INTRODUCTION TO COMPUTER SCIENCE I
PROJECT 4C
Substitution Cipher, Take II

Onto the last step, where you will do a bit of *cryptanalysis*—code breaking!

1 Frequency Analysis

In part **a**, when you lacked a key for the Caesar cipher, you simply tried all of the possible keys. Since the number of keys was small, it was just a matter of time before this *brute force attack*—a simple form of *cryptanalysis*—was successful.

However, what about the substitution cipher that we used for Project 4b? Would a similar brute force attack, where each possible key was tried, work? In our implementation, the key k is represented by an `int`. Since each `int` is a 32-bit value, then there are $2^{32} = 4,294,967,296$ distinct keys to try.[1] This number is far too large for us to perform a brute-force attack of the keys, so we'll have to try something different.

Luckily for us, it seems that all of the encrypted messages that we are using are straightforward English text. Such a natural (human) language has tremendous regularities that we can exploit. Specifically, we know that the letters used in English text are not used with uniform frequency. Moreover, we know that the encryption method is simple substitution—that each cleartext character is replaced with the **same** ciphertext character throughout the message. By putting together these observations, we can *measure the frequency of each ciphertext character in order to determine which cleartext character it represents.*

That is, we know that the vowel *e* occurs most frequently in English text. Therefore, if we take a ciphertext created with a substitution cipher, and we find that the character *r* appears most frequently, then we can deduce that *r* is the ciphertext charcater substituted for all cleartext *e*'s. If we know the frequency of enough of the most frequently used English letters—where "enough" is, say, the set of characters that account for, say, $\frac{3}{4}$ of the text—then we can match the most frequently used ciphertext characters from an encrypted message to their cleartext counterparts. Doing so, and then *partially* decrypting the message by substituting our suspected cleartext characters for their ciphertext counterparts, should show us enough of the message to fill in most of the rest of the substitutions.[2]

An example: In order to see how this type of *frequency analysis* works, let's try a simple example. Assume here that we are using only the 26 uppercase alphabetic letters of the English alphabet for both our cleartext and ciphertext messages. Furthermore, consider the following ciphertext:[3]

[1] And even this set of combinations in restrictive. If we used more bits to represent the keys, we could have at least one key for each of the $256! \approx 8 \times 10^{506}$—a staggeringly large number.

[2] There are some subtitutions that we may never figure out, simply because those substitutions are for characters so infrequently used—perhaps not at all—that determining the cleartext-to-ciphertext correspondence is irrelevant for cracking the code and reading the encrypted message.

[3] I have broken the message across multiple lines so that it fits on the page. However, the *newline* character is not part of this alphabet, so the statistics presented below in the tables ignore these *newline* characters. That is, when you count the characters, just imagine both the ciphertext and the cleartext as a single line of text.

```
LRESLBBSYBYEXPBQFPKBETHQBPCB
FYKYLQRUFETHSYFYKYLQRUQXPCB
NFYTLIYBESSQBYERYHCXFYKYRYQFPCB
BLXRYPAYKETHXPCKRYPCB
```

Here is a table of the frequencies with which each ciphertext character occurs in the above ciphertext message:

character	**frequency**
A	$\frac{1}{107}$
B	$\frac{13}{107}$
C	$\frac{6}{107}$
D	$\frac{0}{107}$
E	$\frac{7}{107}$
F	$\frac{7}{107}$
G	$\frac{0}{107}$
H	$\frac{4}{107}$
I	$\frac{1}{107}$
J	$\frac{0}{107}$
K	$\frac{6}{107}$
L	$\frac{6}{107}$
M	$\frac{0}{107}$
N	$\frac{1}{107}$
O	$\frac{0}{107}$
P	$\frac{8}{107}$
Q	$\frac{7}{107}$
R	$\frac{7}{107}$
S	$\frac{5}{107}$
T	$\frac{4}{107}$
U	$\frac{2}{107}$
V	$\frac{0}{107}$
W	$\frac{0}{107}$
X	$\frac{5}{107}$
Y	$\frac{17}{107}$
Z	$\frac{0}{107}$

To make this table more useful, we reorder it by decreasing frequency:

character	frequency
Y	$\frac{17}{107}$
B	$\frac{13}{107}$
P	$\frac{8}{107}$
E	$\frac{7}{107}$
F	$\frac{7}{107}$
Q	$\frac{7}{107}$
R	$\frac{7}{107}$
C	$\frac{6}{107}$
K	$\frac{6}{107}$
L	$\frac{6}{107}$
S	$\frac{5}{107}$
X	$\frac{5}{107}$
H	$\frac{4}{107}$
T	$\frac{4}{107}$
U	$\frac{2}{107}$
A	$\frac{1}{107}$
I	$\frac{1}{107}$
N	$\frac{1}{107}$
D	$\frac{0}{107}$
G	$\frac{0}{107}$
J	$\frac{0}{107}$
M	$\frac{0}{107}$
O	$\frac{0}{107}$
V	$\frac{0}{107}$
W	$\frac{0}{107}$
Z	$\frac{0}{107}$

Finally, here is the table of frequencies for the cleartext characters used to form the original message, also sorted by decreasing frequency:

character	frequency
E	$\frac{17}{107}$
T	$\frac{13}{107}$
O	$\frac{8}{107}$
A	$\frac{7}{107}$
H	$\frac{7}{107}$
M	$\frac{7}{107}$
S	$\frac{7}{107}$
I	$\frac{6}{107}$
R	$\frac{6}{107}$
U	$\frac{6}{107}$
L	$\frac{5}{107}$
P	$\frac{5}{107}$
D	$\frac{4}{107}$
N	$\frac{4}{107}$
Y	$\frac{2}{107}$
G	$\frac{1}{107}$
V	$\frac{1}{107}$
W	$\frac{1}{107}$
B	$\frac{0}{107}$
C	$\frac{0}{107}$
F	$\frac{0}{107}$
J	$\frac{0}{107}$
K	$\frac{0}{107}$
Q	$\frac{0}{107}$
X	$\frac{0}{107}$
Z	$\frac{0}{107}$

By matching up these tables, we can substitute the cleartext characters for the ciphertext ones. For example, replace each `Y` with an `E`, each `B` with a `T`, and so on. When we do so, we will end up with the cleartext message:

```
IMALITTLETEAPOTSHORTANDSTOUT
HEREISMYHANDLEHEREISMYSPOUT
WHENIGETALLSTEAMEDUPHEREMESHOUT
TIPMEOVERANDPOURMEOUT
```

Warning: It is terribly important to observe that (a) these frequencies are **not** those of normal English text, and (b) this table of cleartext frequencies matches the ciphertext table of frequencies exactly, which is utterly unrealistic. In dealing with real ciphertexts, the frequencies of the most used characters will match reasonably well, but not exactly, and sometimes will be slightly out of order. Worse, this technique is harder to apply to short messages, where the frequencies don't match well because so few characters are used. However, for your use of this technique, the correspondence of frequencies are good enough to crack the message.

2 Your tasks

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

1. Login to remus/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-4c/* .
   ```

4. List your files, which should include:

   ```
   $ ls
   project-4c.ciphertext 4c-sample.cleartext ForcedSub.class
   MakeMap.class CountFrequency.java
   ```

Write the frequency counter: Use *Emacs* to open `CountFrequency.java`. As usual, `main()` is already written and should not be changed.

`CountFrequency.java` also contains the beginnings of two more methods: `count()` and `printSorted()`. Fill in the body of each of these methods, following the guidance given by the comments that begin `FIX ME`. The first method, `count()`, must determine how many times each character occurs in a given character array; the second method, `printSorted()`, must print the frequency of each character's use, by decreasing order of frequency. This latter method should call on the final, fully written method, `printLine()`, to do the actual printing of a given character and its frequency.

Use the frequency counter: Once you have written the frequency counter, you must use it to measure the frequencies with which characters occur in both a sample of plaintext and a ciphertext for which you have no key.[4] Specifically, `project-4c.ciphertext` is the ciphertext you must crack, and `4c-sample.cleartext` is meant to provide a baseline for frequency of character. So, to use your program on each:

```
$ java CountFrequency 4c-sample.cleartext 4c-sample.frequencies
$ java CountFrequency project-4c.ciphertext project-4c.frequencies
```

[4] To make matters easier, the sample cleartext and the ciphertext are largely identical, thus making the frequencies almost exactly match. In real cryptanalysis of this kind, a bit more effort is required to account for fluctuations in character use.

Go ahead and open these two new files, `project-4c.frequencies` and `4c-sample.frequencies`, to see which characters occur with what frequencies. Each file will look something like this:

```
32 (<space>) = 0.1833944648216072
116 (t) = 0.0713571190396799
105 (i) = 0.051683894631543846
110 (n) = 0.048349449816605536
10 (<line feed>) = 0.04701567189063021
```

Each line shows the frequency information for a single character through a given file. Specifically, the information shown is:

1. **Character number:** The first value is an integer that shows, in numeric form, the underlying value of a given character. For example, if the character in question is `A`, then the number shown here would be `65`.

2. **Character/character name/octal number:** Between parentheses, the actual character itself, as normally printed, although with a few exceptions. Some characters are *whitespace*, providing space between other, normal characters. For these—the *space*, *tab*, *line feed*, and *carriage return* characters—the name of that character is provided between angle-braces (<>). Other characters are unprintable *control characters*. These may appear with the carat prefix (e.g., `^G` for *control-G*), or as an octal value (e.g., `\211`, which is character number $235_8 = (2 \times 8^2) + (3 \times 8^1) + (5 \times 8^0) = 128 + 24 + 5 = 157_{10}$).

3. **Frequency:** Following the equals sign, the floating-point value provided is the frequency with which that character occurred in the input. These values are fractions of the total. That is, if the total number of characters in a file is $k = 1,000$, and of those, $k_A = 200$ is the number of occurrences of the character `A`, then the frequency shown for that character is $\frac{k_A}{k} = \frac{250}{1,000} = 0.25$.

You will notice, when comparing your two frequency files, that the list of decreasing frequencies nearly match, strongly suggesting which ciphertext characters (shown in the `project-4c.frequencies` file) should be replaced by which cleartext characters (shown in `4c-sample.frequencies`).

Creating a ciphertext-to-cleartext map: I have provided a program that will use a pair of frequency files to create a *substitution map*—a file that determines which charcater should be replaced with which other character in performing an encryption/decryption substitution. That is, this file allows you to explicitly dictate the values in the map that you normally, when using a substitution cipher, generate with a random permutation. This file reads the *character number* from each line of two frequency files, and then pairs them up. So, if you were to run this program, named `MakeMap`, on your two frequency files, like so ...

```
$ java MakeMap project-4c.frequencies 4c-sample.frequencies 4c.map
```

... then you can open the newly created `4c.map` file with *Emacs* and see ...

```
77 65
137 32
16 105
...
```

Specifically, each line in the file contains two numbers that specify how one character should be substituted for another:

1. **Original character number:** The underlying number of some character that may occur in a file. For example, the first line shown above specifies character number 77 (which represents the character `M`) as the original character.

2. **Substitute character number:** The underlying number of a character that should be used to replace the original character wherever it occurs in some file. Again using the first line of the above, the second number listed is 65 (also known as the character `A`). Thus, taken together, this line specifies that each occurrence of an `M` in some original file should be replaced with an `A`.

Applying the map to the ciphertext: Now that you've measured the frequency of the ciphertext message and the sample cleartext message, and then constructed a map of ciphertext-to-cleartext characters based on those frequencies, it is time to use that map to decrypt the ciphertext. I have provided another small program, `ForcedSub`, that performs a character-by-character substitution based on a given map. You can run the program like so:

```
$ java ForcedSub project-4c.ciphertext project-4c.cleartext 4c.map
```

This program will use the given substitution map (from `4c.map`) and then replace each original/ciphertext character in `project-4c.ciphertext` with its corresponding substitute/cleartext character to create `project-4c.cleartext`. Thus, if your frequency measurements were performed correctly in your `CountFrequency` program, then you should be able to read `project-4c.cleartext`. Note that the instructions ask you to create a new file: name it `message-source.cleartext` and, when you encrypt it, call the encrypted version `message-source.ciphertext`.

Notice, however, that the recovery of the cleartext message is likely to be imperfect: the frequencies of the characters don't quite match, and the substitution ordering may be slightly off. The message is likely to be readable, and that's good enough to follow the directions in the decrypted file. For your own edification, you may want to open `4c.map` and change a few entries to improve the decryption performed by `ForcedSub`.[5] However, this task is optional for the curious who wish to delve more deeply into how this form of cryptanalysis works.

[5] If you attempt this task, you may wish to consult a table of ASCII character encodings so that you can determine which entries to change.

3 How to submit your work

Once again, use the `cs111-submit` command:

```
cs111-submit project-4c CountFrequency.java
                        message-source.ciphertext
```

Part C is due on Thursday/Friday, October 25/26, at the start of lab