FIT2004 S1/2022: Assignment 4

DEADLINE: Friday 27^{th} May 2022 16:30:00 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://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, `assignment4.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 Sleepless Sysadmins (4 marks + 1 mark for documentation)

You work on a big tech company and are responsible for defining the schedules of the system administrators that will be on duty in the night shifts for the next 30 nights (numbered $0, 1, \ldots, 29$).

There is a set of n system administrators (numbered $0, 1, \ldots, n-1$) that are available for taking some night shifts and get paid a higher hourly rate. But each of those system administrators have their own preferences regarding the nights they want to work (and the ones they don't want to work).

You are given as input a list of lists `preferences`. Each interior list represents a different day. `preferences[i][j]` is equal to 1 if the sysadmin numbered j is interested in working in the shift of the night numbered i, and `preferences[i][j]` is equal to 0 otherwise.

Your company established the following policies:

- There is an integer parameter `sysadmins_per_night` that specifies the exact number of sysadmins that should be on duty each night (the same number of sysadmins will be needed on each night). It holds that $0 \leq$ `sysadmins_per_night` $\leq n$.
- There is an integer parameter `max_unwanted_shifts`, and each of the n sysadmins should be allocated to at most `max_unwanted_shifts` night shifts for which s/he was not interested. It holds that $0 \leq$ `max_unwanted_shifts` ≤ 30.
- There is an integer parameter `min_shifts`, and each of the n sysadmins should be allocated to at least `min_shifts` night shifts. It holds that $0 \leq$ `min_shifts` ≤ 30.
- As long as you comply with the three constraints above, you are free to allocate one sysadmin to as many night shifts as you want.

To solve this problem, you should write a function `allocate(preferences, sysadmins_per_night, max_unwanted_shifts, min_shifts)` that returns:

- `None` (i.e., Python NoneType), if an allocation that satisfy all constraints does not exist.
- Otherwise, it returns a list of lists `allocation`. `allocation[i][j]` should be equal to 1 if the sysadmin numbered j is allocated to work on the the night numbered i, and `allocation[i][j]` should be equal to 0 if the sysadmin numbered j is not allocated to work on the the night numbered i.

1.1 Complexity

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

2 A Chain of Events (4 marks + 1 mark for documentation)

There is usually a tragic backstory to every villain – you as Dr Weird want to understand why Master X turned to the dark side. In order to do so, you retrieve the timelines of Master X being evil; with focus on the chain of events that caused him extreme sadness.

For timeline 1:

- Event 1: Master X lost Daisy, his pet rabbit.
- Event 2: Master X lost 10 games of games against Professor Chaos.
- Event 3: Master X's taco fell to the floor on Taco Tuesday.

For timeline 2:

- Event 1: Master X got chased by a scary clown holding a red balloon.
- Event 2: Master X lost 10 games of games against Professor Chaos.
- Event 3: Master X's taco fell to the floor on Taco Tuesday.
- Event 4: Master X pet hamster Hammy bit him.
- Event 5: Master X couldn't return home because the Tree Sentinel is blocking his way.

For timeline 3:

- Event 1: Master X watched the final season of Game of Thorns.
- Event 2: Master X got chased by a scary clown holding a red balloon.
- Event 3: Master X lost 10 games of games against Professor Chaos.
- Event 4: Master X's taco fell to the floor on Taco Tuesday.

From the 3 timelines above, the longest shared chained events for all 3 events are (1) Master X lost 10 games of games against Professor Chaos; followed by (2) Master X's taco fell to the floor. These are the 2 events that drove Master X to despair and to be the villain that he is today in all 3 timelines.

There could be countless timelines, and each timeline could have countless events. Thus, you once again harness the magic of Algorithms and Data Structures! To do so, you turn this problem into a **trie**; or more specifically a generalised suffix trie[1] using encoding magic where each timeline is turned into a word consisting of events as characters. Thus:

- Timeline 1 as `abc`
- Timeline 2 as `dbcef`
- Timeline 3 as `gdbc`
- ... and the longest chain of events that drove Master X mad in all 3 timelines are events `bc`.
- ... and the longest chain of events that drove Master X mad in 2 timelines for example are events `dbc`.

[1] A suffix trie consisting suffixes of multiple words.

2.1 Generalised Suffix Trie Data Structure (2 mark)

You must write a class `EventsTrie` that encapsulates the all of the timelines and their corresponding events. The `__init__` method of `EventsTrie` would take as an input a list of timelines, `timelines`, represented as a list of strings with:

- N timelines, where N is a positive integer.
- The longest timeline would have M events, represented by M characters. M is a positive integer.
- Events can occur more than once in a single timeline. Thus, a string of `baacbb` is valid.
- It is possible for more than a single timeline to have the same events. Have a look at the example in Section 2.4.
- You can assume that there is only a maximum of 26 unique events in total, represented as lower case characters from **a** to **z**.

```
# The EventsTrie class structure
class EventsTrie:
    def __init__(self, timelines):
        # ToDo: Initialize the trie data structure here
    def getLongestChain(self, noccurence):
        # ToDo: Performs the operation needed to find the chain of events
        occurencing at least noccurence time.
```

For the example stated earlier, the trie will contain all the suffixes of strings `abc`, `dbcef` and `gdbc`.

Consider implementing a `Node` class to help encapsulate additional information that you might want to store in your `EventsTrie` class to help solve this problem.

2.2 Finding Chain Events (2 mark)

You would now proceed to implement `getLongestChain(self, noccurence)` as a function within the `EventsTrie` class. The function accepts 1 argument:

- `noccurence` is a positive integer in the range 1 to N.

The function would then return a string that represents the longest chain of events that occurs in at least `noccurence` timelines.

- If such a chain exists, return the chain as a string.
- If multiple occurrences of such a chain exist, you can return any of the chains as long as it is the longest.
- If such a chain does not exist, return `None`.

2.3 Complexity

The complexity for this task is separated into 2 main components.

The `__init__(timelines)` constructor of `EventsTrie` class would run in $O(NM^2)$ time and space where:

- N is the number of timelines in `timelines`.
- M is the number of events in the longest timeline.

The `getLongestChain(self, noccurence)` of the `EventsTrie` class would run in $O(K)$ time where:

- K is the length of the longest event chain that occur at least in `noccurence` number of timelines.

2.4 Example

Consider the following example from the example given in the description.

```
# Example 1
# Example in the description
timelines = ["abc", "dbcef", "gdbc"]

# Creating a EventsTrie object
mytrie = EventsTrie(timelines)
```

Running `getLongestChain(self, noccurence)` would return different results depending on the `noccurence` value. The 3 simple examples provided are shown below:

```
# Example 1.1 All timeline
noccurence = 3

>>> mytrie.getLongestChain(noccurence)
bc
```

```
# Example 1.2 At least 2 timeline
noccurence = 2

>>> mytrie.getLongestChain(noccurence)
dbc
```

```
# Example 1.3 At least 1 timeline
noccurence = 1

>>> mytrie.getLongestChain(noccurence)
dbcef
```

In the following example, we have more timelines.

```
# Example 2
timelines = ["abaaac", "dbce", "aabcba", "dbce", "caaaa"]

# Creating a EventsTrie object
mytrie = EventsTrie(timelines)
```

In the first example below, both `abaaac` and `aabcba` are the longest chain of events. Thus it is alright to return any of them.

```
# Example 2.1 A single timeline
noccurence = 1

>>> mytrie.getLongestChain(noccurence)
abaaac
```

When we increase `noccurence` to 2, the longest chain is `dbce` where it is the longest continous chain, regardless of there being 2 timelines with the same events `dbce`.

```
# Example 2.2 Chain is continous
noccurence = 2

>>> mytrie.getLongestChain(noccurence)
dbce
```

```
# Example 2.3 Chain is continous
noccurence = 3

>>> mytrie.getLongestChain(noccurence)
bc
```

2.5 Hints

While heroes never need hints, Dr Weird could use some help...

- You are building a single trie with multiple event suffixes from multiple timelines. You might want to differentiate different timelines or track the timelines in such a way that the information helps you traverse down the trie more efficiently.
- You are looking for the longest chain of events, so what can you store or update during the construction of the trie to help guide your traversal?

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.

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