FIT2004 S2/2022: Assignment 3

DEADLINE: Friday 21st October 2022 16:30:00 AEDT.

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://forms.monash.edu/special-consideration` and fill out the appropriate 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 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;
- 3) Compare and contrast various abstract data types and use them appropriately;
- 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

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 two or three sentence explanation of what this function does and the approach undertaken within the function.
- For each function, specify what the input to the function is, and what output the function produces or returns (if appropriate).
- For each function, the Big-O time and space complexity of that function, in terms of the input size. 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!).

A suggested function documentation layout would be as follows:

```
def my_function(argv1, argv2):
    """
    High level description about the functiona and the approach you
    have undertaken.
    :Input:
        argv1:
        argv2:
    :Output, return or postcondition:
    :Time complexity:
    :Aux space complexity:
    """
    # Write your codes here.
```

1 Sharing the Meals (10 marks)

You and your 4 housemates eat breakfast and dinner together every day, and share the load of preparing the meals.

Now it is time for preparing the schedule of who is responsible for preparing each meal in the next n days (numbered $0, 1, \ldots, n-1$). Ideally, the five of you would like to divide the task in such a way that each meal is assigned to one person, no person is assigned to prepare both meals of the same day, and each person is assigned to exactly $2n/5$ meals (so that the load is perfectly distributed among you).

However, there are some complications:

- Perhaps, $2n/5$ is not an integer number.
- You all have busy schedules and only have time availability to prepare meals on specific times.

To solve the problem, you initially collected the data about the time availability of each person. The five persons will be numbered $0, 1, 2, 3, 4$. You get as input a list of lists `availability`. For a person numbered j and day numbered i, `availability[i][j]` is equal to:

- 0, if that person has neither time availability to prepare the breakfast nor the dinner during that day.
- 1, if that person only has time availability to prepare the breakfast during that day.
- 2, if that person only has time availability to prepare the dinner during that day.
- 3, if that person has time availability to prepare either the breakfast or the dinner during that day.

After some conversations, you agree that a perfect allocation might not be possible, and someone might have to prepare more meals than others. Moreover, you realised that you might have to order some meals from a restaurant. Nevertheless, you want to achieve a fair distribution and not order many meals from restaurants, so you agreed on the following constraints:

- Every meal is either allocated to exactly one person or ordered from a restaurant.
- A person is only allocated to a meal for which s/he has time availability to prepare.
- Every person should be allocated to at least $\lfloor 0.36n \rfloor$ and at most $\lceil 0.44n \rceil$ meals.
- No more than $\lfloor 0.1n \rfloor$ meals should be ordered from restaurants.
- No person is allocated to both meals of a day. There are no restrictions on ordering both meals of a day if the other constraints are satisfied.

As the computer scientist in the house, your housemates asked you to design a program to do the allocation. And your housemates are fine with any allocation you give to them (even if it favours you) as long as it satisfy the constraints above!

To solve this problem, you should write a function `allocate(availability)` that returns:

- `None` (i.e., Python NoneType), if an allocation that satisfy all constraints does not exist.
- Otherwise, it returns `(breakfast, dinner)`, where lists `breakfast` and `dinner` specify a valid allocation. `breakfast`$[i] = j$ if person numbered j is allocated to prepare breakfast on day i, otherwise `breakfast`$[i] = 5$ to denote that the breakfast will be ordered from a restaurant on that day. Similarly, `dinner`$[i] = j$ if person numbered j is allocated to prepare dinner on day i, otherwise `dinner`$[i] = 5$ to denote that the dinner will be ordered from a restaurant on that day.

1.1 Example

Consider the following example in which the function returns one valid allocation for the specified input.

```
# Example
availability = [[2, 0, 2, 1, 2], [3, 3, 1, 0, 0],
[0, 1, 0, 3, 0], [0, 0, 2, 0, 3],
[1, 0, 0, 2, 1], [0, 0, 3, 0, 2],
[0, 2, 0, 1, 0], [1, 3, 3, 2, 0],
[0, 0, 1, 2, 1], [2, 0, 0, 3, 0]]

>>>  allocate(availability)
([3, 2, 1, 4, 0, 2, 3, 2, 2, 3], [4, 0, 3, 2, 5, 4, 1, 1, 3, 0])
```

Note that are other possible allocations for this specific input. It is fine to return any of the allocations that satisfy all constraints! For the curious ones: in every valid solution for this example there is one meal that is ordered from a restaurant.

1.2 Complexity

Your solution should have a worst-case time complexity of $O(n^2)$.

2 Similarity Detector (10 marks)

You are writing a text similarity detector which compares pairs of short text answers submitted by students for some assessment, and detects those submissions which show unusual levels of similarity to other submissions.

You can assume that all of the text submissions have been preprocessed to remove any punctuation and non-alphabetic characters other than spaces, and have been converted to all lowercase characters. In other words, the alphabet consists of the 26 lower case characters from a to z, plus the space character.

For this task, you must write a Python function called `compare_subs(submission1, submission2)` which will use a retrieval data structure to compare two submissions and determine their similarity. The input, output and complexity for this function is discussed in their individual sections Section 2.1, Section 2.2 and Section 2.3 respectively.

2.1 Input

Your program takes as input two submissions, `submission1` and `submission2`. Each submission is given in the form of a string containing only characters in the range [a-z] or the space character.

For example:

```
>>> submission1 = 'the quick brown fox jumped over the lazy dog'
>>> submission2 = 'my lazy dog has eaten my homework'
>>> compare_subs(submission1, submission2)
```

2.2 Output

The function `compare_subs(submission1, submission2)` returns a list of findings with three elements:

- the longest common substring between `submission1` and `submission2`
- the similarity score for `submission1`, expressed as the percentage of `submission1` that belongs to the longest common substring (rounded to the nearest integer[1]), and
- the similarity score for `submission2`, expressed as the percentage of `submission2` that belongs to the longest common substring (rounded to the nearest integer)

[1] if the fractional part of the number is exactly 0.5, then it is rounded up

Consider the following example inputs and outputs:

```
"""
Example 1
"""

>>> submission1 = 'the quick brown fox jumped over the lazy dog'
>>> submission2 = 'my lazy dog has eaten my homework'
>>> compare_subs(submission1, submission2)
[' lazy dog', 20, 27]
```

The longest common substring between the two submissions is `' lazy dog'`, which is 9 characters long. This is 20% of the total length of `submission1`, and 27% of the total length of `submission2`.

```
"""
Example 2
"""
>>> submission1 = 'radix sort and counting sort are both non
...  comparison sorting algorithms'
>>> submission2 = 'counting sort and radix sort are both non
...  comparison sorting algorithms'
>>> compare_subs(submission1, submission2)

[' sort are both non comparison sorting algorithms', 68, 68]
```

The longest common substring between the two submissions is `' sort are both non comparison sorting algorithms'`, which is 48 characters long. This is 68% of the total length of both `submission1` and `submission2` (since both strings have the same total length of 71 characters).

2.3 Complexity and Algorithm Requirements

Let M and N denote the length of strings `submission1` and `submission2`, respectively. The function `compare_subs(submission1, submission2)` should consist of two processes (each of which should be defined in separate functions):

- building a suffix tree, which should be done in a time and space complexity of $O((N+M)^2)$, and
- computing the comparison between the two strings, which should be done in a time and space complexity of $O(N+M)$.

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 in-built 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.

Please be reasonable with your submissions and follow the coding practices you've been taught in prior units (for example, modularising functions, type hinting, appropriate spacing). While not an otherwise stated requirement, extremely inefficient or convoluted code will result in mark deductions.

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