COSC122 (2020su2) Assignment

Part 2 – Hash Tables & Sorted Linked Lists

This part of the assignment will explore improving the performance of the algorithm implemented in part one.

Essential Information

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

- *Part 1*: Due 11:59pm on Sunday 10th January 2021 (10% of course mark).
- *Part 2*: Due 11:59pm on Sunday 31st January 2021 (10% of course mark).

The Part 1 deadline is a soft deadline to get feedback. You may submit your work for part 1 at the due date and time of part 2.
Submissions should be made through the quiz server site — submission quizzes will be made available closer to the due time. 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 you haven't made any trivial errors.

Improving on Part 1

As you were implementing and testing part one, you may have noticed a few inefficiencies in how it works.

In particular, looking up each guess (in `filter_possible_chars`) is quite slow; and the performance of it degrades linearly as the size of the corpus increases. Furthermore, it seems a bit silly to walk through the entire corpus and sort the list of characters *every time* we want to guess a letter.

What if, instead of walking through the corpus for each guess, we walk through the corpus only once, at the start, recording the possible characters for *every possible guess*? That is, we build up a list of every possible character pair in the corpus and the possible letters that follow it (a technique known as *pre-caching*).

Using this strategy, we'd be able to make each guess *much* faster, because the list of possible characters would already be known.

Coding Part 2

Hash Tables & Sorted Linked Lists

You have been supplied with the file `shannon_p2.py`, which implements the basic machinery you saw in part one. You only have to complete the marked function stubs (those that currently just contain `pass`). You may add new functions for your convenience, but the existing functions must continue to behave as specified. *It is crucial that you implement these functions* ***exactly*** *as described in their docstrings.*

IMPORTANT: A lot of the work in the part of the assignment will be in getting a good conceptual understanding of the classes provided and their usage. So, don't just dive straight into coding. Try to sketch out an overview of how the classes fit together and what the container types contain, eg, what are the items in the PrefixTable and how are they used. Don't be disheartened if it takes you a while to come to grips with the code, the time isn't wasted. You are practising your code reading which is a very important skill to develop (plus it will help you see why you should write clear, easy to follow, code).

What Needs to be Implemented

Instead of building up a `FrequencyList` for each guess, a similar list will be built for *every possible guess* at the start of the program (in the `process_corpus` function). These lists of possible guesses will be stored in a hash table that will be searched to fetch the lists of guesses.

The `SortedFrequencyList` class serves the same purpose (and exposes the same interface) that `FrequencyList` did in part one, but with the notable exception that the items must be stored in sorted order (from highest frequency to lowest). (If your `FrequencyList` implementation worked well in part one, you might be able to re-use some of your code.)

The `PrefixTable` is a hash table implementation that stores the connections between character prefixes (pairs) and the `SortedFrequencyList` of possible characters that follow it. Each entry in `PrefixTable` stores a prefix (character pair) and the head of the linked list that has the guesses for that character pair. It uses instances of `PrefixItem` (implemented for you) internally to store the association between an prefix and the `SortedFrequencyList` (linked list) of possibilities. `PrefixTable` must use *open addressing* with *linear probing* to store the items based on the hash of the prefix. We have implemented the `__hash__` method for `PrefixItem`s so you can get raw hash values by using the hash function, eg, `raw_hash = hash(a_prefix_item)`.

Finally, you need to implement the `process_corpus` function that takes a corpus and turns it into a fully populated `PrefixTable`. For each prefix (pair of characters) in the corpus, you should see if it is in `PrefixTable`; if it is, the linked list should be updated (increment the count of the character and possibly move it up the list), and if not, a new entry should be created and the character added with a count of 1.

How it All Fits Together

There are many more components that fit together in this part of the assignment, so it might be a bit tricky for you to get your head around it all. At the core of this version of the program is the *process_corpus* function, which will build up a hash table of character prefixes and guesses, similar to that shown below.

An example hash table is given below. The slots are numbered from 0 up to the number of slots minus 1. In each slot there will be a *PrefixItem* object that contains a letter pair and a *SortedFrequencyList* of *Frequency* objects, which each contain a letter and a frequency count. The PrefixItem will be in the form `PfI(letter_pair : SortedFrequencyList)`. The *SortedFrequencyList* will be in the form `SFL(Frequency, Frequency,...)`. *Frequency* items are in the format `<letter: frequency>`.

```
Prefix hash Table
-----------------
0:  None
1:  None
2:  None
...
19: None
20: PfI('ui': SFL(<'d': 2>), SFL(<'e': 1>))
21: None
...
41: None
42: PfI('la': SFL(<'z': 1>))
43: PfI('la': SFL(<'n': 1>))
44: PfI('an': SFL(<'g': 1>))
45: None
...
49: None
50: PfI('li': SFL(<'n': 1>))
51: None
...
```

The `PrefixTable` will use `PrefixItem`s in each slot of the hash table to store the association between the *prefix* and *possibles*. The *possibles* are themselves a *SortedFrequencyList* that uses

Frequency objects to pair *letters* to their *frequency*.

If you're not sure where to start, the best way to tackle this problem is to start from the inside and work your way out. That is, you should start by getting your `SortedFrequencyList` working first because it's free of dependencies and can be tested without worrying about other classes; then work your way out to the `PrefixTable`, and finally the `process_corpus` method that ties it all together.

Testing Your Code

As with part one, 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_p2.py
>>> doctest.testmod(verbose=True)
...
...
Test passed.
TestResults(failed=0, attempted=55)
>>> process_corpus('riff raff', 5)
Prefix hash Table
-----------------
0: PfI(' r': SFL(<'a': 1>))
1: None
2: PfI('ra': SFL(<'f': 1>))
3: None
4: None
5: None
6: PfI('if': SFL(<'f': 1>))
7: None
8: None
9: None
10: PfI('ri': SFL(<'f': 1>))
11: PfI('af': SFL(<'f': 1>))
12: None
13: None
14: None
15: None
16: PfI('ff': SFL(<' ': 1>))
17: None
18: None
19: None
20: None
21: PfI('f ': SFL(<'r': 1>))
22: None
23: None
24: None
```

You should consider the supplied doctests to be *very inadequate*, and test your code with many different inputs that exercise your code thoroughly (and add them as doctest to ensure later modifications don't break something unexpected).

Running the Game

Once you have completed all of the missing functions, the game can be played in the same way that part one can (refer to the handout for instructions).

When running the game, you should notice that loading the corpus takes much longer than it did before, but it can guess the correct letter almost instantly!

Your output when running the game on `corpus.txt`, with `dead war` as the phrase, should be something like the following (with the only difference being the time taken):

```
Loading corpus... corpus.txt
Corpus loaded. (1484 characters)
de______  (0)
dea_____  (1)
dead____  (3)
dead ___  (2)
dead w__  (5)
dead wa_  (2)
dead war  (1)
Solved it in 14 guesses!
Took 0.000245 seconds
```

Reflection

Once you have your code working you should run some trials to see how the number of guesses and the time taken vary in relation to the size of the corpus. Drawing graphs such as the following will help you see the trend:

- number of guesses versus size of corpus
- time taken versus size of the corpus

Using a long sentence as the phrase to guess (eg, `'Hello isnt it a lovely day today.'` or `'The quick brown fox jumps over the lazy dog'`) will be insightful and will help test your code. You should be able to identify the point where increasing the size of the corpus doesn't really improve things, ie, after what corpus size does the number of guesses flatten out? Can you explain why the number of guesses eventually flattens out, ie, why there will eventually come a point where increasing the corpus size does not help.

Does the graph of time vs corpus size look like what you would expect given the method being used to make guesses?

Why does running the tests seem to take a long time even though the actual time taken to guess a phrase is very small?

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 1anyone 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 theclass 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.

Short Answer Questions

The following are short discussion questions that will be marked once the submission quiz has closed. You should consider plotting trial data as appropriate, and discussing any significant results. As the corpus size needs to be varied you could simply slice a large corpus into appropriate sizes.

1. Discuss the relationship between the size of corpus used and the number of guesses made for the phrase 'The quick brown fox jumps over the lazy dog'. For this question we suggest using progressively larger slices of a large corpus, such as War and Peace.

2. Discuss the relationship between the size of corpus used and the number of guesses made for the pseudo random phrase 'jaaugftvfjcyzzmxyosavusezfkfytnlabmnraqybna'. For this question we suggest using progressively larger slices of a large corpus, such as War and Peace.
   Note the above phrase was generated using this function:

   ```
   def random_string(size):
       from random import randint
       return "".join(
           chr(ord('a') + randint(0 , 25)) for _ in range(size)
       )
   ```

3. As the size of the corpus increases the number of guesses tends for a phrase to decrease but would it be a good idea to use a corpus that combines all the text from all the 60,000+ books on Project Gutenberg? Explain why or why not. Looking at the overall time taken might give you some clues here.