FIT2004 S2/2020: Assignment 3 - Tries and Trees

DEADLINE: Friday 16th October 2020 23:55:00 AEST

LATE SUBMISSION PENALTY: 10% penalty per day. Submissions more than 5 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 visit this page:
`https://www.monash.edu/connect/forms/modules/course/special-consideration`
and fill out the appropriate form. **Do not** contact the unit directly, as we cannot grant special consideration unless you have used the online form.

PROGRAMMING CRITERIA: It is required that you implement this exercise strictly using the **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, and your documentation.
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 named `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 completely to others. If someone asks you for help, ask them to visit a consultation 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;
- 3) 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.
4. It is often helpful to solve the relevant tutorial problems, or review them. You can watch the tutorial solution videos for a detailed explanation.
   - 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.
5. 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 will also get a feel for the kinds of edge cases your code will have to deal with.
6. Write down a high level description of the algorithm you will use.
7. 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.
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 filename matches the specification.
- Make sure your functions are named correctly and take the correct inputs.

Documentation and submission (4 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!).

You are also required to submit your assignment in the format specified.

1 Constructing strings (13 marks)

In this task we consider the problem of constructing a string from pieces of another string. Given a string S and another string T, we want to find a set of substrings of S which, when concatenated, equal T. Specifically, we want to find the smallest such set.

To solve this problem, you will write a function `build_from_substrings(S, T)`

1.1 Input

Two strings, S and T. The strings will consist only of lowercase a-z characters, and both will be non-empty, and can be of any size.

1.2 Output

If T can be constructed from substrings of S, `build_from_substrings` should return a list of tuples, where each tuple represents a substring. The first element in each tuple is the index of the first character in the substring, and the second element is the index of the last character in the substring.

These tuples should be in order, i.e. if the tuples are concatenated in the order they are given in the output, the result should be T.

The values in the output should correspond to the smallest (fewest elements) set of substrings of S, when concatenated, equal T. In other words, you should have as few tuples in your output as possible.

If multiple smallest sets exist, you may return any of them (but only one).

If T cannot be constructed from substrings of S, `build_from_substrings` should return `False`.

1.3 Example

```
>>> build_from_substrings("abbcc", "bccabcca")
[(2, 4), (0, 1), (3, 4), (0, 0)]
>>> build_from_substrings("abbcc", "bccdabcca")
False
```

1.4 Complexity

`build_from_substrings` must run in $O(N^2 + M)$ where

- N is the number of characters in S
- M is the number of characters in T

This time complexity is not optimal. As usual, you will not lose marks for solving it in a lower time complexity.

2 Position in a text (13 marks)

In this task we consider the problem of determining the position of a word in a text, with respect to alphabetical ordering. Given a text and a query word, we need to find the number of words in the text which are alphabetically less than the query.

To solve this problem you will write a function `alpha_pos(text, query_list)`

2.1 Input

Two lists of strings, `text` and `query_list`. Every string in each list will only contain lowercase a-z characters, and will be non-empty. Note that the words in `query_list` may or may not appear in `text`.

2.2 Output

A list of integers, where the i^{th} integer is the number of words in `text` which are alphabetically less than `query_list[i]`.

2.3 Example

```
>>> alpha_pos(["bac","aaa","baa","aac"],
              ["ba", "aab", "zaa", "aa", "baa", "b"])
[2, 1, 4, 0, 2, 2]
```

2.4 Complexity

`alpha_pos` must run in $O(C + Q)$ where

- C is the total number of characters in `text`
- Q is the total number of characters in `query_list`

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!