INTRODUCTION TO COMPUTER SCIENCE I
PROJECT 4A
Shift Cipher

Project 4 is a three-part, two-week assignment that will exercise our new-found abilities with the `char` data type and *arrays*. Specifically, we are going to delve into the world of *cryptography*, writing programs that implement two different *ciphers*, and, in the end, performing a bit of *cryptanalysis* (code breaking). In doing so, we will need to manipulate and use arrays of characters in many different ways.

1 Cryptography

Cryptography is, loosely speaking, the study of methods for taking "human readable" data (*cleartext* or *plaintext*) and then scrambling (*encrypting*) it into an obfuscated form (*ciphertext*) that cannot easily be read. Of course, such obfuscation is useless unless there is some means by which the ciphertext can be restored (*decrypted*) to its original cleartext form.

A *cipher* defines a *one-to-one correspondence* between plaintexts to ciphertexts. That is, it defines a pair of algorithms, one for encryption and one for decryption, that respectively scramble and unscramble data. Each cipher uses a single parameter called the *key* that is used as part of the encryption and decryption calculations. For a given ciphertext, the same key must be used to decrypt it as was used when creating it during encryption.[1]

The ciphers that we are going to use are *monoalphabetic substitution ciphers*. Specifically, for these ciphers, encryption is accomplished by replacing each character in the plaintext with a different letter in the ciphertext. Therefore, the lengths of the plaintext and the ciphertext are equal, and both encryption and decryption can each be performed one character at a time.

2 The Caesar Cipher

For this first part of this project, we will use the *Caesar cipher* (also known as a *shift cipher*). To encrypt a message using this cipher, each character of the cleartext is *shifted* forward by k characters in the alphabet, thus producing a ciphertext. Likewise, decrypting a message requires shifting each character in the ciphertext backwards by the same k characters in the alphabet. Moreover, since the alphabet of characters is a fixed-sized list, it is possible to shift "off the end" of the list. Whenever this situation arises, the *shift* operation "wraps around" to the beginning of the alphabet.

To see how this cipher works, consider only the 26 uppercase letters of the English alphabet. Furthermore, consider using the key value $k = 5$. The last 5 characters in this alphabet, when shifted, are wrapped-around to the first five characters. So, the correspondence between the cleartext and ciphertext characters would be:

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

[1]The use of the same key value for encryption and decryption makes a cipher *symmetric*. There are *asymmetric* ciphers, most notably public-key cryptography, where different keys are used for encryption and decryption, but those are beyond the scope of this course. Consider the Networks and Cryptography course.

Thus, the following message, if encrypted using the same key of $k = 5$ would be:

Cleartext message	`THEQUICKBROWNFOX`
Ciphertext message	`YMJVZNHPGWTBSKTC`

Implementation hints: In order to write a program that performs this type of encryption and decryption, there are two valuable observations that may help:

1. **A `char` is an integer in disguise:** Since each character is really a numeric value, then you can perform all of the common arithmetic operations on them. You can add k to shift forward for encryption, and you can subtract k to reverse the shift for decryption.

2. **There are 256 possible `char` values:** Each `char` datum is a numeric value between 0 and 255 (inclusive). If, by shifting a character (adding or subtracting k) you obtain a result that is outside this range, then you can use the *modulus* operator to appropriately "wrap around" the result into the correct range. That is, in Java, `x % y` yields the remainder after dividing `x` by `y`. So, if you take a value *mod 256*, then the remainder after dividing any value by 256 must be between 0 and 255.

3 Your tasks

Now that you know how the Caesar cipher works, you will need some code with which to get started. Follow these steps to obtain Java code that leaves only the interesting part—the encryption and decryption tasks—for you to write:

1. Login to remus/romulus.

2. Create and change into a directory for this project:

```
$ mkdir project-4a
$ cd project-4a
```

3. Copy a few files from a directory of mine:

```
$ cp ~sfkaplan/public/COSC-111/project-4a/* .
```

4. Check that you have all four files:

```
$ ls
CaesarDecrypt.java  CaesarEncrypt.java
project-4a.ciphertext  Tools.class
```

Write the encryptor: Use *Emacs* to open `CaesarEncrypt.java`. First, you will see that `main()` is already written. **Do not make any modifications to `main()`**; it is constructed to use the already-compiled code in `Tools.class` to read and write the cleartext and ciphertext files.

The part on which you must focus your attention is the `encrypt()` method, whose body is initially empty. Specifically, note the first line that declares the method's name, return type, and parameter list—it's *signature*:

```
public static char[] encrypt (char[] cleartext, int key) {
```

This method is passed an already-constructed array of characters, `cleartext`. As the parameter name suggests, this variable points to an array that contains the sequence of characters that is the cleartext message. Additionally, the `key` parameter is an integer that specifies by how many positions in the alphabet each cleartext character should be shifted to produce its ciphertext substitute—that is, it is the value k described in Section 2.

This method must create a new array of `char` of the same length as the cleartext that will hold the ciphertext. Each element of the cleartext array should be copied into the same position in the ciphertext array, except that the character value inserted into the ciphertext array must be the shifted equivalent of the corresponding cleartext character. When every cleartext character has been copied and shifted into the ciphertext array, then a pointer to that new array must be returned to the caller.

Write the decryptor: Repeat the above process. Open `CaesarDecrypt.java` with *Emacs*. The `main()` method in this program will be similarly structured to the one from `CaesarEncrypt.java`, except with the roles of cleartext and ciphertext inverted. Again, you should leave unchanged the code provided in `main()` that reads and writes the ciphertext and cleartext files.

Of course your task for this decryptor program is to fill in the body of the `decrypt()` method. It's signature is analogous to the signature for `encrypt()`, except with the roles of ciphertext and cleartext inverted. That is, this method is passed an array that contains the ciphertext, as well as a key k by which to reverse the shift of each character. The method must return a newly constructed array that contains the recovered cleartext.

Testing your encryption/decryption: Once you've written encryption and decryption programs that at compile, and that you suspect may just possibly work, then you need to test your implementation of this cipher. To do so, follow these steps:

1. **Create a sample cleartext file:** Use *Emacs* to open a new file[2] into which you can write whatever you like. I suggest that you write a short, simple cleartext message for yourself:

```
$ emacs my-message.cleartext &
```

[2]What you choose to name this new file is immaterial. The *suffix*—the portion of the name that follows the dot (`.`)—means nothing to *Emacs*, nor to your encryption and decryption programs. I suggest here, by example, a naming convention that simply helps us humans to keep the files and their uses straight.

2. **Encrypt:** Run your encryption program on your newly created cleartext file. For your tests, choose a key value $1 \leq k \leq 255$ to use for both encryption and decryption.[3] For the examples shown here, we will use a key value of $k = 8$. Invoke the encryption program as follows, and it will (if correctly written) produce a new ciphertext file:

   ```
   $ java CaesarEncrypt my-message.cleartext
                        my-message.ciphertext 8
   ```

   For the first time, we are providing *command-line arguments* to our programs. Specifically, we are specifying the name of the cleartext file, the name of a new ciphertext file, and the key value as part of the command that starts the program. You need not worry about how to make your program handle these arguments; `main()` is already written to handle this problem. What matters is that you specify these file names and the key value every time you encrypt (and decrypt).

   After the encryptor runs, you can use *Emacs* to open your ciphertext, `my-message.ciphertext`. However, what you see should be an unreadable bunch of junk. Certainly look at it, as a point of curiousity, but don't expect to discern much that is meaningful by examining the ciphertext directly in this fashion.

3. **Decrypt:** Now run your decryption program on your ciphertext file using the same k value. When running the decryptor, you must provide the name of the ciphertext file, the name of a new cleartext file, and the key value. For example:

   ```
   $ java CaesarDecrypt my-message.ciphertext
                        my-message.decrypted 8
   ```

   Here, the ciphertext is taken from the file `my-message.ciphertext`, it is decrypted with the key $k = 8$, and a new cleartext file named `my-message.decrypted` will be created and will store the decrypted cleartext. Notice that I chose **not** to re-use the original cleartext file name, `my-message.cleartext`. If I were to do so, then the decryptor will overwrite the original cleartext, eliminating the original data for comparison. By using a different file name for the decryption result, I will have two versions of the cleartext that I can compare.

4. **Compare:** For a small and simple message, you simply can open both `my-message.cleartext` and `my-message.decryption` with *Emacs* and compare the two. Of course, if they do not match, then one or both of your encryption and decryption code contains an error. For larger messages, exact comparison may be difficult and tedious. Thus, you can automate the comparison:

   ```
   $ cmp my-message.cleartext my-message.decrypted
   ```

[3] Of course, any integer value for k is legitimate. If $k = 0$, then the cleartext and ciphertext will be identical. If $k < 0$ or $k >= 256$, then k will produce the identical results to $k' = k \bmod 256$. For example, if $k = 258$, then the ciphertext produced will be identical to the one that $k = 2$ would yield.

The `cmp` (compare) command examines the two files for differences. If they are identical, then **no output appears**—you will just see your prompt reappear. If they files do not match, then the `cmp` program will indicate the first location at which the files differ.

Decrypting the secret message: One of the files that your copied from my directory is `project-4a.ciphertext`. If you open this file with *Emacs*, you will see that it is the same kind of unreadable goop as your own cipertext file. However, you now have a working decryptor program. Use it.

Notice that I have not specified the key. However, there are only 255 of them. Try each until you find one that works. You may attempt this brute force *attack* on the key by running the program many times, each with a different key value, or by adding some special-purpose code to your decryptor program. What matters is that you both decrypt the message, and that you discover and record the key used for that decryption. The decrypted message will itself provide instructions for moving onto part B of the assignment.

4 How to submit your work

When you have completed this part of the assignment, submit your encryptor, decryptor, and the decrypted secret message:

```
cs111-submit project-4a CaesarEncrypt.java
                        CaesarDecrypt.java
                        project-4a.cleartext
```

This assignment is due on October 11/12, at the start of lab