

INTRODUCTION TO COMPUTER SCIENCE II

PROJECT 3

A Substitution Cipher

1 Encryption

Taking data—on computers, a sequence of bytes or characters in a well-known format, known as a *cleartext*—and *encrypting* it involves re-encoding that data such that all the information remains in the encrypted version (the *ciphertext*) is impenetrable to someone who does not know how to read that new encoding. Some forms of encryption (*ciphers*) are harder to figure out how to read than others, but they all represent this same approach of *encrypting a cleartext to produce a ciphertext* such that one can later *decrypt the ciphertext back into the original cleartext*.

The history of encrypting data is long and the subject of many books and courses. Here, we are going to play around with old and simple ciphers to see how they work.

1.1 Caesar Cipher

One of the oldest known ciphers was apparently used by Julius Caesar, which is why we know it as the Caesar cipher. In order to protect the messages that he sent out into the world (where the carrier and thus the message might be intercepted), he would write his messages using a *rotated alphabet*. That is, he would take the original message (the *cleartext*), and then re-write the message by replacing each letter in the message with whichever letter was *3 positions forward* in the alphabet to produce the *ciphertext*. That is, where the letter A appeared in the cleartext, it would be replaced with D; B would be replaced with E; and so on. The rotation of the alphabet would *wrap around* to the beginning, so that the table of replacements looked like this:

Cleartext char	A	B	C	D	E	F	G	...	S	T	U	V	W	X	Y	Z
Ciphertext char	D	E	F	G	H	I	J	...	V	W	X	Y	Z	A	B	C

In order to decrypt such a ciphertext, the message was again rewritten, this time replacing each letter with the one that was *3 positions backwards* in the alphabet, thus recovering the original cleartext.

This approach worked well at the time because few people knew how to read, and the trick was sufficiently new or obscure that it befuddled anyone who intercepted a ciphertext. Of course, once someone knows the approach, then they are able to decrypt any message easily.

The method can be made better by rotating the alphabet by a different amount for different messages. This *encryption key* is a parameter to the encryption and decryption processes,

making for a different mapping between cleartext letters and ciphertext ones. Unfortunately, this approach is not much of an improvement: If I have a 26-character alphabet, then there are really only 25 possible values for the key. Trying all of them isn't difficult, making it easy to crack the code.

1.2 Substitution Cipher

A more substantial improvement on this idea is a generalized *substitution cipher*. Like the Caesar cipher, this cipher replaces each character in the cleartext with some other character to form the ciphertext. While the Caesar cipher chose the replacement characters by rotating the ciphertext alphabet, the substitution cipher uses a *randomly permuted ciphertext alphabet*. Thus, our map of cleartext to ciphertext letters looks like this:

Cleartext char	A	B	C	D	E	F	G	...	S	T	U	V	W	X	Y	Z
Ciphertext char	Q	D	A	E	H	Y	W	...	B	Z	P	I	M	S	U	V

The permutation cannot be purely random, because that same permutation must be used for both encryption and decryption; that is, we must have some way to recover the mapping used for encryption in order to later decrypt. The permutation must be determined in part by the *key* k that is chosen to create the random permutation of the ciphertext alphabet. Specifically, one must use a *pseudorandom number generator (PRNG)*, where the *seed* for that algorithm is k , in order to generate the same permutation twice.¹

2 Getting started

Get the code: Start a *Terminal*. Then, grab some starting source code and open it:

```
$ cd
$ curl -L https://bit.ly/cosc-112-24f-project-3 -o project-3.zip
$ unzip project-3.zip
$ cd project-3
$ code .
```

The source includes the following files:

¹Keep in mind the number of possible permutations. For the 26 uppercase letters, there are $26! \approx 4 \times 10^{26}$ permutations; for the complete set of 256 `char` values, there are $256! \approx 8 \times 10^{506}$ permutations. Thus, even if we use an `long` for the seed/key, then we can specify at most $2^{64} \approx 1.6 \times 10^{19}$ different key values, and thus only a small fraction of the possible set of permutations. For our purposes, that's enough, but for a real implementation of this cipher, a much larger range of key values should be used. A question for the curious: How many bits should we use for the key to ensure that we can specify at least 256! key/seed values?

- `Crypt.java`: The class that contains `main()` and its helper methods. This is the class you will run when invoking the program. It reads the input data, uses the requested cipher to perform the requested operation (encryption or decryption), and writes the result. More on the use of this class below.
- `Cipher.java`: An abstract class that defines how a specific cipher must be implemented as a subclass. It holds the secret key value used by any cipher, but it leaves the encryption and decryption methods abstract.
- `CaesarCipher.java`: An example subclass of `Cipher`. It implements the Caesar cipher. More importantly, it provides a template for the class you must write. More on that in Section 3.
- `NiceList.java`: The familiar `NiceList` interface.
- `NiceArrayList.java`: A working `NiceList` implementation for you to test things out. More below on using your `NiceLinkedList` instead, but this one is always here for debugging purposes.

If you try to compile this code, you will discover that two missing files prevent it from compiling:

- `NiceLinkedList.java`: You should copy yours (along with `NiceLink.java`, and the sentinel classes) from Lab-6 into this directory.
- `SubstitutionCipher.java`: The class you must write, as a subclass of `Cipher`, that implements a substitution cipher.

Once you add these files, you can compile the code (even if the `SubstitutionCipher` isn't complete).

Choosing a `NiceList` implementation: Although you should use your `NiceLinkedList`, you may temporarily want to use `NiceArrayList`, just in case you are not entirely certain that your `NiceLinkedList` is working properly. To do so, notice that in `Cipher.java`, there is a special method at the bottom:

```
public static NiceList<Character> createList ()
```

This method creates and returns a new, empty `NiceList` of `Character` (which is what this code needs throughout). Specifically, its one-line body allows you to specify **which implementation** of this interface to create. Currently, it makes and returns a `NiceLinkedList`, but you may change it to create a `NiceArrayList`. Throughout the code, calls to `Cipher.createList()` are used to create new `NiceLists` wherever they are needed, this centralizing the control of which kind to make.

Running the program: Once you have compiled, you can try running the program. If you run the `Crypt` program with no arguments, you will see this usage message:

```
$ java Crypt
USAGE: java Crypt <cipher [Caesar|Substitution]> <operation [encrypt|decrypt] <key>
```

The user must choose: which cipher to use (`Caesar` or `Substitution`); which operation to perform (`encrypt` or `decrypt`); and, the key used for that operation. However, notice that no file names are expected. So how do you specify a file to encrypt, and where does the encrypted result go?

This program uses the *standard input* and *standard output* for reading and writing, respectively, of the cleartext and ciphertext. That is, the input is read from the console (you could type it), and the output is written to the console (you could see it printed). These channels for input and output are the defaults for any program we run, and are often abbreviated as *stdin* and *stdout*. You likely know them better, in Java, as `System.in` and `System.out`.

We do not really want to type in the input to this program, nor do we merely want to see the output appear in the console window. Luckily, at the command line, we can direct the program to use a file of our choosing as the standard input, and likewise use some file as the standard output. This trick is known as *redirection*, and uses some special symbols on the command line to make it happen. Specifically, the *less-than* (<) symbol is used for *input redirection*, and the *greater-than* (>) symbol for *output redirection*. Using them would look like this:

```
$ java Crypt Caesar encrypt 42 < original.txt > encrypted.txt
```

Here, our program would perform a Caesar cipher encryption, using the key value of 42, on the data read from the file, `original.txt`. The result would then be written into the file, `encrypted.txt`.

When decrypting, the role of ciphertext and cleartext reverse:

```
$ java Crypt Caesar decrypt 42 < encrypted.txt > decrypted.txt
```

Notice that the contents of `decrypted.txt` should exactly match the contents of `original.txt`. You can, in fact, use the `diff` command to compare the two files automatically:

```
$ diff original.txt decrypted.txt
```

If the files match exactly, then no output is generated. If there are differences, the `diff` command will print them. No news is good news.

3 Your assignment

Write the `SubstitutionCipher` class and test that it works. You should use a `Random` object to randomly permute the alphabet of character values from 0 to 255. The key should be used to set the `Random` seed, allowing any given permutation to be recreated. See the Java 21 API for more information on `Random` objects.

4 How to submit your work

Submit **all of your .java files** by uploading it into the `project-3` folder in your shared Google Drive folder for this course.

This assignment is due on Sunday, Dec-08, 11:59 pm.