INTRODUCTION TO COMPUTER SCIENCE I
PROJECT 4B
Substitution Cipher, Take I

Congratulations! You've made it to part **b** of this project. Here, you will employ a completely different cipher that challenges some of your other programming skills.

1 The Substitution Cipher

For parts **b** and **c** of this project, we will use a different type of cipher known as a substitution cipher. Much like the Caesar/shift cipher that was used in part **a**, this cipher replaces each character in the cleartext with some corresponding character to form the ciphertext. However, with the Caesar/shift cipher, the relationship between a cleartext character and its ciphertext character was a simple, additive one. For this cipher, the ciphertext alphabet is **randomly ordered**. That is, if we were considering only the 26 uppercase letters of the English alphabet, then one possible correspondence could be:

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

That is, to form a ciphertext alphabet, we take the plaintext alphabet and *permute* its characters. The permutation cannot be purely random, because that same permutation must be used for both encryption and decryption. Thus, the permutation must be determined in part by the *key* k that is chosen for a given encryption/decryption. Specifically, one must use a *pseudorandom number generator (PRNG)*, where the *seed* for that algorithm is k, in order to gernerate the same permutation twice.[1]

We can create an array containing each of the characters. We may then use k as a seed to randomly permute those characters. Note also that once we create the randomly ordered array, we can use it to map cleartext to ciphertext characters and back again.

2 Pseudo-Random Number Generator basics

In order to create this random permutation of characters, you will need access to a pseudo-random number generator. Specifically, there are some "magic" lines of code that you must add to the beginning of your program, and then there are methods that you call to obtain a sequence of pseudo-random numbers.

[1] Keep 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 $2^{64} \approx 1.6x10^{19}$ 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?

Leading boilerplate: Recall that, when we want the user to be able to enter a value, we add lines at the beginning of our program that allow us to call methods like `keyboard.nextInt()`. Likewise, we must add two lines of code to the beginning of each program that will use a PRNG. These lines are:

1. `import java.util.Random;`

   This line must come **before** the one that names and begins the program (e.g., `public class SubEncrypt {`).

2. `public static Random random = new Random();`

   This odd line must come **after** the one that names and begins the program, but **not** within any method. Put it immediately after the name-and-begin-the-program line, after its opening brace.

Seeding the PRNG: A PRNG is not actually random—it uses a *seed* value to initiate the generation of a seemingly unpredictable sequence of numbers. However, if you seed the PRNG with the same value, it will then provide the exact same sequence of numbers—hardly *random* as we know it. However, this replicability of the pseudo-random sequence will be useful to us here, since both encrypting and decrypting will need to use identical random number sequences.

In order to set the seed of a PRNG, you must call a method, passing it that value:

```
random.setSeed(mySeedValue);
```

This method, each time it is called with a given value, resets the PRNG to a particular state that will then yield one particular sequence of random values.[2] Thus, the *key* used by your encryption/decryption algorithms can be used as this seed, thus allowing both encrypter and decrypter to obtain the same random number sequence when randomly permuting the character set. That is, if both encrypter and decrypter can seed the PRNG with the same value, then both can come up with the same jumbling of the alphabet.

Obtaining values from the PRNG: For this assignment, you will need a sequence of random values between 0 and 255.[3] In general, to choose a random value between 0 and $k - 1$ (inclusive), you may use the following method call:

```
int randomValue = random.nextInt(k);
```

Each call to this method will yield a different random value. Much like `keyboard.nextInt()`, each call requires that a new value be returned.

[2]The legnth of this random sequence, before it repeats, is exceedingly long. Even in real applications—as opposed to our simplified problems in this course—it is unlikely that anyone would request a long enough sequence of random numnbers to make the sequence begin to repeat.

[3]These values will be randomly chosen indices into an array that stores the possible characters that a message may use.

Conisder that your program set the seed value and then call `random.nextInt()` some number of times—let's say k calls to this method. If your program then sets the **same seed value**, and your program once again calls `random.nextInt()` another k times, then the same sequence of k values will be returned as the first k calls to `random.nextInt()`. This ability to generate some sequence of random values in an exactly replicable manner is going to be critical to creating our encrypter and decrypter.

3 Your tasks

Once again, you will need some code with which to get started. Follow these steps:

1. Login to `remus` or `romulus`.
2. Change into a your directory for this project:

   ```
   $ cd project-4
   ```

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

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

4. Check that you have these two new files:

   ```
   $ ls
   SubEncrypt.java SubDecrypt.java
   ```

Write the encryptor: Use *Emacs* to open `SubEncrypt.java`. Just as you did in working with `CaesarEncrypt.java`, you will see that `main()` is already written. Again, **do not make any modifications to `main()`**; it is constructed exactly as it should be.

The rest of this file contains the beginnings of two methods: `encrypt()` and `generateMap()`. Each of these contain small amounts of beginning code, along with comments, that strongly suggest how you should go about writing the remainder of these method bodies. In particular, the latter method should perform the task of creating and shuffling an array of all possible 256 `char` values; the former method should call the letter to obtain that random scrambling.

Write the decryptor: Repeat the above process. Open `SubDecrypt.java` with *Emacs*. Again, you will find a fully formed `main()`, along with another method (`decrypt()`. Having written the corresponding `encrypt()` method, you must now figure out how to reverse the process. That is, you must recover the scrambled correspondence of cleartext to ciphertext characters, and then transform each ciphertext character back to its cleartext original.

Testing your encryption/decryption: This process is the same as with part **a**. You must create a file with a cleartext message. Then, with some choice of key value, you encrypt and decrypt that message, ensuring that the decrypted result matches the original cleartext. The programs are run just as the `CaesarEncrypt` and `CaesarDecrypt` programs, where you provide the filenames for the cleartext and ciphertext, along with the key, as part of the command that runs each program.

Obtaining the next secret message: Once again, you must use your program to obtain a secret message. Here, however, a program of mine will use **your** encryptor to produce a ciphertext that you must then decrypt using the correct decryption key. Follow these steps to obtain the secret message:

1. **Obtain the ciphertext:** From within your `project-4` directory, which must contain your compiled `SubEncrypt` and `SubDecrypt` programs, use this command:

   ```
   $ secret-encrypt-4b.sh
   ```

   When this command has completed its work—it shouldn't take long—then you can use the `ls -l` command to see, in your directory, a new file named `project-4b.ciphertext`. Assuming that your `encrypt` method worked correctly, you should now have an encrypted secret message.

2. **Decrypt the ciphertext:** In completing part **a**, you obtained a few critical numbers. Particular, you have the two year numbers on the statue, as well as the key used to decrypt the secret message from part **a**. You must concatenate these three numbers into a single integer that forms the key for decrypting your new ciphertext. In particular, the message is the concatenation, in order, of: *the earlier year*; *the later year*; *the shift cipher key*.

   Use this newly cobbled together key along with `project-4b.ciphertext` on your `SubDecrypt` program to obtain `project-4b.cleartext`. This new cleartext message brings you to part **c** of this project.

4 How to submit your work

As usual, use the `cs111-submit` command:

```
cs111-submit project-4b SubEncrypt.java
                        SubDecrypt.java
                        project-4b.cleartext
```

Part B is due on Mar-27, at 11:59 pm