FIT2004 S1/2023: Assignment 2

DEADLINE: Friday 26^{th} May 2023 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 minutes late means 1 day late, 27 hours late is 2 days late.
For special consideration, please visit the following page and fill out the appropriate form:

- `https://forms.monash.edu/special-consideration` for Clayton students.
- `https://sors.monash.edu.my/` for Malaysian students.

The deadlines in this unit are strict, last minute submissions are at your own risk.

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, `assignment2.py`. Moodle will not accept submissions of other file types.

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 (or even zero marks for the unit as penalty) and, as a result, the large majority of those students failed the unit. Helping others to solve the assignment is NOT ACCEPTED. Please do not share your solutions partially or completely to others. Using contents from the Internet, books etc without citing is plagiarism (if you use such content as part of your solution and properly cite it, it is not plagiarism; but you wouldn't be getting any marks that are possibly assigned for that part of the task as it is not your own work).

The use of generative AI and similar tools is not allowed in this unit!

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. Try to resolve these questions by viewing the FAQ on Ed, or by thinking through the problems over time.
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.
- Remove print statements and test code from the file you are going to submit.

Documentation

For this assignment (and all assignments in this unit) you are required to document and comment your code appropriately. Whilst part of the marks of each question are for documentation, there is a baseline level of documentation you must have in order for your code to receive marks. In other words:

Insufficient documentation might result in you getting 0 for the entire question for which it is insufficient.

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.
- Your main function in the assignment should contain a generalised description of the approach your solution uses to solve the assignment task.
- 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 appropriate Big-O or Big-Θ 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):
    """
    Function description:

    Approach description (if main function):

    :Input:
        argv1:
        argv2:
    :Output, return or postcondition:
    :Time complexity:
    :Aux space complexity:
    """
    # Write your codes here.
```

There is a documentation guide available on Moodle in the Assignment section, which contains a demonstration of how to document code to the level required in the unit.

1 Fast Backups (10 marks)

You are the system administrator responsible for processing the backups of your company and the data needs to be backed up in different data centres. During certain times of the day you have total control over your company's data centres and their network connections, but you need to process the backup requests as fast as possible.

Your company has D data centres represented by $0, 1, \ldots, |D| - 1$. And you have a list `connections` of the direct communication channels between the data centres. `connections` is a list of tuples (a, b, t) where:

- $a \in \{0, 1, \ldots, |D|-1\}$ is the ID of the data centre from which the communication channel departs.
- $b \in \{0, 1, \ldots, |D| - 1\}$ is the ID of the data centre to which the communication channel arrives.
- t is a positive integer representing the maximum throughput of that channel.

Regarding `connections`:

- You cannot assume that the communication channels are bidirectional.
- You can assume that for each pair of data centers there will be at most one direct communication channel in each direction between them.
- You can assume that for every data centre $\{0, 1, \ldots, |D| - 1\}$ there is at least one communication channel departing or arriving at that data centre.
- You cannot assume that the list of tuples `connections` is given to you in any specific order.
- The number of communication channels $|C|$ might be significantly less than $|D|^2$, therefore you should not assume that $|C| = \Theta(|D|^2)$.

Moreover, each data centre has overall limits on the amount of incoming data it can receive per second, and also on the amount of outgoing data it can send per second. `maxIn` is a list of integers in which `maxIn[i]` specifies the maximum amount of incoming data that data centre i can process per second. The sum of the throughputs across all incoming communication channels to data centre i should not exceed `maxIn[i]`. Similarly, `maxOut` is a list of integers in which `maxOut[i]` specifies the maximum amount of outgoing data that data centre i can process per second. The sum of the throughputs across all outgoing communication channels from data centre i should not exceed `maxOut[i]`.

The backup request that you receive has the following format: it specifies the integer ID `origin` $\in \{0, 1, \ldots, |D| - 1\}$ of the data centre where the data to be backed up is located and a list `targets` of data centres that are deemed appropriate locations for the backup data to be stored. `targets` is a list of integers such that each integer i in it is such that $i \in \{0, 1, \ldots, |D| - 1\}$ and indicates that backing up data to server i is fine. Regarding those inputs:

- You can assume that `origin` is not contained in the list `targets`.
- You cannot assume that the list of integers `targets` is given to you in any specific order, but you can assume that it contains no duplicated integers.
- The data to be backed up can be arbitrarily split among the data centres specified in `targets` and each part of the data only needs to be stored in one of those data centres.

Your task is to determine the maximum possible data throughput from the data centre `origin` to the data centres specified in `targets`.

You should implement a function `maxThroughput(connections, maxIn, maxOut, origin, targets)` that returns the maximum possible data throughput from the data centre `origin` to the data centres specified in `targets`.

1.1 Complexity

Given an input with $|D|$ data centres and $|C|$ communication channels, your solution should have time complexity $O(|D| \cdot |C|^2)$

1.2 Example

Consider the example below:

```
# Example
connections = [(0, 1, 3000), (1, 2, 2000), (1, 3, 1000),
(0, 3, 2000), (3, 4, 2000), (3, 2, 1000)]
maxIn = [5000, 3000, 3000, 3000, 2000]
maxOut = [5000, 3000, 3000, 2500, 1500]
origin = 0
targets = [4, 2]

# Your function should return the maximum possible data throughput from the
# data centre origin to the data centres specified in targets.
>>> maxThroughput(connections, maxIn, maxOut, origin, targets)
4500
```

2 catGPT (10 marks)

Everyone is now talking about chatGPT. As amazing as it is, it is simply a well trained next word prediction model, similar to how the auto-complete on your phone but on a larger scale on the vast knowledge of the internet. In other words, chatGPT helps you finish your ~~sandwiches~~ sentences. You recall learning Tries from FIT2004 which can also be used for auto-complete. Thus, you set out to implement something similar using Tries.

Unfortunately, you do not have the computing power needed to process human language. Your friend, Kim Katdashian who is a cat expert suggest doing a variant for cats because it is simpler. She call this catGPT. The concept is simple – the model would be able to complete what does a cat say automatically:

1. Let say Tabby the cat goes "meow meoow meow meowth...".
2. Then catGPT will be able to complete Tabby's sentence with "... meow meuw nyan".

Kim states that there are a total of 26 words in a cat's vocabulary. You realized that it is possible to map this to the 26 letters in the English language. For example:

- meow = a
- meoow = b
- meuw = c
- meuuw = d
- meuow = e
- ...
- nyan = y
- meowth = z

In the example above, Tabby's initial sentence could be viewed as `abaz` and then it is auto-completed to be `acy`. It is auto-completed this way because many many cats would finish their sentence in such a way [1]. Luckily, Kim has collected a large set of cat `sentences` to help your Trie decide how a cat sentence would be completed. These sentences are stored in a long list of sentences, formatted as follows:

- `abc`
- `abazacy`
- `dbcef`
- `gdbc`
- `abazacy`
- `xyz`
- ...
- `abazacy`
- `dbcef`

[1] Do note ask us why, we are not cute cudly cats.

2.1 The Trie Data Structure

You must write a class `CatsTrie` that encapsulates all of the cat sentences. The `__init__` method of `CatsTrie` would take as input a list of timelines, `sentences`, represented as a list of strings with:

- N sentences, where N is a positive integer.
- The longest sentence would have M characters, as mapped from the cat vocabulary. M is a positive integer.
- A cat word can occur more than once in a single sentence. For example, the string `baacbb` represents a valid sentence.
- It is possible for more than a single sentence to have the same words. Have a look at the example in Section 2.4.
- You can assume that there is only a maximum of 26 unique cat words in total, represented as lower case characters from **a** to **z**.

```
# The CatsTrie class structure
class CatsTrie:
    def __init__(self, sentences):
        # ToDo: Initialize the trie data structure here
    def autoComplete(self, prompt):
        # ToDo: Performs the operation needed to complete the prompt,
        based on the most commonly used sentences.
```

For the example stated earlier, the `CatsTrie` object will contain all the following sentences `abc`, `abazacy`, `dbcef`, `gdbc`, `abazacy`, `xyz`, ..., `abazacy`, `dbcef` .

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

2.2 Auto-Completing Sentences

You would now proceed to implement `autoComplete(self, prompt)` as a function within the `CatsTrie` class. The function accepts 1 argument:

- `prompt` is a string with characters in the set of `[a...z]`. This string represents the incomplete sentence that is to be completed by the trie.

The function would then return a string that represents the completed sentence from the prompt. Do note the following:

- If such a sentence exist, return the completed sentence with the highest frequency in the cat `sentences` list.
- If there are multiple possible auto-complete sentences with the same highest frequency, then you would return the lexicographically smaller string.
- If such a sentence does not exist, return `None`

2.3 Complexity

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

The `__init__(sentences)` constructor of `CatsTrie` class would run in $O(NM)$ time and space where:

- N is the number of sentence in `sentences`.
- M is the number of characters in the longest sentence.

The `autoComplete(self, prompt)` of the `CatsTrie` class would run in $O(X+Y)$ time where:

- X is the length of the `prompt`.
- Y is the length of the most frequent sentence in `sentences` that begins with the `prompt`, unless such sentence do not exist (in which case `autoComplete(self, prompt)` should have complexity $O(X)$). Thus, this is an output-sensitive complexity.

2.4 Example

Consider the following example.

```
# Example 1, similar to the introduction
# but with some additional sentences

sentences = ["abc", "abazacy", "dbcef", "xzz", "gdbc", "abazacy", "xyz",
    "abazacy", "dbcef", "xyz", "xxx", "xzz"]

# Creating a CatsTrie object
mycattrie = CatsTrie(sentences)
```

Running `autoComplete(self, prompt)` would return different results depending on the `prompt` value. We provide a few examples below, all of which can be run any number of times after the `mycattrie` object is created once:

```
# Example 1.1
# A simple example
prompt = "ab"

>>> mycattrie.autoComplete(prompt)
abazacy
```

```
# Example 1.2
# Another simple example
prompt = "a"

>>> mycattrie.autoComplete(prompt)
abazacy
```

```
# Example 1.3
# What if the prompt is the same as an existing sentence?
prompt = "dbcef"

>>> mycattrie.autoComplete(prompt)
dbcef
```

```
# Example 1.4
# What if the length is longer?
prompt = "dbcefz"

>>> mycattrie.autoComplete(prompt)
None
```

```
# Example 1.5
# What if sentences doesn't exist.
prompt = "ba"

>>> mycattrie.autoComplete(prompt)
None
```

```
# Example 1.6
# A scenario where the tiebreaker is used
prompt = "x"

>>> mycattrie.autoComplete(prompt)
xyz
```

```
# Example 1.7
# A scenario where the prompt is empty
prompt = ""

>>> mycattrie.autoComplete(prompt)
abazacy
```

```
# Example 2, similar to the Example 1
# but with some minor channges

sentences = ["abc", "abczacy", "dbcef", "xzz", "gdbc", "abczacy", "xyz",
    "abczacy", "dbcef", "xyz", "xxx", "xzz"]

# Creating a CatsTrie object
mycattrie = CatsTrie(sentences)

# Example 2.1
# A scenario where the prompt is the same as an existing sentence
# but it can be completed with higher frequency
prompt = "abc"

>>> mycattrie.autoComplete(prompt)
abczacy
```

Do note that these are just a few of the many possible scenarios and cases associated with this question. catGPT isn't that smart, and thus you would need to carefully think about how catGPT would behave for a wide range of prompts.

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!**