15-110: Principles of Computing

Homework 08

Due: 11th October, 2022 at 10:00pm

- You need to complete the Python file for this assignment, and submit it to Gradescope.
- There are 100 points.
- You must solve the tasks **individually**, always abiding by the course and CMU's academic integrity policy.
- We are not giving you any starter code this week. That means you need to create your file from scratch and define your own test cases. For writing test cases, you may follow the style of test cases used in the previous homework.

1. (20 points) **Caesar Cipher**

   Julius Caesar used a system of cryptography, now known as Caesar Cipher, which shifted each letter 2 places further through the alphabet (e.g. "A" shifts to "C", "R" shifts to "T", etc.). At the end of the alphabet we wrap around, that is "Y" shifts to "A". We can, of course, try shifting by any number.

   Implement the function `decode(T, n)` that takes an input a string `T`, representing the encode text, and the shift `n` that was used to encode it. The function returns the original text. You can assume that there are only letters in the text (no spaces or punctuation) and that all letters are uppercase.

   You should know this, but to avoid any confusion, this is the alphabet:

   ```
   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
   ```

   For example:

   - `decode("LIPPSASVPH", 4)` should return `"HELLOWORLD"`
   - `decode("ECV", 2)` should return `"CAT"`

2. (20 points) **Tautogram**

   Fatima has always loved poetry, and recently she discovered a fascinating poetical form. Tautograms are a particular case of alliteration, which is the occurrence of the same letter at the beginning of adjacent words. In particular, a sentence is a tautogram if all of its words start with the same letter.

   For instance, the following sentences are tautograms:

   - Flowers Flourish from France
   - Sam Simmonds speaks softly
   - Peter pIckEd pePPers
   - truly tautograms triumph

Fatima wants to impress her friends by writing a letter full of this kind of sentences. Please help Fatima check if each sentence she wrote down is a tautogram or not.

Implement the function `tautogram(S)` that takes a sentence as a string (a sequence of words separated by spaces), and returns `True` if `S` is a tautogram and `False` otherwise. You may assume that the words are formed only by letters, and there are no punctuation marks.

For example, `tautogram("Flowers Flourish from France")` should return `True`.

3. (25 points) **Relevant Word Frequency**

   You are tasked to write a program for finding out the category a text belongs to. In order to find this out, you decided to see which words occur more frequently in the text, and you wrote a python program that computes the *frequency* of each word, i.e., the number of times the word occurs divided by the total number of relevant words.

   However, on a first attempt, you realized that the most frequent words are usually irrelevant for guessing a category, such as "the", "is", and "are". You also noticed that your program would count "Python" and "Python's" as two different words because of the contraction.

   There are a lot of improvements you can make to your code, so you have decided to implement it again, this time removing all the parts of the text that are not relevant.

   Implement the function `relevantWordFrequency(text)` that takes a string `text` as a parameter, and returns a list of tuples of two elements, where the first element of each tuple is a word in `text` and the second element is the frequency of the word.

   The function should satisfy the following:

   1. You should only count words that have more than 3 characters (so "are" and "is" should not be counted, for example).
   2. All punctuation should be ignored. With this aim, you can use the string constant `string.punctuation` from the module `string` (`https://docs.python.org/3/library/string.html`).
   3. All contractions should be ignored. They are:
      - 'm
      - 're
      - 's
      - 've
      - 'll
      - 'd
      - n't
   4. Capitalization should be ignored, and the words in the returned list should appear lower case.

   For example, if `text` is the string:

```
'''
I am hungry.

Me too, I'm really hungry! Let's get some   food!
'''
```

   then `relevantWordFrequency(text)` should return the list:

```
[
  ('food', 0.2)
  ('hungry', 0.4),
  ('really', 0.2),
  ('some', 0.2)
]
```

The returned list must be *sorted* in ascending order. In the example, `('food', 0.2)` is the first element of the list since, alphabetically, the letter 'f' comes before the other first letters of the words.

To better organize the code, you must write a (helper) function `cleanText(text)` and use it inside `relevantWordFrequency(text)`. The function `cleanText(text)` takes as input a string `text` and removes from the input string all punctuation, contractions, and spaces. Moreover, all letters are lowercased. The function returns a 'cleaned' list of words from the input string in the same order as they occurred in the string.

For instance, `cleanText(text)` where `text` is the above example string, returns `['i', 'am', 'hungry', 'me', 'too', 'i', 'really', 'hungry', 'let', 'get', 'some', 'food']`.

Most likely, the first step in `relevantWordFrequency(text)` is precisely to invoke `cleanText(text)` ;-)

4. (35 points) **DNA Match**

   In the following problems, a DNA sequence is represented by a string composed of the characters "A", "C", "T", and "G" only.

   (a) An important problem in working with DNA is *subsequence matching*. In short, subsequence matching determines if a short DNA sequence occurs within a longer sequence. (All the letters on the subsequence occur consecutively within the original sequence.) Implement the function `simpleSubSeqMatch(sequence, subSeq)` which, given a DNA sequence `sequence` and a shorter DNA sequence `subSeq`, returns how many times `subSeq` occurs within `sequence`.

   For example:

   - `simpleSubSeqMatch("ACCCCTTT", "CCT")` should return `1`.
   - `simpleSubSeqMatch("ACCTACCT", "CCT")` should return `2`.
   - `simpleSubSeqMatch("ACCCTTTT", "TT")` should return `3`.

   (b) Sometimes, when doing subsequence matching, we allow partial matches instead of perfect matches. A partial match is one where we don't care about the value of certain characters. If we don't care about a character, we give it the value `"N"`. For example, `"ANC"`can match any three letter sequence that starts with `"A"` and ends with `"C"`, such as `"ATC"`, `"AGC"`, etc.

   Implement the function `subSeqMatch(sequence, subSeq)` which, given a DNA sequence `sequence` and a shorter DNA sequence `subSeq` which may contain `N` values, returns how many times `subSeq` occurs within `sequence`.

   For example:

   - `subSeqMatch("ACCTCT", "TNT")` should return `1`.
   - `subSeqMatch("ACCTAGCT", "NCT")` should return `2`.
   - `subSeqMatch("ACCCTTTT", "NTT")` should return `3`.

   You may find it helpful to create helper functions to solve this task.