COSC122 (2019su2) Assignment

Part 1 – Linear Search

Your assignment is to write the key algorithmic components of a simple guessing game. The goal is to give you some experience with several data structures and algorithms, and with the issues surrounding algorithm design and optimisation. The assignment consists of two parts; the second part explores a better way of solving the problem, but is independent of the first (that is, part two does not require you to have a completely functional part one).

Essential Information

The entire assignment is worth 20% of your final grade.

- *Part 1*: Due 10pm on Sunday 12 Jan 2020 (10% of course mark).
- *Part 2*: Due 10pm on Sunday 2 Feb 2020 (10% of course mark).

Submissions should be made through the quiz server site — submission quizzes will be made available closer to the due time (feel free to remind us if the submission quiz hasn't surfaced by the middle of January). These quizzes won't really test your code until after the deadline so it is up to you to test that your code works. The submission quizzes will initially just check that your code works for a trivial case, ie, check that you haven't made a trivial error.

Shannon's Game

The assignment is based on an experiment designed in the 1950s by the mathematician (and juggler) Claude Shannon [1] to examine the *information content* of English. The experiment was similar to the game of *Hangman*, where players attempt to reveal a hidden word by guessing which letters are in it. However, instead of guessing single words, subjects were asked to guess entire phrases they had never seen before, and had to guess each letter sequentially.

The number of attempts it took for subjects to correctly guess each letter is interesting because people don't guess randomly. Each additional letter that is revealed allows people to make better predictions about what will come next—even if they are unfamiliar with the word or phrase. For example, if you know that the first two letters of a word are "`TH`", then you will probably be able to make a pretty good guess for the most likely possibilities for the next letter.[2]

Although this may seem like just an interesting curiosity, the game has a very important application: predicting the next character of text or pixel in an image is the basis of compression systems, and the better the prediction is made, the better the compression that can be achieved. This idea underpins many compression methods that are now commonplace, including zip, rar, bzip, jpeg, mp3 and mpeg compression.

Playing the Game

A simple strategy for playing the game follows these steps:

1. Using a *corpus* of English text (a large body of text that is representative of the words and sentence structure of English), a program can "learn" about the probabilities of particular letter combinations.
2. When asked to guess the next letter of a phrase, it takes the last couple of letters that are already known in the guessed phrase, and looks through the corpus to find possibilities for the next letter.
3. These possibilities are ranked by how frequently they occur in the corpus.
4. It will "guess" each of these possibilities from most likely to least likely until it one of them matches. (If it runs out of guesses, it will have to fall back on guessing randomly.)

[1] C. E. Shannon, "Prediction and Entropy of Printed English", *Bell System Technical Journal*, vol. 30, pp. 50–64, 1951.

[2] Probably an `E`, `I`, `A`, `R`, or `O`.

Let's look at an example of this in action. Suppose we have the following corpus of text (in reality, this will probably be an entire book—or several!):

```
ONCE MORE UNTO THE BREACH, DEAR FRIENDS, ONCE MORE; OR CLOSE THE WALL UP WITH OUR ENGLISH
DEAD. IN PEACE THERE'S NOTHING SO BECOMES A MAN AS MODEST STILLNESS AND HUMILITY: BUT WHEN
THE BLAST OF WAR BLOWS IN OUR EARS, THEN IMITATE THE ACTION OF THE TIGER; STIFFEN THE SINEWS,
SUMMON UP THE BLOOD, DISGUISE FAIR NATURE WITH HARD-FAVOUR'D RAGE; THEN LEND THE EYE A
TERRIBLE ASPECT; LET PRY THROUGH THE PORTAGE OF THE HEAD LIKE THE BRASS CANNON; LET THE
BROW O'ERWHELM IT AS FEARFULLY AS DOTH A GALLED ROCK O'ERHANG AND JUTTY HIS CONFOUNDED
BASE, SWILL'D WITH THE WILD AND WASTEFUL OCEAN. NOW SET THE TEETH AND STRETCH THE NOSTRIL
WIDE, HOLD HARD THE BREATH AND BEND UP EVERY SPIRIT TO HIS FULL HEIGHT. ON, ON, YOU NOBLEST
ENGLISH. WHOSE BLOOD IS FET FROM FATHERS OF WAR-PROOF! FATHERS THAT, LIKE SO MANY ALEXANDERS,
HAVE IN THESE PARTS FROM MORN TILL EVEN FOUGHT AND SHEATHED THEIR SWORDS FOR LACK OF ARGUMENT:
DISHONOUR NOT YOUR MOTHERS; NOW ATTEST THAT THOSE WHOM YOU CALL'D FATHERS DID BEGET YOU.
BE COPY NOW TO MEN OF GROSSER BLOOD, AND TEACH THEM HOW TO WAR. AND YOU, GOOD YEOMAN, WHOSE
LIMBS WERE MADE IN ENGLAND, SHOW US HERE THE METTLE OF YOUR PASTURE; LET US SWEAR THAT
YOU ARE WORTH YOUR BREEDING; WHICH I DOUBT NOT; FOR THERE IS NONE OF YOU SO MEAN AND BASE,
THAT HATH NOT NOBLE LUSTRE IN YOUR EYES. I SEE YOU STAND LIKE GREYHOUNDS IN THE SLIPS,
STRAINING UPON THE START. THE GAME'S AFOOT: FOLLOW YOUR SPIRIT, AND UPON THIS CHARGE CRY
'GOD FOR HARRY, ENGLAND, AND SAINT GEORGE!'
```

And that we're trying to guess the next letter in this quote:

```
EVEN AT T_____________________
```

Instead of looking at everything we know, let's just look at last two characters—"␣T" (a space followed by an "T").[3] If we go through the corpus and look for any characters following a "␣T" (underlined in the text above) and keep a tally of how many times each occurs, we'll arrive at:

```
H: 34
I: 2
E: 3
O: 3
```

If each of those letters is guessed in the order of frequency, we'll find that the first guess—H—correctly matches. [4]

Part 1: Linear Searching

For part 1, you will implement a simple algorithm that performs a linear search on the corpus of text looking for guesses for each letter of the phrase using the previous two—as described in the example above.

You have been supplied with the file `shannon_p1.py`, which implements some basic machinery. You only have to complete the marked function stubs (those that currently just say `pass`). *It is crucial that you implement these functions* ***exactly*** *as described.* You may add new functions for your convenience, but the existing functions must continue to behave as specified.

The Missing Components

The three module-level functions in `shannon_p1.py` roughly correspond to the steps of the algorithm described above:

- `filter_possible_chars`: returns a list of characters that immediately follow the given two characters in the `last` parameter in a `corpus` of text (step 2).
- `count_frequencies`: counts the frequencies of each element in a list returned by `filter_possible_chars` (step 3).
- `select_next_guess`: finds the letter with the highest guess in a list returned by `count_frequencies` and removes it from the list (step 4). If more than one letter has the highest frequency then the first letter in the list with that frequency is returned.

`count_frequencies` and `select_next_guess` should do their work using the supplied `FrequencyList` and `Frequency` classes that will implement a linked list. You do not have to implement `Frequency`, but your `FrequencyList` implementation will create and manipulate instances of it.

[3] You'll try different approaches in a later part of the assignment.

[4] It worked well this particular time; however, the corpus isn't large enough to solve the *entire* quote without making some random guesses.

Some Extensions

Here are some ideas that you could play around with if you find this assignment is very easy to do:

- The common implementation of the len method takes $O(n)$ to execute, write a version that takes $O(1)$ to execute. What is the memory trade off in this case?
- Using a list comprehension you can write the `filter_possible_chars` in a single not so unreadable line.
  - I find list comprehensions hugely valuable and I would recommend learning how to use the language feature, though don't start with the task above.
- You can implement your `FrequencyList` using doubly linked nodes, this somewhat complicates adding and removing elements but I found it fun to think about and test.

If you are the first to implement any one of the ideas above (email the code to me) I will award you bragging rights and a crunchy bar to be collected whenever suits you.

Testing Your Code

You won't be able to play the game until you have completed all of the missing functions, but you can test each piece of your code as you go along using a combination of *Wing's* shell and Python doctests:

```
Evaluating shannon_p1.py
>>> doctest.testmod()
TestResults(failed=0, attempted=33)
>>> filter_possible_chars('scurrying furred small friars', 'rr')
['y', 'e']
```

You should test your code with as lots of different inputs that exercise your code thoroughly (and add them as doctests).

If you want to test large corpora of text, you can use the supplied `load_corpus_and_play` function to load a given corpus file and play the game with the given phrase and phrase length:

```
load_corpus_and_play('test_file.txt', 'Big boy')
load_corpus_and_play('test_file.txt', 'Bi', 7)
```

Playing around in the shell will soon get boring so it's a good idea to collect up your tests in a function and call that function from the `test` function. We have provided a function that does just that, ie, the `run_some_trials` function. Feel free to write all your testing code in there so you can simply hit play and run it.

If you want to find some more giant files to feed your tests the you should check out Project Gutenberg (`http://gutenberg.org`) and Wikisource (`http://wikisource.org/`).

Running the Program

Once you have completed all of the missing functions, you can play a game! The game can either run automatically, where it will attempt to guess and match a phrase that you give it; or interactively, where it will ask you to confirm each guess.

To begin with, the program is setup to simply call the `main` function, which in turn just runs the `test` function. You should put whatever tests you want to run in the *test* function. To help with debugging you can set break-points and run in debug mode — allowing you to go through your program line by line and view the contents of variables etc...

```
load_corpus_and_play('corpus.txt', 'dead war')
Will give:
Loading corpus... corpus.txt
Corpus loaded. (1484 characters)
de______  (0)
dea_____  (1)
dead____  (3)
dead ___  (2)
dead w__  (7)
dead wa_  (1)
dead war  (1)
 Solved it in 15 guesses!
Took 0.002092 seconds
```

```
load_corpus_and_play('corpus.txt', 'de' 8)
Will give:
Loading corpus... corpus.txt
Corpus loaded. (1484 characters)
de______  (0)
 'a'? (y/n) y
dea_____  (1)
 'r'? (y/n) n
 'c'? (y/n) n
 'd'? (y/n) y
dead____  (3)
 '.'? (y/n) n
 ' '? (y/n) y
dead ___  (2)
 't'? (y/n) n
 'b'? (y/n) n
 's'? (y/n) n
 'h'? (y/n) n
 'r'? (y/n) n
 'l'? (y/n) n
 'w'? (y/n) y
dead w__  (7)
 'a'? (y/n) y
dead wa_  (1)
 'r'? (y/n) y
dead war  (1)
 Solved it in 15 guesses!
```

You will notice that our code provides the time taken to guess the phrase but only if the game is run in auto-mode. In manual/interactive mode the user will spend various lengths of time responding to prompts and therefore the total time wouldn't represent the actual computer time used.

Extra Information & Submission

The work in this assignment is to be carried out individually, and what you submit needs to be your own work. You must not discuss code-level details with anyone other than the course tutors and lecturers. You are, however, permitted to discuss high-level details of the program with other students. If you get stuck on a programming issue, you are encouraged to ask your tutor or the lecturer for help. *You may not copy material from books, the internet, or other students.* We will be checking carefully for copied work.

If you have a general assignment question or need clarification on how something should work, please use the class forums on Learn, as this enables everyone to see responses to common issues, but NEVER post your own program code to Learn. Remember that the main point of this assignment is for you to exercise what you have learnt from lectures and labs, so make sure you do the relevant labs first, so don't cheat yourself of the learning by having someone else write material for you.

You are strongly advised to submit early versions of your work; not only does this provide a backup in case you lose your material, but also helps to establish you as the original author should someone copy your work and submit it as their own later.

What to submit

The assignment submission topic on the quiz server will provide a quiz for you to submit your code into. Any extra submission requirements will be outlined there.

IMPORTANT NOTE: The submission quiz will only run a few basic tests to ensure your code is ready for marking. *The quiz will not mark your code!* We will mark your code by hand and we **will be marking your style and doctests**. So make sure you add in a good number of doctests to test sensible usage cases and edge cases.

Multi-Choice Questions

We also want you to think about the time complexity of various aspects of the program you have been developing. We will present these questions as multi-choice questions when we release the submission quiz.

- What is the *best case* asymptotic complexity of `FrequencyList.find` method acting on a linked list with n elements?
- What is the *average case* asymptotic complexity of `FrequencyList.__len__` method acting on a linked list with n elements?
- Do the best and worst case asymptotic complexity's of `FrequencyList.remove` method differ?
- Given a corpus of size m and a phrase of size n, what is the *worst case* asymptotic complexity to guess all characters in the phrase?

Looking forward Part 2

As you build and test your assignment, you may find some parts of it are particularly slow or inefficient. Part two will explore ways of removing these inefficiencies. Part 2 will be issued soon. Please email us if you are ready for part 2 and it hasn't been made available yet . . .