CMPUT 299, Winter 2018, Assignment 7

All assignment submissions must conform to the Assignment Submission Specifications *posted on eClass. Ensure that your submission follows these specifications before submitting your work.*

You will produce a total of three files for this assignment: “preproc.py” for problem 1, “vigenereIC.py” for problems 2, 3, and 4, and a README file (also in either PDF or plain text). **If your code requires any external modules or other files to run, include them in your submission as well. Your submission should be self-contained.**

Problem 1

One way to make a ciphertext harder to hack is to apply some *pre-processing*, intentionally modifying the original plaintext in order to thwart one or more known methods of hacking.

For example, we saw in chapter 18 that we could hack the simple substitution cipher by matching ciphertext words to dictionary words with the same pattern. If we know about this weakness, we could make our ciphertexts stronger by deliberately misspelling words. For example, *“Thiss sentince iz ann eksample uv ths mehthod.”* For someone who knows the key, these misspellings are merely annoying; with a little effort, the intended message can still be read. However, the dictionary hacking method will fail, as the letter pattern has changed completely. Through pre-processing, the message was made harder to hack.

In this exercise, you will use pre-processing to make short messages encrypted with the Vigenere cipher, which you saw in chapter 19, more resilient to Kasiski examination, which you saw in chapter 21.

Recall that Kasiski Examination (also known as the Babbage Attack) begins by determining the key length; if the key length cannot be correctly determined, the method is much less effective. Recall further that Kasiski Examination determines the key length by finding repeated sequences of letters in the ciphertext. **If the ciphertext does not contain any repeated sequences of length at least 3, Kasiski Examination is not effective.** It is this flaw in Kasiski Examination that you will exploit.

For example, consider this example from chapter 21:

```
 PPQCAXQVEKGYBNKMAZUYBNGBALJONITSZMJYIMVRAG
VOHTVRAUCTKSGDDWUOXITLAZUVAVVRAZCVKBQPIWPOU
```

The repeated sequences – underlined for convenience – are what allows the Kasiski Examination to guess the key length. This ciphertext enciphers the following plaintext, enciphered using the key “WICK” (plaintext sequences corresponding to the repeated ciphertext sequences are again underlined):

```
 THOSEPOLICEOFFICERSOFFEREDHERARIDEHOMETHEY
TELLTHEMAJOKETHOSEBARBERSLENTHERALOTOFMONEY
```

Suppose we instead pre-process the plaintext as follows:

```
THOSEPOLICEOFFICERSXOFFEREDHERARIDEHOMETHEY
TELLTHEMAJOKETHOSEBARBERSLENTHERALOTOFMONEY
```

All that changed is the following: before the start of the first repeated ciphertext sequence, in this case ‘YBN’, another character (in this case, an ‘X’) was added into the corresponding point in the plaintext. This forces the key to increment once more before the second instance of ‘OFF’, so it will not be enciphered as ‘YBN’ this time. **This ensures that, up to this point in the ciphertext, there are no repeated sequences for a Kasiski Examination to use.**

Repeating this process a few times to remove all remaining repeated sequences, we get this plaintext:

```
 THOSEPOLICEOFFICERSXOFFEREDHERARIDEHOMETHEY
TELLXTHEMAJOKETHOSEBARBERSLENXTHERALOTOFMONEY
```

and this ciphertext:

```
 PPQCAXQVEKGYBNKMAZUHKNHONMFRAZCBELGRKUGDDMA
DATNHPPGWWRQUABJYOMDKNJGBOTGXTBJONINYPWHWKVGI
```

Try running the “hackVigenere” function on this new ciphertext; you will see that it fails, since there are no repeated sequences for the Kasiski examination to use! All we did was insert characters at a few key points in the ciphertext – leaving it not much harder to read – and our ciphertext is now immune to the hacking program introduced in chapter 21.

Your task is to automate this process. Create a module called “preproc.py”. In this module, create a function “antiKasiski”, which does the following:

- Takes two parameters: a key, and a plaintext, in that order.
- Removes all non-letter characters from the plaintext.
- Obtains the ciphertext from the Vigenere cipher with the given key.
- Repeats the following in a loop until there are no repeated sequences of length at least 3 in the ciphertext:

  Find the beginning of the the first repeated ciphertext sequence of length at least 3, which could be useful to Kasiski examination.

  (Hint: the ‘findRepeatSequencesSpacings’ function can be modified to help you here.)

  Insert a random letter into the corresponding position in the plaintext.
- Returns a string containing the final ciphertext.

Like the “misspelling” method at the start of this exercise, this method is not perfect, but does offer increased security, especially on shorter messages. On longer ciphers, the chances of repeated sequences arising by chance is much higher (and eventually becomes inevitable), so this method will be less effective. Your program will be considered correct if it works on ciphers as long as the one in the example.

Problem 2

Even if no repeated sequences are present, a mathematical concept known as the *index of coincidence* (IC) can be used to deduce the key length, provided the cipher is reasonably long. The rest of this assignment will guide you toward using this technique to identify the length of the key used to produce a Vigenere ciphertext, without depending on repeated sequences.

Let c_i be the number of times letter i occurs in the text. For example, in the ciphertext ABA, $c_A = 2$, $c_B = 1$, and all other c_i values are 0. Let N be the number of letters in the ciphertext. Then the IC is given by the following formula:

$$\frac{\sum_{i=A}^{i=Z} c_i(c_i - 1)}{N(N-1)}$$

Create a module "vigenereIC.py". Add to this module a function "stringIC", which takes a string as input, and returns the IC of that string. For example, "vigenereIC.stringIC('ABA')" should return 0.3333... since

$$\frac{c_A(c_A-1) + c_B(c_B-1)}{N(N-1)} = \frac{2(1)+1(0)}{3(2)} = \frac{2}{6} = \frac{1}{3} = 0.3333\ldots$$

Problem 3

Finally, recall that, given a key of length n, the Vigenere cipher will encrypt every n^{th} character with the same monoalphabetic substitution cipher. For example, given a key of length 3 (so $n = 3$), characters 0, 3, 6, 9,... will be enciphered with one monoalphabetic substitution cipher, characters 1, 4, 7, 10,... will be enciphered with another, and characters 2, 5, 8, 11,... will be enciphered with another. Thus, a Vigenere ciphertext consists of n interleaved subsequences, each consisting of every n^{th} character, each starting from some index between 0 and $n-1$. The function "getNthSubkeysLetters" in "vigenereHacker" can be used to retrieve these subsequences. For example:

```
>>> import vigenereHacker
>>> vigenereHacker.getNthSubkeysLetters(1, 4, 'ABCDABCDABCDABCDABCD')
'AAAAA'
>>> vigenereHacker.getNthSubkeysLetters(2, 4, 'ABCDABCDABCDABCDABCD')
'BBBBB'
>>> vigenereHacker.getNthSubkeysLetters(3, 4, 'ABCDABCDABCDABCDABCD')
'CCCCC'
>>> vigenereHacker.getNthSubkeysLetters(4, 4, 'ABCDABCDABCDABCDABCD')
'DDDDD'
```

Add a function to your "vigenereIC.py" module called "subseqIC", which takes as input a string containing a Vigenere ciphertext, and a key length, in that order, and returns the average IC of the subsequences of the ciphertext induced by that key length. For example:

```
>>> import vigenereIC
>>> vigenereIC.subseqIC('PPQCAXQVEKGYBNKMAZUHKNHONMFRAZCBELGRKUGDDMA', 3)
0.03736263736263736
>>> vigenereIC.subseqIC('PPQCAXQVEKGYBNKMAZUHKNHONMFRAZCBELGRKUGDDMA', 4)
0.05909090909090909
>>> vigenereIC.subseqIC('PPQCAXQVEKGYBNKMAZUHKNHONMFRAZCBELGRKUGDDMA', 5)
0.016666666666666663
```

Problem 4

One important property of the index of coincidence is that it is substitution invariant – that is, applying a monoalphabetic substitution to a text will not change its IC.

Another important property of the IC is that a text written in English will tend to have a higher IC than a text with a more uniform frequency distribution, such as a Vigenere ciphertext. The expected average value for the IC can be computed from the relative letter frequencies of the source language as $\sum f_i^2$, where $f_i = c_i/N$. The expected IC value for English is about 0.066. The IC value for a uniform distribution (random text) of the same alphabet is $\sum (1/26)^2 = 26(1/26)^2 \approx 0.039$.

Putting this together, we have the following heuristic for deducing the length of the key used to create a Vigenere ciphertext: *The higher the average IC of the subsequences of the ciphertext induced by a key length, the more likely that key length equals the length of the key used to create the ciphertext.*

We can see this heuristic at work in the example at the end of Problem 3: guessing a key length of 4 gives a higher average sub-sequence IC than key length 3 or key length 5, and indeed, that ciphertext comes from problem 1, where the keyword used was "WICK", which of course has length 4.

Add a function to your "vigenereIC.py" module called "keylengthIC", which takes as input a ciphertext, and which returns the top five most likely key lengths, in order from most to least likely, according to the above heuristic (that is, it returns the five key lengths which give the highest average IC, over the subsequences they induce). Your function should try all key lengths between 1 and 20, inclusive (you need not handle longer key lengths).

Your vigenereIC.py module should now contain at least the following three functions: stringIC (Problem 2), subseqIC (Problem 3), and keylengthIC (Problem 4).