FIT2004 S2/2020: Assignment 3 - Tries and Trees

DEADLINE: Friday 7^{th} May 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. **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, `assignment3.py`

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 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 DNA fragments (9 marks)

You are maintaining a database for a study into drug resistance in hospitals. During the study, you will receive **bacterial DNA** collected from patients. Each **bacterial DNA** sample contains a **drug resistant gene sequence** at the **start**.

The researchers will need to query your database for various different **drug resistant gene sequences** over the course of the study (details below), but you will also need to be able to update the contents of the database with new **bacterial DNA** samples.

To solve this problem, you will need to create a **class** `SequenceDatabase`, to represent the **database**. This class will need to have two methods, `addSequence(s)` and `query(q)`.

As usual, you are welcome to create other functions/methods.

Note: DNA is typically represented using A, T, C and G. For the purpose of easy coding, we will use A, B, C and D (since they have adjacent ASCII values).

1.1 Input

The input to `addSequence` is a single nonempty string of uppercase letters in uppercase [A-D].

The input to `query` is a single (possibly empty) string of uppercase letters in uppercase [A-D].

1.2 Output

`addSequence(s)` should not return anything. It should appropriately store `s` into the database represented by the instance of `SequenceDatabase`. We define the **frequency** of a particular string to be the number of times it has been added to the database in this way.
`query(q)` should return a string with the following properties:

- It must have `q` as a prefix
- It should have a higher **frequency** in the **database** than any other string with `q` as a prefix
- If two or more strings with prefix `q` are tied for most frequent, return the lexicographically least of them

If no such string exists, `query` should return `None`.

1.3 Example

```
db = SequenceDatabase()
db.addSequence("ABCD")
db.addSequence("ABC")
db.addSequence("ABC")
db.query("A")
>>> "ABC"
db.addSequence("ABCD")
db.query("A")
>>> "ABC"
db.addSequence("ABCD")
db.query("A")
>>> "ABCD"
de.query("B")
>>> None
```

Note that the name `"db"` in the above example is arbitrary (i.e. the instance of `SequenceDatabase` could be called anything)

1.4 Complexity

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

- The `__init__` method of `SequenceDatabase` should run in $O(1)$
- `addSequence(s)` should run in $O(len(s))$
- `query(q)` should run in $O(len(q))$. Note that although you need to return a string longer than $len(q)$, since strings are pass-by-reference, returning a string of any length is $O(1)$.

2 Open reading frames (8 marks)

In Molecular Genetics, there is a notion of an Open Reading Frame (ORF). An ORF is a portion of DNA that is used as the blueprint for a protein. All ORFs start with a particular sequence, and end with a particular sequence.

In this task, we wish to find all sections of a genome which start with a given sequence of characters, and end with a (possibly) different given sequence of characters.

To solve this problem, you will need to create a class `OrfFinder`. This class will need a method, `find(start, end)`. Also note that the constructor for this class takes a string `genome` as a parameter, unlike the class in Problem 1 (shown in the example below).

2.1 Input

`genome` is a single non-empty string consisting only of uppercase [A-D]. `genome` is passed as an arguement to the `__init__` method of `OrfFinder` (i.e. it gets used when creating an instance of the class).

`start` and `end` are each a single non-empty string consisting of only uppercase [A-D].

2.2 Output

`find` returns a list of strings. This list contains all the substrings of `genome` which have `start` as a prefix and `end` as a suffix. There is no particular requirement for the order of these strings. `start` and `end` must not overlap in the substring (see the last two cases of the example below).

2.3 Example

```
genome1 = OrfFinder("AAABBBCCC")
genome1.find("AAA","BB")
>>> ["AAABB","AAABBB"]
genome1.find("BB","A")
>>>[]
genome1.find("AA","BC")
>>>["AABBBC","AAABBBC"]
genome1.find("A","B")
>>> ["AAAB","AAABB","AAABBB","AAB","AABB","AABBB","AB","ABB","ABBB"]
genome1.find("AA","A")
>>> ["AAA"]
#note that "AA" is not valid, since start and end would need to overlap
genome1.find("AAAB","BBB")
>>> []
# note that "AAABBB" is not valid, since start and end would need to overlap
```

2.4 Complexity

- The `__init__` method of `OrfFinder` must run in $O(N^2)$ time, where N is the length of genome.
- `find` must run in $(len(start) + len(end) + U)$ time, where U is the number of characters in the output list.

As an example of what the complexity for `find` means, consider a string consisting of $\frac{N}{2}$ "B"s followed by $\frac{N}{2}$ "A"s.
If we call `find("A","B")`, the output is empty, so U is $O(1)$. On the other hand, if we call `find("B", "A")` then U is $O(N^2)$.

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!