FIT2004 S2/2022: Assignment 2

DEADLINE: Friday 16^{th} September 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 minutes 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. 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`.

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. Part of the marks of each question are for documentation. 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 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):
    """
    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 Gotta Go Fast
(10 marks, including 2 marks for documentation)

Caffeine runs through your veins and you can't function without it. No matter where you go, you would need to grab a cup of coffee[1] on the way to the destination. You, however, often find yourself late to your destination due to this; especially when the cafes have a long waiting time. Thus, you vow to never be late again – for a computer scientist is never late nor early, one arrives precisely when one means to... cause one is efficient using Graph algorithms of course!

With that, you have downloaded the travel time between key locations in your city which you can use to estimate the travel time from a point to another. You have also time the waiting time of each cafe that you would grab your coffee from.

Given your FIT2004 study, you have come to the realisation that it is possible to model your travels using the `RoadGraph` data structure:

- The `RoadGraph` class implementation is as described in Section 1.1. An example of the graph structure is provided below.
- You can then calculate the optimal routes for your commute while grabbing coffee along the way using a function in the `RoadGraph` class called `routing(self, start, end)` detailed in Section 1.2.

```
class RoadGraph:
    def __init__(self, roads, cafes):
        # ToDo: Initialize the graph data structure here
    def routing(self, start, end):
        # ToDo: Performs the operation needed to find the optimal route.
```

[1] Or tea if you prefer.

1.1 Graph Data Structure (3 mark)

You must write a class `RoadGraph` that represents the road network in the city.

The `__init__` method of `RoadGraph` would take as an input a list of roads `roads` represented as a list of tuples (u, v, w) where:

- u is the starting location ID for a road, represented as a non-negative integer.
- v is the ending location ID for a road, represented as a non-negative integer.
- w is the time taken to travel from location u to location v, represented as a non-negative integer.
- You cannot assume that the list of tuples are in any specific order.
- You cannot assume that the roads are 2-way roads.
- You can assume that the location IDs are continuous from 0 to $|V| - 1$ where $|V|$ is the total number of locations.
- The number of roads $|E|$ can be significantly less than $|V|^2$, therefore you should not assume that $|E| = \Theta(|V|^2)$.

The `__init__` method of `RoadGraph` also takes as an input a list of cafes `cafes` represented as a list of tuples (`location`, `waiting_time`) where:

- `location` is the location of the cafe; represented as a non-negative integer.
- `waiting_time` is the waiting time for a coffee in the cafe, represented as a non-negative integer.
- You cannot assume that the list of tuples are in any specific order.
- You can assume that all of the `location` values are from the set {0, 1, . . . , |V|-1}.
- You can assume that all of the `location` values are unique.
- You cannot assume `waiting_time` to be within any range except that it is a value > 0.

Consider the following example in which the roads and cafes of a city are stored as lists of tuples:

```
# The roads represented as a list of tuples
roads = [(0, 1, 4), (0, 3, 2), (0, 2, 3), (2, 3, 2), (3, 0, 3)]
# The cafes represented as a list of tuple
cafes = [(0, 5), (3, 2), (1, 3)]
```

We have that:

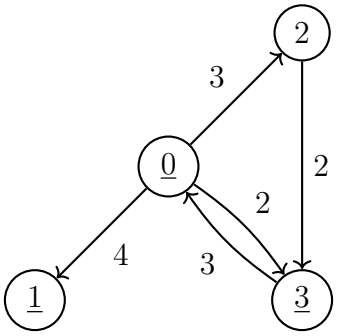
- There are 4 locations in the city with ID from 0 to 3.
- There are 5 roads in total in the city connecting the locations.
- There is a road from location 0 to location 1 that will have a travel time of 4 minutes.
- There is a road from location 0 to location 3 that will have a travel time of 2 minutes.
- There is a road from location 0 to location 2 that will have a travel time of 3 minutes.
- There is a road from location 2 to location 3 that will have a travel time of 2 minutes.
- There is a road from location 3 to location 0 that will have a travel time of 3 minutes.
- Since you can't assume the roads are 2-way roads, we observe only location 0 and location 3 are connected through a 2-way road although there is a difference in the travel time.

Regarding the cafes:

- There are 3 cafes in the city, at the locations with IDs 0, 1 and 3.
- The cafe at location 0 has a waiting time of 5 minutes.
- The cafe at location 1 has a waiting time of 3 minutes.
- The cafe at location 3 has a waiting time of 2 minutes

Running the following code would create a `RoadGraph` object called `mygraph`, which is then visualised below with the cafes location underlined:

```
# Creating a RoadGraph object based on the given roads and cafes
mygraph = RoadGraph(roads, cafes)
```



Based on the information above, you would need to implement the `RoadGraph` class using either an adjacency matrix or adjacency list representation. Do note that one implementation is less efficient than the other in both time and space complexity. Thus, consider your implementation carefully in order to achieve full marks.

1.2 Optimal Route Function (5 marks)

You would now proceed to implement `routing(self, start, end)` as a function within the `RoadGraph` class. The functions accepts 2 arguments:

- `start` is a non-negative integer that represents the starting location of your journey. Your route must begin from this location.
- `end` is a non-negative integer that represents the ending location of your journey. Your route must end in this location.
- Do note that it is possible for the locations in `cafes` to include the `start` and/or `end` as well.

The function would then return the shortest route from the `start` location to the `end` location, going through at least 1 of the locations listed in `cafes`.

- If such route exist, it would return the shortest route as a list of integers. If there are multiple shortest routes satisfying the constraint of going through at least 1 of the locations listed in `cafes`, return any one of those shortest routes.
- If no such route going from the `start` location to one of the locations in `cafes` and proceeding next to the `end` location is possible, then the function would return `None`.

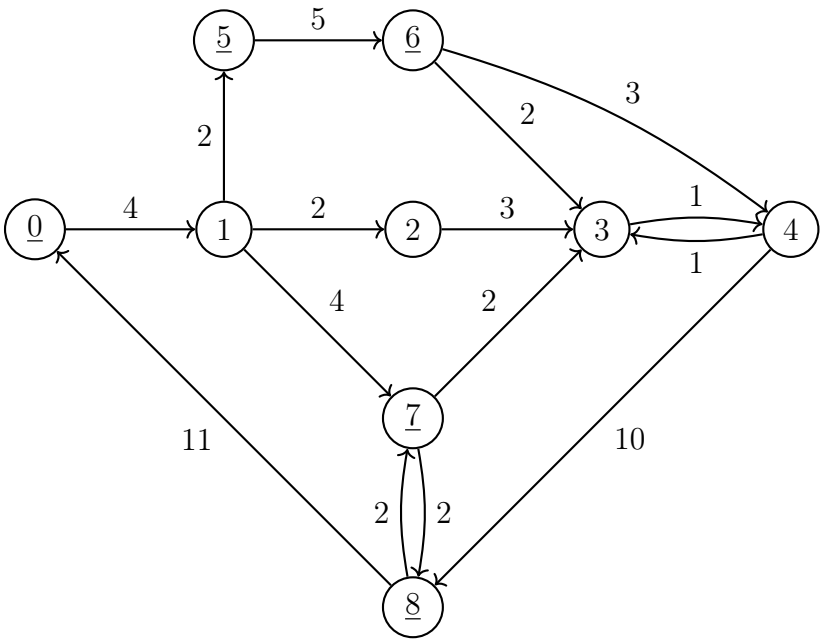
Several examples are provided in Section 1.3.

1.3 Examples

Consider the example road network below:

```
# Example 1
# The roads represented as a list of tuple
roads = [(0, 1, 4), (1, 2, 2), (2, 3, 3), (3, 4, 1), (1, 5, 2),
(5, 6, 5), (6, 3, 2), (6, 4, 3), (1, 7, 4), (7, 8, 2),
(8, 7, 2), (7, 3, 2), (8, 0, 11), (4, 3, 1), (4, 8, 10)]
# The cafes represented as a list of tuple
cafes = [(5, 10), (6, 1), (7, 5), (0, 3), (8, 4)]

# Creating a RoadGraph object based on the given roads
mygraph = RoadGraph(roads, cafes)
```



Running the following functions will yield:

```
# Example 1.1
start = 1
end = 7

>>> mygraph.routing(start, end)
[1, 7]
```

The simple Example 1.1 above is just going from location 1 to location 7, grabbing coffee at location 7, adding 5 minutes of waiting time. Thus the total trip takes 9 minutes total.

```
# Example 1.2
start = 7
end = 8

>>> mygraph.routing(start, end)
[7, 8]
```

On the other hand in Example 1.2, going from location 7 to location 8, we would grab coffee at location 8 because we would only need to wait for 4 minutes there instead of 5 minutes at location 7. Thus the total time taken is 6 minutes. This example also highlight how it is possible for the `start` and/or `end` to contain a cafe.

```
# Example 1.3
start = 1
end = 3

>>> mygraph.routing(start, end)
[1, 5, 6, 3]
```

In Example 1.3, there are multiple possible routes that pass through a cafe. One of them is $[1, 5, 6, 3]$ with a total travel time of 10 minutes. Another is $[1, 7, 3]$ with a total travel time of 11 minutes. There are other routes but those would be slower routes, especially the ones with cycles. The returned value would be $[1, 5, 6, 3]$, as it is the quickest route.

```
# Example 1.4
start = 1
end = 4

>>> mygraph.routing(start, end)
[1, 5, 6, 4]
```

In Example 1.4 there are 2 shortest route with the same travel time of 11 – $[1, 5, 6, 3, 4]$ and $[1, 5, 6, 4]$. For such scenario, you can return any of the 2 routes.

```
# Example 1.5
start = 3
end = 4

>>> mygraph.routing(start, end)
[3, 4, 8, 7, 3, 4]
```

In Example 1.5 above, we could reach location 4 from location 3 but unfortunately we would need to take a detour in order to grab coffee through location 8, adding 4 minutes of waiting time. This showcase an example where a cafe location is after the `end` location.

There are many more possible scenarios that are not covered in the examples above. It is a requirement for you to identify any possible boundary cases.

1.4 Complexity

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

The `__init__(roads, cafes)` constructor of `RoadGraph` class would run in $O(|V| + |E|)$ time and space where:

- V is the set of unique locations in `roads`. You can assume that all locations are connected by roads (i.e a connected graph); and the location IDs are continuous from 0 to $|V| - 1$.
- E is the set `roads`.
- The number of roads $|E|$ can be significantly smaller than $|V|^2$. Thus, you should not make the assumption that $|E| = \Theta(|V|^2)$.
- Note that the `cafes` is not stated in the complexity. At worst, the size of `cafes` is $|V|$.

The `routing(self, start, end)` of the `RoadGraph` class would run in $O(|E| \log |V|)$ time and $O(|V| + |E|)$ auxiliary space.

2 Maximum Score (10 marks)

You are an avid snowboarder and are doing your best efforts to prepare for the upcoming local tournament.

You made several visits to the resort where the tournament will take place and cautiously studied the trails and practised on them in oder to determine the score that you can obtain in each segment. With your current skill level you are able to move downhill, but you are not able to move properly neither uphill nor on flat surfaces. Therefore, during the tournament you will only go through downhill segments (i.e., the start point of the segment is higher than the end point).

Given your extensive preparation efforts, you are able to determine with perfect precision the score you would be able to get on each downhill segment if you decide to go through it. Now, given the start and end points of the tournament, you want to maximise your score by choosing the best combination of downhill segments to go from the start point to the end point so that the sum of your scores in the used segments is maximised.

Each segment starts and finishes at an intersection point. There are $|P|$ intersection points and they are denoted by $0, 1, \ldots, |P| - 1$. There are $|D|$ downhill segments. You can assume that for each intersection point there is at least one downhill segment that starts or finishes at that intersection point.

From your preparation, you learned `downhillScores`, which is represented as a list of $|D|$ tuples (a, b, c):

- a is the start point of a downhill segment, $a \in \{0, 1, \ldots, |P| - 1\}$.
- b is the end point of a downhill segment, $b \in \{0, 1, \ldots, |P| - 1\}$.
- c is the integer score that you would get for using this downhill segment to go from point a to point b.
- You cannot assume that the list of tuples are in any specific order.

Now, the tournament organisers have decided the starting point `start` and finishing point `finish` of the tournament. We have that $\texttt{start}, \texttt{finish} \in \{0, 1, \ldots, |P| - 1\}$.

Your current job is to implement a function `optimalRoute(downhillScores, start, finish)` that outputs the route that you should use for going from the starting point `start` to the finishing point `finish` while using only downhill segments and obtaining the maximum score:

- If no such route going from the starting point `start` to finishing point `finish` while using only downhill segments exists, then the function would return `None`.
- Otherwise, it would return the optimal route as a list of integers. If there are multiple optimal routes, return any of them.

2.1 Example

Consider the example below:

```
# Example
# The scores you can obtain in each downhill segment
downhillScores = [(0, 6, -500), (1, 4, 100), (1, 2, 300),
(6, 3, -100), (6, 1, 200), (3, 4, 400), (3, 1, 400),
(5, 6, 700), (5, 1, 1000), (4, 2, 100)]

# The starting and finishing points
start = 6
finish = 2

>>> optimalRoute(downhillScores, start, finish)
[6, 3, 1, 2]
```

In this example, your optimal route is to start at point 6, proceed to point 3, then point 1 and finally head to point 2 to finish. And that would give you a total score of 600, which is the maximum you can achieve on any downhill route from point 6 to point 2.

2.2 Complexity

In order to be able to get full marks for this question, your solution should have time complexity $O(|D|)$, where $|D|$ is the number of downhill segments.

If the time complexity of your solution is $O(|D| \cdot |P|)$ - where |P| is the number of intersection points - but not $O(|D|)$, then you will get at most 8 out of 10 marks for this question. Any solution with time complexity asymptotically worse than that will incur more significant lost of marks.

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!**