FIT2004 S1/2021: Assignment 3 - Tries and Trees

DEADLINE: Friday 1st October 2021 23:55:00 AEST

LATE SUBMISSION PENALTY: 10% penalty per day. Submissions more than 7 calendar days late will receive 0. 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 **or** use the google form linked on the Moodle page for short (< 5) day extensions.

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 two python files, `task1.py` and `avl_tree.py`, and a single pdf, `discussion.pdf`

PLAGIARISM: The assignments will be checked for plagiarism using an advanced plagiarism detector. In previous semesters, 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 to solve the assignment 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;
- 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 will also get a feel for the kinds of edge cases your code will have to deal with.
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.
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.
- Make sure you zip your files correctly (if required)

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!).

1 Lexical position (15 marks)

Consider the problem of determining how many words in a text appear lexicographically later than a given word. To solve this problem, you will write a function `lex_pos(text, queries)`.

1.1 Input

`text` is an unsorted list of strings. Each string contains only lowercase a-z characters. `text` **can** contain duplicates.

`queries` is a list of strings consisting only of lowercase a-z characters. Each string in `queries` is a prefix of some string in `text` (note that a string is a prefix of itself).

1.2 Output

`lex_pos` outputs a list of numbers. The i^{th} number in this list is the number of words in `text` which are lexicographically greater than the i^{th} element of `queries`

1.3 Example

```
text = ["aaa","bab","aba","baa","baa","aab","bab"]
queries = ["", "a", "b", "aab"]
lex_pos(text, queries)
>>> [7, 7, 4, 5]
```

1.4 Complexity

Remember that string comparison **is not** considered $O(1)$ in this unit.

`lex_pos` should run in $O(T + Q)$ time, where

- T is the sum of the number of characters in all strings in `text`
- Q is the total number of characters in `queries`

2 Editing databases (12 marks)

Suppose we have two lookup tables (stored as AVL trees). We want to create a new lookup table which contains all the data from both, except we have discovered that some data in one of the tables are corrupted, and we want to exclude those from the result. The number of corrupted items is very small relative to the total number of items in either table.

Importantly, all the keys in one of the tables are lesser than all the keys in the other. Corrupted items only appear in the table with lesser keys.

Nathan has figured out a way to solve the problem using the normal AVL tree insert and delete operations:

For each corrupted element, delete it from the corrupted table. Once all the corrupted elements have been removed from this table, insert each remaining element into the other table.

Your task is to come up with a more efficient way to solve this problem. You will be given code for an `AVLtree` class, with the methods `insert(key)` and `delete(key)` already implemented. You need to implement a method of the `AVLTree` class, `uncorrupted_merge(self, other, corrupted)` which solves this problem.

2.1 Input

`other` is an instance of `AVLTree`. Note that since this is a method of `AVLTree`, self is also an instance of `AVLTree`.

`corrupted` is a list of keys. Every item in `corrupted` appears in `other`. The number of items in `corrupted` is much smaller than the number of items in `other` and `self`

2.2 Output

`uncorrupted_merge` modifies `self` so that it contains all the elements it originally contained, as well as all the elements in `other` which are not in `corrupted`. At the end, it must still be a valid AVL tree.

2.3 AVL Tree Class

Download the AVL tree implementation found here. You may **not** modify any of the existing methods, but you may add new methods (in addition to the required `uncorrupted_merge`). You must submit a modified version of this file as your `avl_tree.py`.

To test your code, we will create two instances of `AVLTree` from `avl_tree.py`, and then call `uncorrupted_merge`.

It is important that your implementation **not** rely on **modifying** the existing code in the AVL tree implementation, since we need to know how to access the internals of the tree for testing purposes.

2.4 Example

```
from avl_tree import AVLTree

#Assume t1 and t2 are instances of AVLTree, containing the following elements:
#t1: 1, 2, 3, 4, 5
#t2: 6, 7, 8, 9, 10

corrupted = [1,3,5]
t2.uncorrupted_merge(t1, corrupted)
#t2 will now contain 2,4,6,7,8,9,10, and be a valid AVL Tree
```

2.5 Discussion

You must submit a pdf, `discussion.pdf`. In this pdf, answer the following questions. Answer each question under a different heading, to keep your answers clearly separated:

1. Explain the high level idea of your algorithm (this should only be a few sentences)
2. Give the complexity of your implementation of `uncorrupted_merge`. Explain why it has this complexity. Be sure to define any variables you use in your complexity (other than the ones defined in section 2.6)
3. Justify that your algorithm works. In other words, explain why the output is a valid AVL tree.
4. Suppose we now relax the constraint that the number of items in `corrupted` is much smaller than the number of items in `t1` and `t2`. Does this change the relative efficiencies of Nathan's approach and your solution? If so, does Nathan's approach ever become more efficient, and roughly when would this occur?

2.6 Complexity

For this task, we are not providing a time complexity. Your solution must be more efficient (i.e. have a better complexity) than Nathan's solution.

In your analysis of your code, let N be the number of nodes in the larger of the two AVL trees. Let k be the number of keys in `corrupted`. We assume that $k << n$ for the inputs you will recieve (but not when we consider the scenario presented in question 4). $<<$ means much smaller, in other words, k is not $\Theta(N)$.

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!