FIT2004 S2/2023: Assignment 1

DEADLINE: Friday 15^{th} September 2023 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 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 python 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. 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 Question 1: Fast Food Chain (10 marks, including 2 marks for documentation)

Burger Queen, a popular chain of fast food restaurants, is planning to open its restaurants along a newly built freeway. They have identified N potential sites along the freeway such that the distance between every two consecutive sites is exactly 1 km. They have also estimated the expected annual revenue generated by each site if a restaurant is opened at the site. Due to different demographics and locations, each site may generate a different annual revenue. It is company policy that no two restaurants can be within d kms of each other. They have hired you to help them choose the sites to open restaurants such that no two restaurants are within d kms of each other and the overall revenue is maximised. The company has an unlimited budget and they are happy to open any number of restaurants as long as no two restaurants are within d kms of each other.

You need to write a Python function named `restaurantFinder(d,site_list)` which takes two arguments: `d` is the distance parameter d; and `site_list` is a list of size N containing annual revenue (in million dollars) for each site, i.e., i^{th} number in the list is r_i denoting the annual revenue of the site i if a restaurant is opened at the site. `site_list` contains the sites in the order they appear along the freeway, e.g., for each i, the distance between i^{th} and $(i+x)^{th}$ site is x kms. You can assume that d is always a non-negative integer and `site_list` always contains at least 1 site.

The function must return a tuple `(total_revenue, selected_sites)` where `selected_sites` is a list containing the site numbers where the company should open their restaurants to maximise the revenue. The list `selected_sites` must contain the site numbers in ascending order. `total_revenue` is the total annual revenue if the company opens restaurants at the sites in `selected_sites`.

Your function must solve the problem using $O(N)$ space and $O(N)$ time in the worst-case.

1.1 Example

Below is a sample input.

```
>>> restaurantFinder(1,[50, 10, 12, 65, 40, 95, 100, 12, 20, 30])
```

The above input shows that the value of d is 1 and there are 10 sites in total. Annual revenue of the first site is 50, for the second site is 10 and so on.
Below are some sample outputs for different values of d but the same list of sites..

```
>>> restaurantFinder(1,[50, 10, 12, 65, 40, 95, 100, 12, 20, 30])
(252,[1,4,6,8,10])
```

```
>>> restaurantFinder(2,[50, 10, 12, 65, 40, 95, 100, 12, 20, 30])
(245,[1,4,7,10])
```

```
>>> restaurantFinder(3,[50, 10, 12, 65, 40, 95, 100, 12, 20, 30])
(175,[1,6,10])
```

```
>>> restaurantFinder(7,[50, 10, 12, 65, 40, 95, 100, 12, 20, 30])
(100,[7])
```

```
>>> restaurantFinder(0,[50, 10, 12, 65, 40, 95, 100, 12, 20, 30])
(434,[1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

If there are more than one possible answers (e.g., multiple ways to achieve the maximum revenue), you can return any of the answers.

2 Question 2: Climb King (10 marks, including 2 marks for documentation)

One day, a tower appeared out of nowhere with the message – "Reach the top of the tower and your wish will be granted". Thus, everyone flocked to the entrance of the tower hoping to have their wish granted. You are one such adventurer, hoping to see the night sky at the top of the tower.

To your surprise, there is a map that shows the layout of floors of the tower. For example, the first floor map shows that:

- There are $|V|$ `locations` in the map.
- The `entrance` is shown in the map. This is where you enter the floor, there is only one entrance to the floor.
- There can be 1 or more `exits` shown in the map. This is where you will always exit to the next floor.

- There are $|E|$ edges that connect the locations of a floor together.
- You can go from location-u to location-v, if a path $e = (u, v)$ exists. However, you can not go from location-v to location-u unless the opposite path $e' = (v, u)$ exist as well in the map.
- It would take x-minutes to traverse that specific path $e = (u, v, x)$. The time to traverse each path could vary, and would be stated in the map itself.

Thus, you would want to use the map to reach an exit as fast as possible. However, you noticed strange markers in the map:

- There are $|K|$ `keys` positioned at certain locations in the map.
- A key is needed in order to move on to the next floor. Without a key, you are powerless at the any of the `exits` location. You can exit the floor using any key.
- Each key is however defended by a monster. It would take y-minutes to defeat the monster a retrieve the key. This vary depending on keys, but you would always need to defeat a monster! Eventually... The map automatically calculates this based on your current capabilities. In other words, you would be given the time it would take you to defeat a monster defending a key.
- If you are at the same location as that of a key, you can choose not to engage with the monster and continue moving along path. In other words, if you choose to fight the monster, you will be able to collect the key but will spend y minutes defeating the monster and getting the key. If you choose not to fight the monster, you do not need to spend any time but you will leave the location without getting the key.

The `start` and `exits` changes every time you enter the floor. However, the map remains unchanged. Thus, this causes a challenge for the adventurers. Unlike them, you are a SSS-ranked computer scientist, with the power of algorithms and data structures – to climb the tower as fast as possible, computing the fastest path to go from the `start` to one of the `exits`. You would model the floor map using the graph ADT as follow:

```
class FloorGraph:
    def __init__(self, paths, keys):
        # ToDo: Initialize the graph data structure here.
                More details to be described in Section 2.1
    def climb(self, start, exits):
        # ToDo: Performs the operation needed to find the optimal route.
                More details to be described in Section 2.2
```

2.1 Graph Data Structure

You must write a class `FloorGraph` that represents the map of a floor.

The `__init__` method of the `FloorGraph` would take as an input a list of `paths` represented as a list of tuples (u, v, x) where:

- u is the starting location ID for the path. This is a non-negative integer.
- v is the ending location ID for the path. This is a non-negative integer.
- x is the amount of time needed to travel down the path from location-u to location-v. This is a non-negative integer.
- You cannot assume that the list of tuples is in any specific order.
- You cannot assume that the paths are 2-way paths.
- You can assume that the location IDs are continuous from 0 to $|V| - 1$ where $|V|$ is total number of locations.
- You can assume that all locations are connected by at least 1 path.
- The total number of paths $|E|$ can be significantly smaller than $|V|^2$ and therefore you should not assume that $|E| = \Theta(|V|^2)$.

The `__init__` method of the `FloorGraph` also takes as an input a list of `keys` represented as a list of tuples (k, y) where:

- k is the location ID where a key can be found in the floor. This is a non-negative integer.
- y is the amount of time needed to defeat the monster and retrieve the key if you choose to fight the monster. This is a non-negative integer.
- You cannot assume that the list of tuples is in any specific order.
- You can assume that all of the k values are from the same set as the location ID, that is from the set of $\{0, 1, 2, ..., |V| - 1\}$.
- You can assume that each of the k values in the list `keys` list is unique.

Consider the following example in which the `paths` and `keys` are stored as a list of tuples:

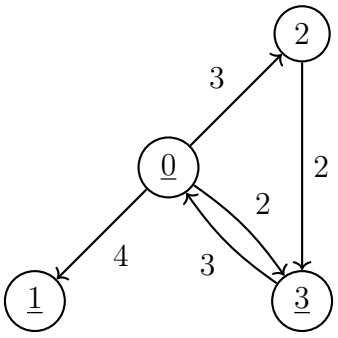
```
# The paths represented as a list of tuples
paths = [(0, 1, 4), (0, 3, 2), (0, 2, 3), (2, 3, 2), (3, 0, 3)]
# The keys represented as a list of tuples
keys = [(0, 5), (3, 2), (1, 3)]
```

There are a total of 5 `paths`, connecting a total of 4 locations from ID 0 to 3 in the floor. The first tuple $(0, 1, 4)$ can be read as – there is a path from location-0 to location-1 and traversing this path takes 4 minutes.

There are a total of 3 `keys` in the floor. The key in location-0 requires 5-minutes to collect, the key in location-3 requires 2 minutes to collect and the key in location-1 requires 3 minutes to collect.

Running the following code would create a `FloorGraph` object. We visualised the graph below for your viewing with the `keys` location underlined; you do not need to visualize the graph in your solution.

```
# Creating a FloorGraph object based on the given paths and keys
myfloor = FloorGraph(paths, keys)
```



2.2 Optimal Route Function

You would now proceed to implement `climb(self, start, exits)` as a function within the `FloorGraph` class. The function accepts 2 arguments:

- `start` is a non-negative integer that represents the starting point in the floor. You begin from here and there is only a single start only in the floor.
- `exits` is a non-empty list of non-negative integers that represents the exit points in the floor, in order to move on to the next floor.
- Do note that it is possible for the `keys` to be in the same location as the `start` and/or `exits`. As stated earlier, you can choose to spend time to engage the monster and collect the key, or to not waste any time and travel along another path without collecting the key.

This function would return one shortest route from `start` to one of the `exits` points that leads to the next floor. This route would need to include defeating one monster and collecting the key in order to go up to the next floor. Thus the function would return a tuple of `(total_time, route)`:

- `total_time` is the time taken to complete the route.
- `route` is the shortest route as a list of integers that represent the location IDs along the path. If there are multiple routes that satisfy the constraints stated, return any one of those routes.

If no such route exist, then the function would return `None`.

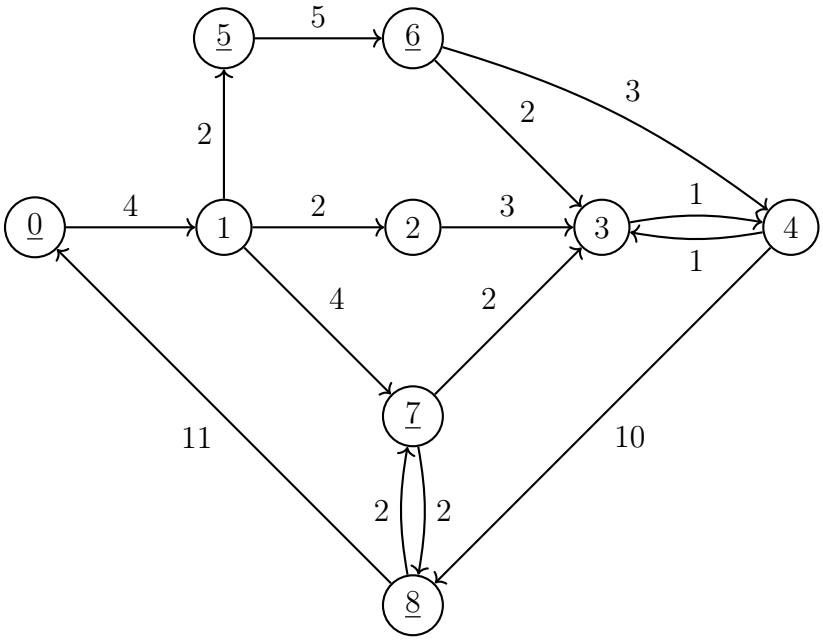
Several examples are provided in Section 2.3.

2.3 Examples

Consider the example floor map below:

```
# Example 1
# The paths represented as a list of tuples
paths = [(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 keys represented as a list of tuples
keys = [(5, 10), (6, 1), (7, 5), (0, 3), (8, 4)]

# Creating a FloorGraph object based on the given paths
myfloor = FloorGraph(paths, keys)
```



Running the following functions will yield:

```
# Example 1.1
start = 1
exits = [7, 2, 4]

>>> myfloor.climb(start, exits)
(9, [1, 7])
```

The simple Example 1.1 above is just going from location-1 to location-7, collecting the key at location-7, adding 5 minutes to defeat the monster. Thus the total trip takes 9 minutes total.

```
# Example 1.2
start = 7
exits = [8]

>>> myfloor.climb(start, exits)
(6, [7, 8])
```

On the other hand in Example 1.2, going from location-7 to location-8, we would collect the key at location-8 because we would only need to spend 4 minutes to defeat the monster 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 `exits` to contain a key.

```
# Example 1.3
start = 1
exits = [3, 4]

>>> myfloor.climb(start, exits)
(10, [1, 5, 6, 3])
```

In Example 1.3, there are multiple possible routes that pass through a key. One of them is $[1, 5, 6, 3]$ with a total time of 10 minutes, defeating the monster at location-6 within 1 minute. Another is $[1, 7, 3]$ with a total time of 11 minutes because defeating the monster at location-7 requires an extra 5 minutes. There are other routes but those would be slower routes such as the ones that ends at location-4, especially the ones with cycles.

```
# Example 1.4
start = 1
exits = [0, 4]

>>> myfloor.climb(start, exits)
(11, [1, 5, 6, 4])
```

In Example 1.4 there are 2 shortest route with the same time of 11 – $[1, 5, 6, 3, 4]$ and $[1, 5, 6, 4]$, both spending 1 minute to defeat the monster at location-6. For such scenario, you can return any of the 2 routes. Any route that exits at location-0 would take too long.

```
# Example 1.5
start = 3
exits = [4]

>>> myfloor.climb(start, exits)
(20, [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 the key at location-8, adding 4 minutes. This showcase an example where a key location is after one of the `exits` location for the optimal solution.

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.

2.4 Complexity

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

The `__init__(paths, keys)` constructor of `FloorGraph` class would run in $O(|V| + |E|)$ time and space where:

- V is the set of unique locations in `paths`. 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 `paths`.
- The number of edges $|E|$ can be significantly smaller than $|V|^2$. Thus, you should not make the assumption that $|E| = \Theta(|V|^2)$.
- Note that the `keys` is not stated in the complexity. At worst, the size of `keys` is $|V|$.

The `climb(self, start, exits)` of the `FloorGraph` class would run in $O(|E| \log |V|)$ time and $O(|V|+|E|)$ auxiliary space. This would run with the same complexity for any combination `start` and `exits`; for any size of `exits`.

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!**