CMPUT 331, Fall 2023, Assignment 6
Version 1, Oct 16, 2023

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

Introduction

You will produce six files for this assignment:

- python solutions to problems 1, 2, 3 and 4 (**a6p1.py**, **a6p2.py**, **a6p3.py**, **a6p4.py**)
- the decipherment produced in problem 4 (**text_plain.txt**)
- a README file (**README.txt** or **README.pdf**)

Submit your files in a zip file named **331-as-6.zip**.

In this assignment, you will revisit the substitution cipher. In Assignment 5, you saw cryptanalysis programs 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 to combine 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. English n-gram frequencies are to be computed based on the included text file "wells.txt".

Problem 1

Create a module called a6p1.py to complete the function called `ngramsFreqsFromFile( textFile, n )`, which takes as input a string path to a text file, and the n-gram integer 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 as a float of that n-gram in that text file (the number of times that n-gram occurs, divided by the total number of character n-grams in the given file).

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 string keys and 27 unique string 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 into a decipherment 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 frequency dictionary.
   (Note that: (1) $c(g)$ is an integer; (2) $f(g)$ is a float between 0 and 1; (3) $f(g) = 0$ if g is not in the frequency dictionary.)
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 a6p2.py to complete the function `keyScore( mapping, ciphertext, frequencies, n )`, which returns the n-gram score, as a floating-point number, computed given that mapping, ciphertext (given as a string), an n-gram frequency dictionary (such as is returned by your `ngramsFreqsFromFile` function), and the n-gram integer 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. 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 a6p3.py to complete the function called `bestSuccessor( mapping, ciphertext, frequencies, n )`, which computes the n-gram score of each possible successor of the input mapping, given a string ciphertext, 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. Otherwise, it should return the original mapping. To maintain consistency when dealing with ties, use the provided `breakKeyScoreTie` function to obtain the better successor when there is a tie between key scores.

Problem 4

Now we are ready to crack some ciphers! Create a module called a6p4.py to complete the function called `breakSub( cipherFile, textFile, n )`. Initially, this function should read the ciphertext from the input cipherFile, that is the path to the ciphertext file. After that, the function should create a mapping based on frequency analysis (so, the i^{th} most frequent cipher character is mapped to the i^{th} most frequent English character). It should then repeatedly apply your `bestSuccessor` function, with the ciphertext being read in, an n-gram frequency dictionary derived from the input textFile (the path to the given text file), and n-gram size n, halting when that function simply returns the same mapping it was given (that is, when a better mapping cannot be found through any swap). Once this happens, the function should generate the decipherment using that mapping and finally save the decipherment as a new text file.

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*.

For submission, the `breakSub` function should take "text_cipher.txt" as the input cipherFile, "wells.txt" as the input textFile, and 3 as the input n, and save the decipherment as an output text file named "text_plain.txt" for the submission.

How to Test

For Problem 1, Problem 2, and Problem 3, we have included a comprehensive set of test cases in the **test.py** file. We encourage you to utilize these test cases extensively to thoroughly validate your implementations before submitting your assignment. These test cases are designed to help you ensure the correctness and robustness of your solutions.

```
$ python test.py
Problem 1: Tests passed :)
Problem 2: Tests passed :)
Problem 3: Tests passed :)
```

For Problem 4, we have provided the test cipher file, "text_finnegan_cipher.txt". To validate your implementation, use this test cipher file as the input (cipherFile) and compare your resulting deciphered text in "text_plain.txt" with the provided reference file "text_finnegan_plain.txt". Following your testing, don not forget to execute your `breakSub` function on the main cipher file "text_cipher.txt" to generate the corresponding "text_plain.txt" for your final submission.