Project 1

Historical cryptography

The goal of this project is to break a substitution cipher. A substitution cipher is a cipher in which a plaintext is converted character by character into ciphertext by using a predetermined rule of association. For example, the Caesar cipher is a special kind of substitution cipher in which

$A \to D$, $B \to E$, $C \to F$, etc.

The Caesar cipher has a very simple form, because each character is simply advanced three places to form the ciphertext. However, in general, the rule can be more complicated.

The following text is given in the file called `ciphertext.txt` in the assignment folder.

```
<X%Z|*aZXcWZa%c[R>Z[%csX%WZ<X%Z+%[PWPcaZcaWZXP|Z|sL[sXPa\Z[c>|Zi%RRZi*RRZLaZ<X%Z[Ls.|Z4XPsX
```

This ciphertext was produced in the following way:

1. A plaintext was chosen consisting only of uppercase alphabetical characters and the space character.
2. A mapping was chosen at random from the ASCII characters {‘’,‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’, ‘G’, ‘H’, ‘I’, ‘J’, ‘K’, ‘L’, ‘M’, ‘N’, ‘O’, ‘P’, ‘Q’, ‘R’, ‘S’, ‘T’, ‘U’, ‘V’, ‘W’, ‘X’, ‘Y’, ‘Z’} into all printable ASCII characters.
3. Each plaintext character was replaced by its image character under the mapping.

In case this is confusing, the following simplified example may be helpful. Suppose that the plaintext is “WHAT ARE YOU SAYING”. We pick a rule of association that maps the plaintext characters to ciphertext characters in the following way:

$A \to X$

$B \to E$

$C \to v$

$D \to \&$

$E \to r$

$F \to z$

$G \to @$

$H \to 6$

$I \to \backslash$

$J \rightarrow 8$

$K \rightarrow 5$

$L \rightarrow e$

$M \rightarrow$:

$N \rightarrow I$

$O \rightarrow D$

$P \rightarrow$)

$Q \rightarrow Y$

$R \rightarrow L$

$S \rightarrow Q$

$T \rightarrow +$

$U \rightarrow$ “

$V \rightarrow \_$

$W \rightarrow n$

$X \rightarrow b$

$Y \rightarrow \#$

$Z \rightarrow <$

space $\rightarrow$ ^

Then, under this rule,

ENC(“WHAT ARE YOU SAYING”) = “`n6X+^XLr^#D"^QX#\I@`”

where ENC denotes encryption. You may want to confirm that the first few letters are consistent with the mapping shown above. For instance, it is indeed true that W maps to n, H maps to 6, etc.

The embedding used in the simple example is not the same embedding that was used to produce the long ciphertext given at the beginning of this project description. Your job is to recover that embedding as well as the unknown plaintext.

Requirements

Your code should meet the following requirements.

1. You can use any of the acceptable programming languages we discussed in class.
2. Your code must read in input from a file. In other words, you must read in the ciphertext from the file ciphertext.txt. Do not hard code it as a string literal. This is essential because I will test your code on other ciphertexts.
3. Whatever language you choose, the basename of your solution should be "sol". For example, if you use C call your solution `sol.c`, if you use Java then call your solution `sol.java`.
4. You are free to work in your own IDE, but I would prefer it if you could ensure that all paths are relative, so that the same code works unchanged on cocalc.

If you would like information about how to compile your code from the command line on cocalc, I would be happy to assist you.

Input/Output Conditions

Corpus cleaner

Please create a program called `sol_cleaner.ext`, where `ext` denotes the file extension corresponding to your choice of programming language (.py, .c, .cpp, .c++, or .java).

This program should open and read in a file named `corpus.txt`. This file should be a large body of work written in English (such as one of the books freely downloadable from Project Gutenberg).

As output, the program should produce a new file called `corpus_clean.txt` which is similar to the input, except that:

- All characters that are not spaces or alphabetical characters have been filtered out.
- All alphabetical characters have been capitalized.

Statistics analyzer (plaintext):

Please create a program called `sol_sap.ext`, where `ext` denotes the file extension corresponding to your choice of programming language (.py, .c, .cpp, .c++, or .java).

This program should open and read in a file named `corpus_clean.txt`, which is the output from `sol_cleaner.ext`. As output it should produce a file called `corpus_freq.txt`, which contains the following. Each row is a pair

`<letter>, <rel_freq>`

where `letter` is a character occurring in the text and `rel_freq` is the relative frequency of the letter in the corpus. The relative frequency of a letter `c` is defined by:

$$relative\ frequency\ of\ c = \frac{\#\ occurrences\ of\ c\ in\ corpus}{\#\ letters\ in\ corpus}$$

Note that this will be a floating point number.

The letter/frequency pairs should be given in order of descending frequency. For example, the file should roughly look like this.

```
e, 0.082198
a, 0.050031
(etc)
z, 0.003000
```

I have made up the values in the above table for the sake of example.

It is possible to implement this algorithm in a way that only makes one pass over the corpus. However, if you prefer to read through the file 27 times, that is feasible.

Statistics analyzer (ciphertext)

Please create a program called `sol_sac.ext`, where `ext` denotes the file extension corresponding to your choice of programming language (.py, .c, .cpp, .c++, or .java).

This program should read in the file `ciphertext.txt`. The output should be a file `cipher_freq.txt` which is much like `corpus_freq.txt`, but with ciphertext characters rather than only letters and spaces. This is very similar to the statistical analyzer for the corpus, and you should be able to reuse the code you have from the previous task with very few changes. In fact there is a way to arrange things so that this program is identical to `sol_sap.ext`. (But please hand in two files to make the grading uniform.)

Cipher cracker

Please create a program called `sol_cracker.ext`, where `ext` denotes the file extension corresponding to your choice of programming language (.py, .c, .cpp, .c++, or .java).

This program reads in three files: `ciphertext.txt`, `cipher_freq.txt`, and `corpus_freq.txt`.

The output of this program should be a file called `cracked.txt`. It should be the result of replacing each charact of ciphertext.txt with a capital Roman letter or a space. The correspondnce is given by the files `cipher_freq.txt` and `corpus_freq.txt`. The character in row i of `cipher_freq.txt` should be replaced with the letter (or space) in row i of `corpus_freq.txt`. (Note that for

this to work, both `cipher_freq.txt` and `corpus_freq.txt` must be sorted by frequency.)

The result will probably not be a perfect decrypt, but the output should be close enough to cleartext to repair the remaining problems manually.

Manual corrections (optional)

In this optional part of the project, you should create a file called `sol_corrections.ext`, where `ext` denotes the file extension corresponding to your choice of programming language (.py, .c, .cpp, .c++, or .java).

This program should read the file `cracked.txt` as input and output a file called `clear.txt`. The output should be the original plaintext used to produce the ciphertext.

This program works by making letter substitutions in `cracked.txt` that you observe to be appropriate, treating the remaining work to be done as a recreational cryptogram.

Reports

In this directory you will see a script called `test.py`. I will use this script, or something like it, to help grade your code. You can execute it from the command line like this:

```
$ python test.py
```

Do not type the `$`, that is only in the above to show where the command prompt is. Please let me know if you get strange results.

The script will produce a file called ‘Pr