CMPUT 396, Fall 2019, Assignment 9

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 6 files for this assignment: four Python modules, your a9.pdf or a9.txt, 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. Modules from the textbook must not be edited – if you wish to modify code from the textbook, put it in a module with a different name.**

In this assignment, you will revisit the simple substitution cipher. In Assignment 5, you saw cryptanalysis programs for this cipher based on character repetition patterns within words (the textbook solver), and character frequencies (frequency analysis). The former tried to recognize patterns in the ciphertext characters, while the latter leveraged statistical information about English. Here, you will implement an approach which combines character sequence information with language statistics. This approach is based on sequences of characters, called *character n-grams*.

A character n-gram is a string consisting of n characters (n is simply a variable with a positive integer value) which appear in sequence in some string. For the purposes of this assignment, the set of all characters will be the 26 uppercase letters of the English alphabet and the "space" character, giving a total of 27 characters. You need not handle punctuation, digits, or any other symbols other than uppercase letters and the space character.

For example the string "AN EXAMPLE" contains ten 1-grams (or "unigrams"), nine 2-grams ("bigrams"), eight 3-grams ("trigrams"), and so on, as shown in the table below (the '␣' symbol is used to make space characters easier to see):

n	n-grams in "AN EXAMPLE'
1	"A", "N", "␣", "E", . . . , "E"
2	"AN", "N␣", "␣E", "EX", . . . , "LE"
3	"AN␣", "'N E', "␣EX", "EXA", . . . , "PLE"
4	"AN E", "N EX", "␣EXA", "EXAM", . . . , "MPLE"
5	"AN EX", "N EXA", "␣EXAM", "EXAMP", . . . , "AMPLE"

Just like individual characters, character n-grams have different relative frequencies in English. For example "TH" and "ING" are relatively common, while "TN" and "YYY" are relatively rare. In this assignment, you will implement a substitution cipher solver based on this linguistic principle.

NOTE: If there are multiple spaces sequentially, we treat it as a single space.

Problem 1

Create a module called a9p1.py. In this module, create a function called `ngramsFreqs( text, n)`, which takes as input a string of text, and the n-gram variable n. It should return a dictionary in which each key is a character n-gram, represented as a single string, and the value of a key is the relative frequency of that n-gram in that string of text (the number of times that n-gram occurs, divided by the total number of character n-grams in the given string of text).

Problem 2

For the purpose of this assignment, a *mapping* will be a Python dictionary which maps our set of 27 characters to itself bijectively (so the mapping is exactly one-to-one, with 27 unique keys and 27 unique values), with the constraint that 'space' is always mapped to itself. Given a mapping, a ciphertext, and an n-gram frequency dictionary, the *n-gram score* of the key is computed as follows:

1. Attempt to decipher the ciphertext using the mapping, by mapping each ciphertext character to the value it has in that key.
2. For a given n-gram g, let $c(g)$ be the number of times it occurs in the decipherment, and let $f(g)$ be its relative frequency in the given dictionary. (Note that $c(g)$ is an integer, while $f(g)$ is a floating point number between 0 and 1.)
3. Letting G be the set of all n-grams which occur in the decipherment, the score of the provided mapping is: $\sum_{g \in G} c(g) \cdot f(g)$

Create a module called a9p2.py. In this module, create a function called `keyScore( mapping, ciphertext, frequencies, n )`, which returns the n-gram score (a floating-point number), computed given that mapping, ciphertext (given as a string), an n-gram frequency dictionary (such as is returned by your ngramsFreqs function), and the n-gram parameter n.

Problem 3

Given a mapping m, we can create a new mapping, call it m', by choosing two keys in the dictionary that is m, and exchanging their values. For example, if m maps 'A' to 'X' and 'B' to 'Y', one such "swap" would have m' be identical to m, except that m' maps 'A' to 'Y' and 'B' to 'X'. Since a mapping has 26 keys which are not fixed (recall that any mapping must map space to itself), the total number of swaps is equal to 325 (there are 25 characters to swap 'A' with, 24 for 'B', and so on).

For a mapping m, let S_m be the set of all mappings which can be created by swapping two values in m in this way. We will call these mappings *the successors of m*. Each successor differs from the original mapping by only two letters. Of course, some successor of m might have a higher score than m itself – that is, in terms of n-gram frequencies, it might be a better mapping.

Create a module called a9p3.py. In this module, create a function called `bestSuccessor( mapping, ciphertext, frequencies, n )`, which computes the n-gram score of each possible successor of the given mapping, given the ciphertext (as a string), an n-gram frequency dictionary, and the value of n. If any such successor has a higher score than the given mapping, the function should return the successor with the highest such score (you make break ties however you wish). Otherwise, it should return the original mapping.

Problem 4

Now we are ready to crack some ciphers! Create a module called a9p4.py. In this module, create a function called `breakSub( ciphertext, freqtext, maxN )`. Initially, this function should create a mapping based on frequency analysis using the text from `freqtext` as a source (so, the i^{th} most frequent cipher character is mapped to the i^{th} most frequent English character). It should then do the following:

- start with $n =$ `maxN`.
- repeatedly apply your `bestSuccessor` function, with the given ciphertext, an n-gram frequency dictionary derived from the given `freqtext` source string, and n-gram size n.
- stop when that function returns the same mapping it was given (that is, when a better mapping cannot be found through any swap).
- set $n = n - 1$, and repeat the above two steps.

Once the final iteration for $n = 1$ is complete, the function should return a string containing the decipherment produced using that mapping.

This strategy of repeatedly replacing the current solution with the best of a set of successor solutions until no further improvement is possible is a strategy called *hill climbing*.

Problem 5 Short answer:

Included in `ciphertext.txt` is a single string message encrypted using a random unknown mapping, and `plaintext.txt` is the corresponding plaintext. Apply the `breakSub` solver from Problem 4, starting with `maxN` $= 1$, and gradually increase until the returned deciphered text does not improve (as measured by decipherment accuracy). Use your decipherment accuracy code from Assignment 5. Answer the following in your a9.txt or a9.pdf:

1. At what level of `maxN` do you notice the attempted deciphered text does not improve? Indicated the value of `maxN`, as well as the decipherment accuracy.
2. Why does changing the value of `maxN` give different decipherment accuracies?
3. Give one benefit of using a high level of `maxN`, and give one bad consequence of using a high level of `maxN`.