# FIT2004 2024 Semester 2: Assignment 1

**DEADLINE:** Friday $6^{th}$ September 2024 23:55:00 AEST.

**LATE SUBMISSION PENALTY:** 5% penalty per day. Submissions more than 7 calendar days late will receive 0. The number of days late is rounded up, e.g. 5 seconds 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`.

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 Pythown file containing all of the questions you have answered, `assignment1.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. Even after the deadline, your solutions/approaches should not be shared before the grades and feedback are released by the teaching team. 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 for the completion of your assignment is not allowed in this unit! Previous students who neglected this were severely penalized.**

# Learning Outcomes

This assignment achieves the Learning Outcomes of:

- Analyse general problem solving strategies and algorithmic paradigms, and apply them to solving new problems;
- Prove correctness of programs, analyse their space and time complexities;
- Compare and contrast various abstract data types and use them appropriately;
- 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-$\Theta$ 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:
   :Time complexity analysis:
    :Space complexity:
   :Space complexity analysis:
    """
    # 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 Two-Gather
# (10 marks, including 2 marks for documentation)

You and your friends have just recently moved to a new city in order to further your studies. Before coming over, you all made a pact – to never do anything alone. Thus wherever you go, you should always go with someone. Unfortunately due to budget constraints, you are all living separated all over the city. Besides that, none of your friends own a vehicle, so they need to commute by train.

Committed to not breaking the pact, you offer to pick up a friend wherever you go. However, you still want to arrive at your destination on time. Thankfully, you are a trained computer scientist and can use the power of Graphs to plan your travel route efficiently.

You have obtained a map of the city's `roads` in order to plan your route better:

- There are $|L|$ locations in the city, from $\ell_0$ to $\ell_{|L|-1}$.
- There are a total of $|R|$ roads shown in the map, from $r_0$ to $r_{|R|-1}$. Each road is from one location to another.
- It is possible to get from any location to any other location by driving along some number of roads.
- You can drive from location $\ell_u$ to location $\ell_v$, if a road exist between those locations. All roads in the city are two-way.
- For each road you know how many minutes it takes to travel along that road. The travel time differs among roads.

While it is possible for you to pick up your `friends` from their homes, it is not always efficient to do so. Instead, your friends might need to use the rail to meet you at a more convenient location for pickup. As you care for your friends, the pickup locations would be within two station stops from where they live. To help you plan the route, you also obtained the `tracks` map for the train service.

- There are a total of $|T|$ tracks shown in the map, from $t_0$ to $t_{|T|-1}$. These tracks connect locations in the city, although not all locations are necessarily connected by rail network.
- You can take a train from location $\ell_u$ to location $\ell_v$, if a track exists from location $\ell_u$ to location $\ell_v$. Unlike roads however, these tracks are one-way only.
- For each track you know how many minutes it takes to travel along that track. Different tracks can take different amounts of time to travel.

Using both maps, you would model the transportation network as a graph ADT and use it to identify the optimal route for your travels.

```
class CityMap:
    def __init__(self, roads, tracks, friends):
        # ToDo: Initialize the graph data structure here.
                To be explained in Section 1.1.
    def plan(self, start, destination):
        # ToDo: Performs the operation needed to find the optimal route.
                To be explained in Section 1.2.
```

## 1.1 Graph Data Structure

You must write a class `CityMap` that represents the transportation network of the city.
The `__init__` method of the `CityMap` would take as an input a list of `roads` represented as a list of tuples $(u, v, m)$ where:

- $u$ is the starting location for the road; and $v$ is the ending location for the road. These are non-negative integers in range of $0, \ldots, |L| - 1$.
- $m$ is the amount of time needed to travel down the road from location-$u$ to location-$v$. This is a positive integer.
- You cannot assume that the list of tuples is in any specific order.
- You can assume that the roads are 2-way roads.
- You can assume that there is a path of roads to travel from any location to any other location in the city.
- The total number of roads $|R|$ can be significantly smaller than $|L|^2$ and therefore you should not assume that $|R| = \Theta(|L|^2)$.

The `__init__` method of the `CityMap` also takes as an input a list of `tracks` represented as a list of tuples $(u, v, m)$ where:

- $u$ is the starting location for the train track; and $v$ is the ending location for the train track. These are non-negative integers in range of $0, \ldots, |L| - 1$.
- $m$ is the amount of time needed for the train to travel through the track from location-$u$ to location-$v$. This is a positive integer.
- You cannot assume that the list of tuples is in any specific order.
- You can assume that the tracks are 1-way tracks. You can also assume that there would not be multiple tracks connecting 2 locations in the city in the same direction.
- You cannot assume that all locations are connected to some track. Therefore, the total number of tracks $|T|$ can be anything from 0 to $|L|^2$.

Lastly, `__init__` method of the `CityMap` also takes as an input a list of `friends` represented as a list of tuples $(x, y)$ where:

- $x$ is the name of your friend. This is a string. This value is unique.
- $y$ is the location where your friend lives. This is a non-negative integer in the range of $0, \ldots, |L| - 1$.
- You can assume that the total number of friends $|F|$ is such that $|F| \leq |L|$.

Consider the following example:

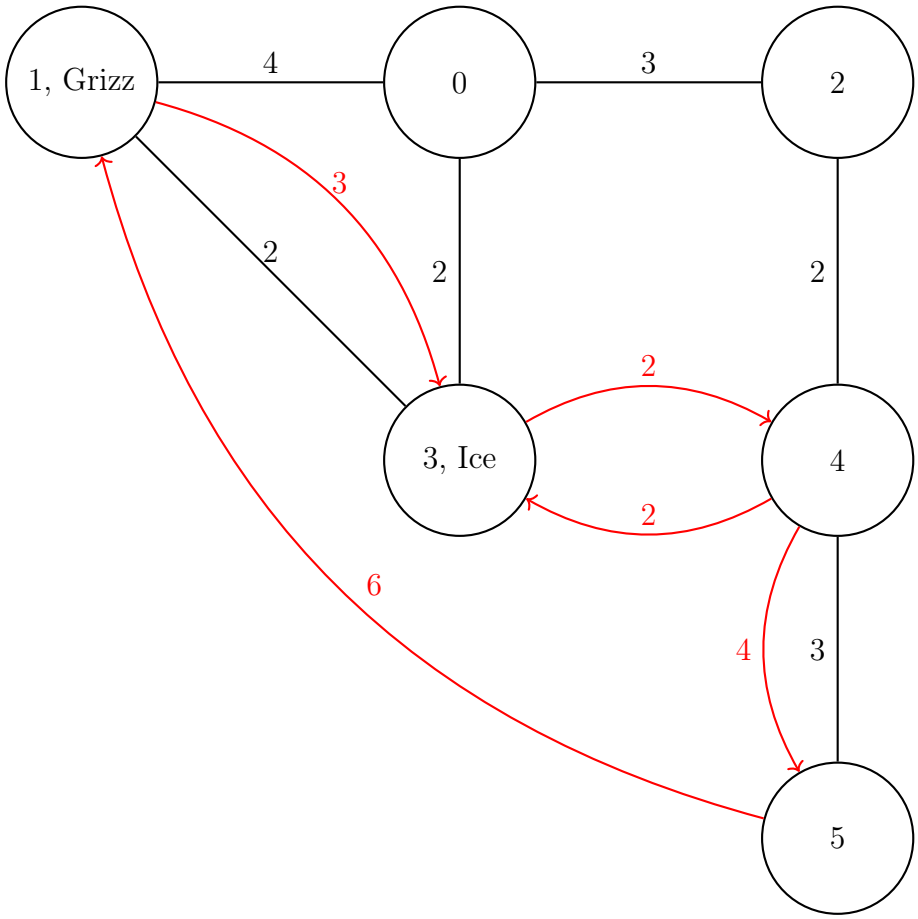
```
# The roads represented as a list of tuples
roads = [(0,1,4), (0,3,2), (2,0,3), (3,1,2), (2,4,2), (4,5,3)]
# The tracks represented as a list of tuples
tracks = [(1,3,3), (3,4,2), (4,3,2), (4,5,4), (5,1,6)]
# The friends represented as a list of tuples
friends = [("Grizz", 1), ("Ice", 3)]
```

There are a total of 6 `roads`, connecting a total of 6 locations from ID 0 to 5 in the city. The first tuple $(0, 1, 4)$ can be read as – there is a road between location-0 and location-1, where traversing this road takes 4 minutes.

Likewise, there are a total of 5 `tracks`, for the train service which connects location within the city. The first tuple $(1, 3, 3)$ can be read as – there is a train service from location-1 to location-3 where the train will take 3 minutes to traverse this track.

Lastly, you have 2 `friends`. The first friend, Grizz lives in location-1 and another friend Ice who lives in location-3. The resulting graph can be visualized as shown below.

```
# Creating a CityMap object based on the given input
myCity = CityMap(roads, tracks, friends)
```




From the illustration, we can observe the complexity of the city's transportation network. Some of the observations to be made include:

- There is always a road path to go from any location in the city to any other location.
- There is no guarantee that the entire city is connected by the train system, thus there could be no tracks connecting certain locations such as between location-0 and location-2; or a track to connect location-3 to location-1.
- You can only pick up friend Grizz from either location-1, location-3, or location-4 given how the train system of the city is setup.
- You can only pick up friend Ice from either location-3, location-4, or location-5 given how the train system of the city is setup.

Of course, the design and implementation of the `CityMap` class, and any other additional classes[1] is left to you. You are free to process the input and make any required adjustments that you deemed appropriate in order to prepare a suitable data structure that can be used to solve the problem efficiently.

[1] If you think additional classes are required.

## 1.2 Optimal Route Planning

You would now proceed to implement `plan(self, start, destination)` as a function within the `CityMap` class. This function accepts two arguments:

- `start` is a non-negative integer that represents a location in the city. You begin your journey from here.
- `destination` is a non-negative integer that represents a location in the city. You end your journey here.

A valid route will get you from location-`start` to location-`destination` given in the function arguments, with the additional requirement of picking up a friend along the route. And the pickup location should be within two station stops from where your friend lives. Don't worry, your friend will always be on time and be at the pickup point when you arrive. Among the set of valid routes you should satisfy the following constraints:

- **Constraint 1:** If there are multiple valid routes, you should choose one of the valid routes that minimises your travel time.
- **Constraint 2:** If there are multiple valid routes satisfying Constraint 1, you should choose a combination of route, friend and pickup location that minimises the number of train stations that your friend will have to travel to get to the pickup location.
- If there are still multiple valid routes satisfying Constraints 1 and 2, you can choose any one of them. In other words, the optimal route is not necessarily unique.

Consider the optimal valid route along with the optimal friend and pickup location according to the above constraints. Your function `plan(self, start, destination)` should return a tuple `(total_time, route, pickup_friend, pickup_location)` where:

- `total_time` is your travel time along the optimal valid route. This should be a non-negative integer.
- `route` is the route that you use to go in time equal to `total_time` from `start` to `destination` while picking up one of your `friends` along the way. This should be a list of integers, that represent the locations in the city that you would travel along the route.
- `pickup_friend` is the name of the friend who you would be picking up.
- `pickup_location` is the location where you would pick up your friend along the route. This would be a non-negative integer, where the value would be in the range from 0 to $|L| - 1$.

To better understand the behaviour of the function, you can refer to the examples provided in Section 1.3. It should be noted that the function `plan(self, start, destination)` can be called multiple times, on the same `CityMap` object and thus your implementation should be able to handle this.

## 1.3 Examples

In this section, we provide several examples for you to gain a better understanding of the intended behaviour and outcome for the `plan(self, start, destination)` function. It is recommended for you to visualize out these examples.

We first start with several examples using the same inputs as the visualized graph in Section 1.1.

```
# Input values similar to Section 1.1
roads = [(0,1,4), (0,3,2), (2,0,3), (3,1,2), (2,4,2), (4,5,3)]
tracks = [(1,3,3), (3,4,2), (4,3,2), (4,5,4), (5,1,6)]
friends = [("Grizz", 1), ("Ice", 3)]
# Create the CityMap
myCity = CityMap(roads, tracks, friends)
```

As a very simple example below consider the case you want to travel from location-2 to location-5. You would travel along the path of location-2 to location-4, picking up your friend Ice there. This would require Ice to take a train from location-3 to location-4. After that, both of you would travel to location-5. It is possible for your friend Grizz to take 2 trains over to location-4 instead, but this is less efficient as it would require Grizz to take 2 trains as opposed to Ice only taking 1 train.

```
# Example 1.1, simple example
>>> myCity.plan(start=2, destination=5)
(5, [2,4,5], "Ice", 4)
```

In the next example below, it is possible for you to meet your friend to be at your destination itself. This can be caused by the friend already being at the destination location, or it is more efficient for you to meet your friend there even if your friend would need to board the train [2].

```
# Example 1.2, meeting the friend at destination
>>> myCity.plan(start=0, destination=4)
(5, [0,2,4], "Ice", 4)
```

For the example below, we have 2 possible solutions with a total time of 7 minutes, and both picking up your friend at their current location – Grizz at location-1 or Ice at location-3. Thus in this example, it is a total tie. The same would apply if both routes would require your friends to take the same number of trains.

```
# Example 1.3, possible solution 1
>>> myCity.plan(start=2, destination=1)
(7, [2,0,1], "Grizz", 1)
```

```
# Example 1.3, possible solution 2
>>> myCity.plan(start=2, destination=1)
(7, [2,0,3,1], "Ice", 3)
```

[2]Meeting your friend at the destination is similar to that of picking up your friend at the destination itself.

There can also be a scenario where you pass your destination, on the way to pick up your friend Ice as with the example below. You wanted to go to location-0 but as your friend is not along the way or there are no trains there, you have no choice but head to location-3 first and then turn back to your destination.

```
# Example 1.4, going beyond the destination
>>> myCity.plan(start=2, destination=0)
(7, [2,0,3,0], "Ice", 3)
```

Consider the example below, with a different set of inputs. While your friend Ice would be able to reach location-5 using train from location-3 in 3 minutes, he would need to take 2 trains. It is cheaper for him to take a single train to location-4 for your pickup instead, before heading to location-5 directly.

```
# Example 2.1, the less train, the better

roads = [(0,1,4), (0,3,2), (2,0,3), (3,1,2), (2,4,2), (4,5,3)]
tracks = [(1,3,4), (3,4,2), (4,3,2), (4,5,1), (5,1,6)]
friends = [("Grizz", 1), ("Ice", 3)]
myCity = CityMap(roads, tracks, friends)

>>> myCity.plan(start=2, destination=5)
(5, [2,4,5], "Ice", 4)
```

Below is the last example given to you for a more specific scenario. In this scenario, there are train tracks that would connect your friend Grizz to go from the current location-1 to location-5. This would allow you to go directly to the destination in location-5 using only 2 minutes, meeting your friend Grizz in locaction-5. However, this would required 3 trains which is above the maximum of 2. Thus, you have no choice but to take a detour going through location-4 in order to pick up Grizz.

```
# Example 2.2, a more complex scenario

roads = [(0,1,4), (0,3,2), (2,0,3), (3,1,2), (2,4,2), (2,5,2)]
tracks = [(1,3,4), (3,4,2), (4,5,1), (5,1,6)]
friends = [("Grizz", 1)]
myCity = CityMap(roads, tracks, friends)

>>> myCity.plan(start=2, destination=5)
(6, [2,4,2,5], "Grizz", 4)
```

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 to ensure that your solution would be able to handle all cases correctly.

## 1.4 Complexity

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

The `__init__(roads, tracks, friends)` constructor of the `CityMap` class should have time complexity $O(|R| + |T|)$ and auxiliary space complexity $O(|R| + |T|)$ where:

- $R$ is the set `roads` and $T$ is the set of `tracks`.
- The total number of roads $|R|$ can be significantly smaller than $|L|^2$, where $L$ is the set of locations. Thus, you should not make the assumption that $|R| = \Theta(|L|^2)$. The same can be said for the number of tracks $|T|$.
- **Just to note:** The auxiliary space complexity here is loosen following discussion with the teaching team to provide some leeway, albeit it can be done in the original $O(|R|)$ requirement. However, needless inefficiency will still be penalized.

The function `plan(self, start, destination)` of the `CityMap` class would run in $O(|R| \log |L|)$ time and $O(|R|)$ auxiliary space. This would run with the same complexity for any combination of `start` and `destination`. Note that neither the number of `tracks` nor of `friends` are stated in the complexity for this function, this is intentional!

# 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!**