CMPUT 331, Fall 2020, Assignment 7 (Version 1.1)

Before beginning work on this assignment, carefully read the Assignment Submission Specifications posted on eClass.

You will produce a total of three files for this assignment: "a7p1.py" for problem 1, "a7p234.py" for problems 2, 3, and 4, and a README file (in either PDF or plain text format). **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.** Input text in this assignment may be assumed to be uppercase and contain only letters.

Problem 1

One way to make a ciphertext harder to crack is to apply some *pre-processing*, intentionally modifying the original plaintext in order to thwart one or more known methods of cryptanalysis. In this exercise, you will use pre-processing to make short messages encrypted with the Vigenere cipher more resilient to Kasiski examination.

Recall 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 the following plaintext and ciphertext respectively:

```
 THOSEPOLICEOFFICERSOFFEREDHERARIDEHOMETHEY
TELLTHEMAJOKETHOSEBARBERSLENTHERALOTOFMONEY

 PPQCAXQVEKGYBNKMAZUYBNGBALJONITSZMJYIMVRAG
VOHTVRAUCTKSGDDWUOXITLAZUVAVVRAZCVKBQPIWPOU
```

The repeated sequences – underlined for convenience – are what allow the Kasiski Examination to guess the key length. If we preprocess by inserting another character before the repeated ciphertext sequences, the key will be incremented before the second instance of the repeated sequence. This results in the sequence not being enciphered by the same characters.

The processed plaintext and its corresponding ciphertext respectively would be:

```
 THOSEPOLICEOFFICERS-OFFEREDHERARIDEHOMETHEY
TELL-THEMAJOKETHOSEBARBERSLEN-THERALOTOFMONEY

 PPQCAXQVEKGYBNKMAZUHKNHONMFRAZCBELGRKUGDDMA
DATNHPPGWWRQUABJYOMDKNJGBOTGXTBJONINYPWHWKVGI
```

Create a module called "**a7p1.py**" with a function "**antiKasiski(key, plaintext)**"

- key: vigenere cipher keyword, uppercase string containing only letters.
- plaintext: uppercase plaintext containing only letters

Your function remove all repeated sequences greater than 3 characters by inserting a "-" character after the first sequence in an instance of a repeated sequence. After each insertion, your program must recompute the subsequences with the characters "-" as part of the text, however, repeated sequences that already contain a "-" can be ignored. Your approach to finding subsequences should be the same as how the vigenereCipher encrypts text. Once your program has removed all repeated subsequences, return a string with each "-" character replaced with a random character. A solution is valid if it does not contain repeated subsequences of length greater or equal to 3 and you can assume that the given plaintext will only contain uppercase letters.

Problem 2

Even if no repeated sequences are present, 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 "**a7p234.py**". Add to this module a function "**stringIC(text)**", which takes a string as input, and returns the IC of that string. For example, "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. 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.py" can be used to retrieve these subsequences.

Add a function to your "**a7p234.py**" module called "**subseqIC(ciphertext, keylen)**", 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:

```
python3 -i a7p234.py
>>> vigenereIC.subseqIC('PPQCAXQVEKGYBNKMAZUHKNHONMFRAZCBELGRKUGDDMA', 3)
0.03882783882783883
>>> vigenereIC.subseqIC('PPQCAXQVEKGYBNKMAZUHKNHONMFRAZCBELGRKUGDDMA', 4)
0.0601010101010101
>>> vigenereIC.subseqIC('PPQCAXQVEKGYBNKMAZUHKNHONMFRAZCBELGRKUGDDMA', 5)
0.012698412698412698
```

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.*

Add a function to your "**a7p234.py**" module called "**keyLengthIC(ciphertext, n)**", which takes as input a ciphertext, and returns a list containing the top "**n**" 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). You can break ties by favouring higher key lengths. Your function should try all key lengths between 1 and 20, inclusive (you need not handle longer key lengths). You can assume **n** will between 1 and 20 inclusive.

Your **a7p234.py** module should now contain at least the following three functions: **stringIC** (Problem 2), **subseqIC** (Problem 3), and **keyLengthIC** (Problem 4).