txt`, and save the key in a file `mykey.txt`:

```
java -jar <your_jar> --rsa --em "rsa␣is␣alright,␣i␣guess"
    --out encr.txt --save mykey.txt
```

- Load an RSA key from a file `mykey.txt`, use it to decrypt the ciphertext in the file `encr.txt`, print the resulting plaintext, and also save the plaintext to a file `decr.txt`:

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

6.2 Error handling

Not all combinations of command line arguments make sense. For example, the following combinations would not make sense:

```
--rsa --dm <message>
--rsa --ef <file> --print
```

Other things could go wrong: missing parameters, requested files that do not exist, malformed requests, etc. In such cases, your program should should handle the error gracefully; no uncaught Java exception should ever be thrown by your program. Your program should detect user errors and find a sensible way to resolve or communicate the problem to the user. For example, if a user attempts to execute incompatible actions such as `--rsa --dm <message>`, it would be reasonable for your program to print a helpful error message to the console (`System.out`) and then to ignore the command.

7 Tips and tricks

This assignment is much more involved than Assignment 1 and demands more careful design. Starting early is essential. Trying to do it all at the last minute will certainly result in messy and broken code. Par for this assignment is about 900 lines of code, assuming you design a reasonable type hierarchy that allows for effective code reuse. The RSA cipher is the most challenging part of the assignment but is not worth more than the other ciphers, so use your time wisely.

Debugging can also be very challenging, 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 very carefully. Try not only expected inputs but also corner cases. The examples provided in Appendix A will be very helpful for testing your code.

7.1 Useful resources

Familiarize yourself thoroughly with the following Java classes. You may find some methods there that can save you some coding.

- `java.math.BigInteger`
- `java.util.Collections`
- `java.lang.String`
- `java.lang.Byte`

Consult the handout on Java I/O for information on character and byte streams and the relevant Java classes for handling them. The following Wikipedia articles also provide useful information:

- Wikipedia article on RSA
- Wikipedia article on Factory design pattern

7.2 Encrypting and decrypting large files

Your implementation of the various ciphers should not crash when attempting to encrypt or decrypt very large files. In particular, **do not assume that you can read the entire file into memory all at once**. We will test your code on files that are larger than the Java heap space. Think in terms of streams. The encryption and decryption processes should be piecewise; that is,

read, process, write; read, process, write; read, process, write,

rather than

read, read, read; process, process, process; write, write, write.

7.3 Assertions

Getting RSA to work can be the most challenging part of the assignment if you are not careful with design, testing, and debugging. Many of the problems come 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** as "sanity checks" at various points in your program to help you 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 Assertions are disabled by default, 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.

You can turn on assertion checking by default in Eclipse. Go to Eclipse > Preferences... > Java > Installed JREs, select your default installation, click Edit..., and enter `-ea` in the "Default VM arguments" field.

7.4 Build and test incrementally

When building large programs, it is helpful to test your code as you go. Implement little pieces and test them thoroughly 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 should have some incomplete but correct code that compiles and runs.

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.

7.5 Error handling

The various ciphers that you build should not print any error messages themselves, but should only throw exceptions in error situations. Think of them as library functions that clients would call to perform encryption and decryption as a service. Let the client (in your case, the main method) catch any exceptions and print an appropriate error message. For the purposes of grading, however, your main method should not throw any exceptions and must handle all errors gracefully.

7.6 Read specs carefully

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

Sign The constructor `BigInteger(byte[] val)` 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. Bytes in the range 0x80 to 0xFF, viewed as decimal numbers, will appear as negative values in the range −128 to −1. However, if you implement RSA correctly, you should not have to worry about this, because no `BigInteger` created by the algorithm should ever be negative.

Character encoding You must use the UTF-8 character encoding for this assignment for any character conversion you do; 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 `"UTF-8"` as an argument to certain `String` methods and constructors to specify that you want to use this encoding.

7.7 Hex viewers and editors

For debugging purposes, it may be helpful to know what is really stored in a file. Viewing a file using a text editor is useless when you are dealing with binary data. On Linux and Mac, any of these commands will show you the contents of the file `output.txt` in hex:

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

On Windows there are various hex viewers and editors available: HexView, HxD, and HexEdit are some. There is also an Eclipse plug-in EHEP that provides hex editor support. These viewers and editors display data in hexadecimal (base 16), whose digits are 0123456789ABCDEF. A single byte of data consists of 8 bits or two hex digits and ranges from 0 to FF in hex (0 to 255 in unsigned decimal, −128 to 127 in signed decimal, 00000000 to 11111111 in binary).

In your Java program, you can get a `String` representation of an integer n in hex using `Integer.toString(`n`,16)`. We have also provided a `Debug` class for displaying byte arrays in hex.

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. Please include your entire `src` folder inside the top level of your `.zip` with all internal package structure intact.
- `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. If you worked with a partner, their name and NetID should also be in here. It should also include descriptions of any extensions you implemented.

Do not include any `.class` files or any other extraneous files. 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 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 public and private key. These keys are also provided in the file `rsaExampleKeys.txt` in the format expected by the `--rsaLoad` command line option.

$$e = 32118814099656117913893317178942584748470526562483554331214998395141915568692406420480136833304197102423727151670984806543053627733061532174886246624634 39$$

$$d = 10800450706267955703896495854691126424866911092946722243265037164556931452167043114609427458150321487772671258566818439847827362717383660229277388471092208660561654623483216578271752146032160992330868438511339214513223343249920612371741848360251859155730557951813206286434192227873538990078392535728352777719$$

$$n = 24052027491043601143600897812895533897239476420770590988874009736480430122531667410157835814784041899559260252072127308366864065080815836668377597364492598595753259400407994945876045182908564410594733715261432842211228671984951399859748762379526456210501270993676460153192414940333805195641888001123051550339$$

A.1 Example 1

Our first example is encrypting the plaintext message "`Inheritance is cool!`" using the RSA cipher constructed with the above keys. After encrypting the plaintext using the RSA public key (e, n), the following is a hex dump of the expected ciphertext.

```
21 ca 90 aa 6d 51 55 7b 56 a0 6a c8 c0 15 dc 6c
4f 2d 47 b8 3c 8f f8 3a ee 38 f0 d8 5b bc 38 c0
85 59 74 c7 59 9a df 54 64 be 98 34 2d c1 a2 2d
98 23 16 2c f7 a0 f9 52 94 bd 0e f1 f1 5b f4 95
ac a8 ff 90 79 05 27 a7 e5 54 f1 b4 4b 06 21 71
77 71 2d 15 b9 dc 02 1d 9b 23 4d 74 a3 a9 1b 4b
52 90 f6 83 78 49 8f f7 32 58 02 ea 22 eb 9d 5a
b7 c3 d8 54 1a a0 57 e9 bb c6 50 e4 f4 4f 56 ef
```

Decrypting this using the private key (d, n) should yield the original message, "`Inheritance is cool!`". If your output differs from this after either encryption or decryption, try debugging with assert statements. Insert asserts for all your assumptions throughout your code—even ones that seem obvious or trivial.

A.2 Example 2

Now encrypt and decrypt the plaintext message "`I love CS2112`" using the RSA cipher constructed above. After encrypting with the public key (e, n), the following is a hex dump of the expected ciphertext. Decrypting this with the private key (d, n) should give the original message.

```
0d 1b 75 85 6d 8a e7 2a e9 d1 20 2e 43 c8 10 7e
d6 e9 60 43 ef 6e 8c fc 89 5f ea 3c 3b 8d 35 88
a1 17 17 0b 61 49 d0 e9 9f 75 5f b4 c8 5d d5 70
7d 38 c9 5d b0 f9 d3 e7 ad 10 b3 c4 26 24 a7 40
56 8d 0e 5f 44 e3 24 82 c2 da 69 ca 8d 7a 39 2b
80 61 6b 5d 79 1f c1 7f 91 b9 06 fc 01 5c 8e 96
3d 35 bf db 88 11 1d 0e b2 97 8c 9a 9d 66 13 9e
df bf 57 b5 f0 e1 7e 6d ed 77 b2 e9 7f f3 86 f6
```

A.3 Example 3

Finally, encrypt and decrypt the plaintext found in the file `rsaExample3.txt`. After encrypting this plaintext using the given RSA public key, the expected ciphertext can be found in the file `rsaExample3Hex`.

The given ciphertext should result in seven chunks being created, the first six having size 126 and the last size 2. If your RSA correctly encrypted and decrypted the previous examples 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.

B Cryptanalysis

This is not an official part of the assignment. No extra credit will be given, but you are welcome to give it a try just for fun and good karma.

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.

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. Do this by completing the public methods provided in class `FrequencyAnalyzer`. Like the encrypter, `FrequencyAnalyzer` should keep track of only lowercase 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.

Document anything you do that goes beyond what is requested in your `README.txt` file. Be especially careful that any extensions you make do not break any required functionality.