FIT2004 S1/2020: Assignment 3 - Tries

DEADLINE: Sunday 24th May 2020 23:55:00 AEST

LATE SUBMISSION PENALTY: 10% penalty per day. Submissions more than 7 days late are generally not accepted. The number of days late is rounded up, e.g. 5 hours late means 1 day late, 27 hours late is 2 days late. For special consideration, please complete the online special consideration form.

PROGRAMMING CRITERIA: It is required that you implement this exercise strictly using **Python programming language** (version should not be earlier than 3.5). This practical work will be marked on the time complexity, space complexity and functionality of your program.
Your program will be tested using automated test scripts. It is therefore critically important that you name your files and functions as specified in this document. If you do not, it will make your submission difficult to mark, and you will be penalised.

SUBMISSION REQUIREMENT: You will submit a single python file, `assignment3.py`.

PLAGIARISM: The assignments will be checked for plagiarism using an advanced plagiarism detector. Last year, many students were detected by the plagiarism detector and almost all got zero mark for the assignment and, as a result, many failed the unit. "Helping" others is NOT ACCEPTED. Please do not share your solutions partially or/and completely to others. If someone asks you for help, ask them to visit us during consultation hours for help.

Learning Outcomes

This assignment achieves the Learning Outcomes of:

- 1) Analyse general problem solving strategies and algorithmic paradigms, and apply them to solving new problems;
- 2) Prove correctness of programs, analyse their space and time complexities;
- 4) Develop and implement algorithms to solve computational problems.

In addition, you will develop the following employability skills:

- Text comprehension
- Designing test cases
- Ability to follow specifications precisely

Assignment timeline

In order to be successful in this assessment, the following steps are provided as a **suggestion**. This is an approach which will be useful to you both in future units, and in industry.

Planning

1. Read the assignment specification as soon as possible and write out a list of questions you have about it.
2. Clarify these questions. You can go to a consultation, talk to your tutor, discuss the tasks with friends or ask in the forums.
3. As soon as possible, start thinking about the problems in the assignment.
    - It is strongly recommended that you **do not** write code until you have a solid feeling for how the problem works and how you will solve it.
4. Writing down small examples and solving them by hand is an excellent tool for coming to a better understanding of the problem.
    - As you are doing this, you may see patterns which allow you to deduce the correct algorithm to solve the problem.
    - You will also get a feel for the kinds of edge cases you will need to handle.
5. Write down a high level description of the algorithm you will use.
6. Determine the complexity of your algorithm idea, ensuring it meets the requirements.

Implementing

1. Think of test cases that you can use to check if your algorithm works.
   - Use the edge cases you found during the previous phase to inspire your test cases.
   - It is also a good idea to generate large random test cases.
   - Sharing test cases **is** allowed, as it is not helping solve the assignment. Check the forums for test cases!
2. Code up your algorithm, (remember decomposition and comments) and test it on the tests you have thought of.
3. Try to break your code. Think of what kinds of inputs you could be presented with which your code might not be able to handle.
   - Large inputs
   - Small inputs
   - Inputs with strange properties
   - What if everything is the same?
   - What if everything is different?
   - etc...

Before submission

- Make sure that the input/output format of your code matches the specification.
- Make sure your filenames match the specification.
- Make sure your functions are named correctly and take the correct inputs.

Documentation (3 marks)

For this assignment (and all assignments in this unit) you are required to document and comment your code appropriately. This documentation/commenting must consist of (but is not limited to)

- For each function, high level description of that function. This should be a one or two sentence explanation of what this function does. One good way of presenting this information is by specifying what the input to the function is, and what output the function produces (if appropriate)
- For each function, the Big-O complexity of that function, in terms of the input. Make sure you specify what the variables involved in your complexity refer to. Remember that the complexity of a function includes the complexity of any function calls it makes.
- Within functions, comments where appropriate. Generally speaking, you would comment complicated lines of code (which you should try to minimise) or a large block of code which performs a clear and distinct task (often blocks like this are good candidates to be their own functions!).

Example of usage

In this assignment you will be creating a Trie class, and implementing methods for that class. This example is to clarify how your class will be called in the testing code. In other words, you should be able to call the methods of your class in the way shown below.

```
#In your file...
class Trie:
    #contents of your class

#In the marker's tester code...
from assignment3 import Trie #assignment3 is your .py file, Trie is the name of the class
text = ["example_string", "another_example", ...] #this is the test input
student_trie = Trie(text) #this runs task 1, building a Trie
string_frequency = student_trie.string_freq("abc") #this runs task 2
prefix_frequency = student_trie.prefix_freq("abb") #this runs task 3
wildcard_prefix_frequency = student_trie.wildcard_prefix_freq("ab?") #this runs task 4
```

1 Trie (6 marks)

In this task, you will preprocess a text to make it ready for the queries you will run in later tasks. To do this you will build a trie containing the words in the text and other information. You must create a `Trie` class, and write the `__init__(self, text)` function, which generates a `Trie` object.

1.1 Input

A list of strings each of which will be composed only of lowercase English alphabet characters. There will be no empty strings in this list, but the list may be empty, and may contain duplicates. The contents of this list will be referred to as the **`text`** throughout this assignment.

1.2 Output

For this task, no output is required. Note that the other three tasks in this assignment all involve implementing methods of the `Trie` class, so be sure to read those tasks carefully, and think about what your `__init__` function will need to do in order to allow you to solve the later tasks in the appropriate time complexities.

1.3 Complexity

Given an input list with total character T over all strings in the list, your `__init__` function must run in $O(T)$ time.

In addition, you will be penalised for significant needless inefficiencies in your code (e.g. a slowdown with a factor of 26) even if they do not result in a worsening of the complexity.

2 String frequency (6 marks)

Given a word, we want to know how many times that word occurred in the **`text`**. To solve this, you will write a method for the `Trie` class, `string_freq(self, query_str)`

2.1 Input

The input, `query_str` is a non-empty string consisting only of lowercase English alphabet characters.

2.2 Output

`string_freq` returns an integer, which is the number of elements of the **`text`** which are exactly `query_string`. Do not count instances where `query_string` occurs as a proper substring of some element of the text. Return 0 if `query_str` does not match any elements of the text.

Example:
If a `Trie` was constructed from the following **`text`**

```
aa
aab
aaab
abaa
aa
abba
aaba
aaa
aa
aaab
abbb
baaa
baa
bba
bbab
```

calling the `string_freq` method with `"aa"` as the input would return 3 (note that strings like `"aab"` are not counted, even though `"aa"` is a substring of them.)

2.3 Complexity and speed

Given a `query_str` of length q, `string_freq` must run in O(q) time.

In addition, you will be penalised for significant needless inefficiencies in your code (e.g. a slowdown with a factor of 26) even if they do not result in a worsening of the complexity.

3 Prefix frequency (6 marks)

Given a string, we want to know how many words in the **text** have that string as a prefix. To solve this, you will write a method for the `Trie` class, `prefix_freq(self, query_str)`

3.1 Input

The input, `query_str` is a possibly empty string, with every character in the string being a lowercase English alphabet character.

3.2 Output

`prefix_freq` returns an integer, which is the number of words in the **text** which have `query_str` as a prefix.

Example:
If a `Trie` was constructed from the following **text**

```
aa
aab
aaab
abaa
aa
abba
aaba
aaa
aa
aaab
abbb
baaa
baa
bba
bbab
```

calling the `string_freq` method with `"aa"` as the input would return 8 (referring to `"aa"`, `"aab"`, `"aaab"`, `"aa"`, `"aaba"`, `"aaa"`, `"aa"`, `"aaab"`)

3.3 Complexity and speed

Given a `query_str` of length q, `prefix_freq` must run in O(q) time. Think of what information you need to store in your Trie to achieve this.

In addition, you will be penalised for significant needless inefficiencies in your code (e.g. a slowdown with a factor of 26) even if they do not result in a worsening of the complexity.

4 Wildcard prefix search (9 marks)

Given a string containing a single wildcard, we want to know which strings in the **text** have that string as a prefix. To solve this, you will write a method for the `Trie` class, `wildcard_prefix_freq(self, query_str)`

4.1 Input

The input, `query_str` is a non-empty string consisting only of **lowercase English alphabet characters** (possibly no characters), and **exactly one '?'** character, representing a wildcard. A wildcard can match any single character, and it must match exactly one character. For example, the string `?ot` matches the words `bot, cot, dot, hot ....` The wildcard can occur at any position in `query_str`.

4.2 Output

`wildcard_prefix_freq` returns a list containing all the strings which have a prefix which matches `query_str`. These strings must be in **lexicographic order**.

Example:
If a `Trie` was constructed from the following **text**

```
aa
aab
aaab
abaa
aa
abba
aaba
aaa
aa
aaab
abbb
baaa
baa
bba
bbab
```

calling the `wildcard_prefix_freq` method with `"aa?"` as the input would return the list `["aaa", "aaab", "aaab", "aab", "aaba"]`. Note that `"aa?"` does not match `"aa"`.

4.3 Complexity and speed

Given a `query_str` of length q, `wildcard_prefix_freq` must run in $O(q + S)$ time. S is the total number of characters in all strings of the **text** (inclusive of duplicates) which have a prefix matching `query_str`.

In addition, you will be penalised for significant needless inefficiencies in your code (e.g. a slowdown with a factor of 26) even if they do not result in a worsening of the complexity.

Warning

For all assignments in this unit, you may **not** use python **dictionaries** or **sets**. This is because the complexity requirements for the assignment are all deterministic worst case requirements, and dictionaries/sets are based on hash tables, for which it is difficult to determine the deterministic worst case behaviour.

Please ensure that you carefully check the complexity of each inbuilt python function and data structure that you use, as many of them make the complexities of your algorithms worse. Common examples which cause students to lose marks are **list slicing**, inserting or deleting elements **in the middle or front of a list** (linear time), using the `in` keyword to **check for membership** of an iterable (linear time), or building a string using **repeated concatenation** of characters. Note that use of these functions/techniques is **not forbidden**, however you should exercise care when using them.

These are just a few examples, so be careful. Remember, you are responsible for the complexity of every line of code you write!