FIT2004 2024 Semester 1: Assignment 1

DEADLINE: Friday 26^{th} April 2024 23:55:00 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 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 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. 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!

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-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:
    :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 Question 1: Ultimate Fuse (10 marks, including 2 marks for documentation)

You are an adventurer in FITWORLD – a magical world where humans and FITMONs live in harmony. FITMONs are insanely cute creatures [1] which make everyone around them happy. Each FITMON has a `cuteness_score` where a higher score means a cuter FITMON.

Recently, it was discovered that it is possible to fuse FITMONs together. The fusing process could increase or decrease the `cuteness_score` of the resulting FITMON. Thus, you set out to fuse FITMONs together, to create the very cutest FITMON possible, that no FITMON ever was. In order to do so, you head over to a FITMON Center.

1.1 Input

You have a list of `fitmons`:

- Contains N `fitmon` in the list $[0...N-1]$. N is a non-zero, positive integer where $N > 1$.
- Each `fitmon` is identified by their index in the `fitmons` list.
- Each `fitmon` in the list is a list of 3 values `[affinity_left, cuteness_score, affinity_right]`.
- `affinity_left` is a positive float in the range of 0.1...0.9 inclusive. Only the left-most `fitmons[0]` will have an `affinity_left` of 0 as there is no `fitmon` on the left for it to fuse.
- `affinity_right` is a positive float in the range of 0.1...0.9 inclusive. Only the right-most `fitmons[N-1]` will have an `affinity_right` of 0 as there is no `fitmon` on the right for it to fuse.
- `cuteness_score` is a non-zero, positive integer.

An example of the input list `fitmons` is illustrated below:

```
fitmons = [
    [0, 29, 0.9],
    [0.9, 91, 0.8],
    [0.8, 48, 0.2],
    [0.2, 322, 0]
]
```

In this input, `fitmons[1]` has a:

- `affinity_left` of 0.9.
- `cuteness_score` of 91.
- `affinity_right` of 0.8.

[1] Not to be confused with other P-Mons.

1.2 Fusing Logic

From the `fitmons` list, you realize that each `fitmon` can only fuse with the adjacent `fitmon` in the adjacent `fitmon`. Your goal is to fuse all of the given `fitmons` into only 1 `fitmon`. Thus:

- `fitmons[i]` can only fuse with either `fitmons[i-1]` or `fitmons[i+1]`.
- The affinity for the 2 fusing `fitmons` are the same as illustrated in the example input. We see that `fitmons[i][0]` would have the same value as `fitmons[i-1][2]` and similarly `fitmons[i][2]` would have the same value as `fitmons[i+1][0]`.
- However, `fitmons[0]` can only fuse with `fitmons[1]` as there is no `fitmon` on its left.
- Likewise, `fitmons[N-1]` can only fuse with `fitmons[N-2]` as there is no `fitmons` on its right.
- Once a `fitmon` is fused, it no longer exist and thus cannot be used for fusing again.

When 2 `fitmons` fuse, their cuteness changes based on their `cuteness_score` and the `affinity` of the fuse. Fusing `fitmons[i]` with `fitmons[i+1]` will create a new `fitmon` with:

- The `affinity_left` will be based on the `affinity_left` of the left `fitmon`,
  `affinity_left = fitmons[i][0]`
- The `cuteness_score` is computed using the `affinity_score` of the fusing `fitmons` multiplied with their `cuteness_score` based on the following equation,
  ```
  cuteness_score = fitmons[i][1] * fitmons[i][2] +
                   fitmons[i+1][1] * fitmons[i+1][0]
  ```
- The `affinity_right` will be based on the `affinity_right` of the right `fitmon`,
  `affinity_right = fitmons[i+1][2]`
- Note: as the `cuteness_score` is an integer, you can use `int()` on it after each and every fuse; before the next fuse (if any).

Using the example input earlier:

- Fusing `fitmons[0]` with `fitmons[1]`, will produce a fitmon with the value of $[0, 108, 0.8]$.
- Fusing `fitmons[1]` with `fitmons[2]`, will produce a fitmon with the value of $[0.9, 111, 0.2]$.
- Fusing `fitmons[2]` with `fitmons[3]`, will produce a fitmon with the value of $[0.8, 74, 0]$.

Therefore, you implement a function called `fuse(fitmons)` which accepts the list `fitmons` and this function would fuse all of the `fitmon` in the list into a single ultimate final `fitmon`. The resulting `fitmon` will have the highest possible `cuteness_score` from the fusing.

1.3 Output

The `fuse(fitmons)` function would return an integer, where it is the `cuteness_score` of the highest possible `cutenness_score` from fusing all of the `fitmons` – if there are N `fitmonns` then you would need $N - 1$ fuses in total. We illustrate a simple example in the next section 1.5.

1.4 Complexity

The function `fuse(fitmons)` must run at the worst-case, $O(N^3)$ time and $O(N^2)$ space where N is the number of items the list `fitmons`. It is possible for your solution to run better than the complexity stated here.

1.5 Example

Consider the following input list of `fitmons` with a total of 3 `fitmon` in it.

```
fitmons = [
    [0, 29, 0.9],
    [0.9, 91, 0.8],
    [0.8, 48, 0]
]
```

Figure 1: The 3 FITMONs from the input.

We can proceed to fuse any 2 of the `fitmons` in Figure 1 as long as they are next to each other. As a start, we can choose to fuse `fitmons[0]` with `fitmons[1]`, creating a `fitmon` with the stats of $[0, 108, 0.8]$ as illustrated below in Figure 2:

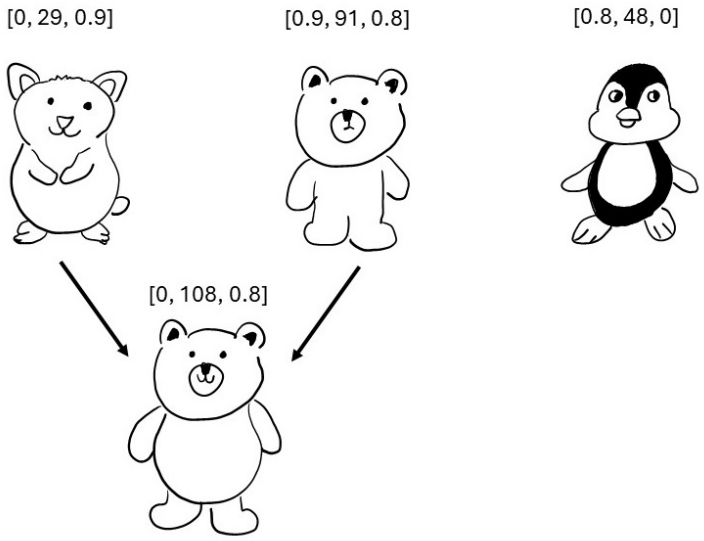


Figure 2: Fusion of `fitmons[0]` with `fitmons[1]`.

Then we can fuse this new `fitmon` with `fitmons[2]`, creating the final `fitmon` with the stats of $[0, 124, 0]$ as illustrated below in Figure 3.

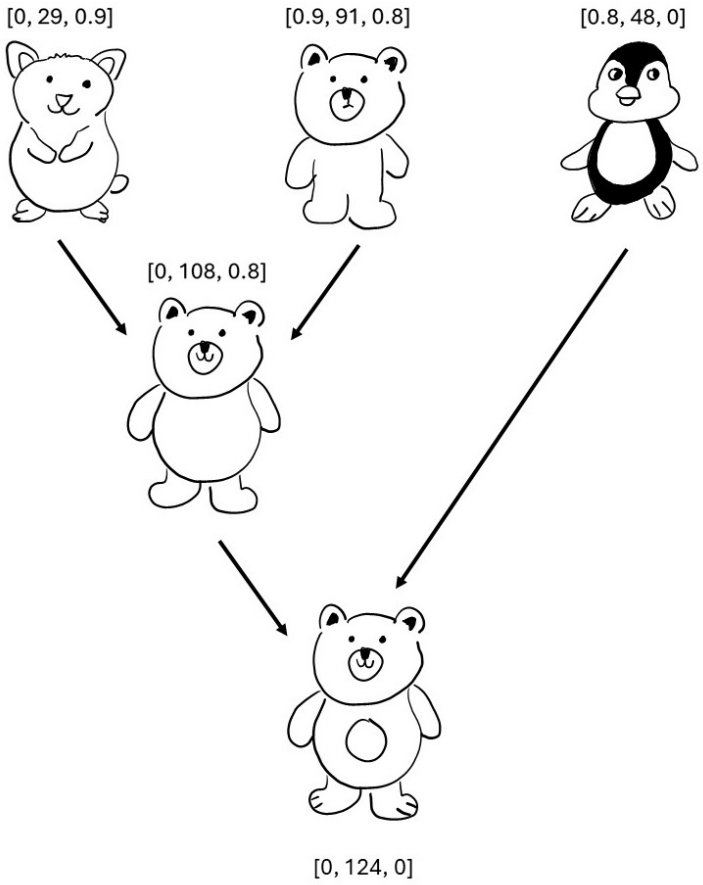


Figure 3: Fusion of `fitmons[0]` with `fitmons[1]`, which is then fused with `fitmons[2]` after.

Alternative, the fusion can be done as illustrated in Figure 4 – where we first fuse `fitmons[1]` with `fitmons[2]` first. Then `fitmons[0]` is fused with the resulting `fitmon`. This creates the final `fitmon` with the stats of $[0, 126, 0]$.

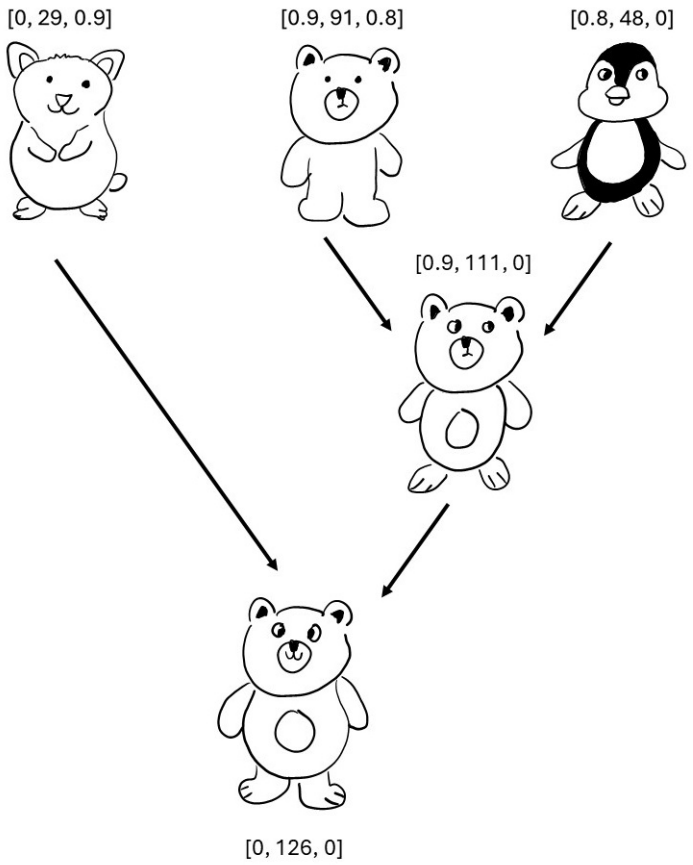


Figure 4: Fusion of `fitmons[1]` with `fitmons[2]`, which is then fused with `fitmons[0]` after.

Putting both fusion order side by side as in Figure 5, we can see that the latter fusion order described (the one on the right) resulted in a cuter final `fitmon` with a `cuteness_score` of 126.

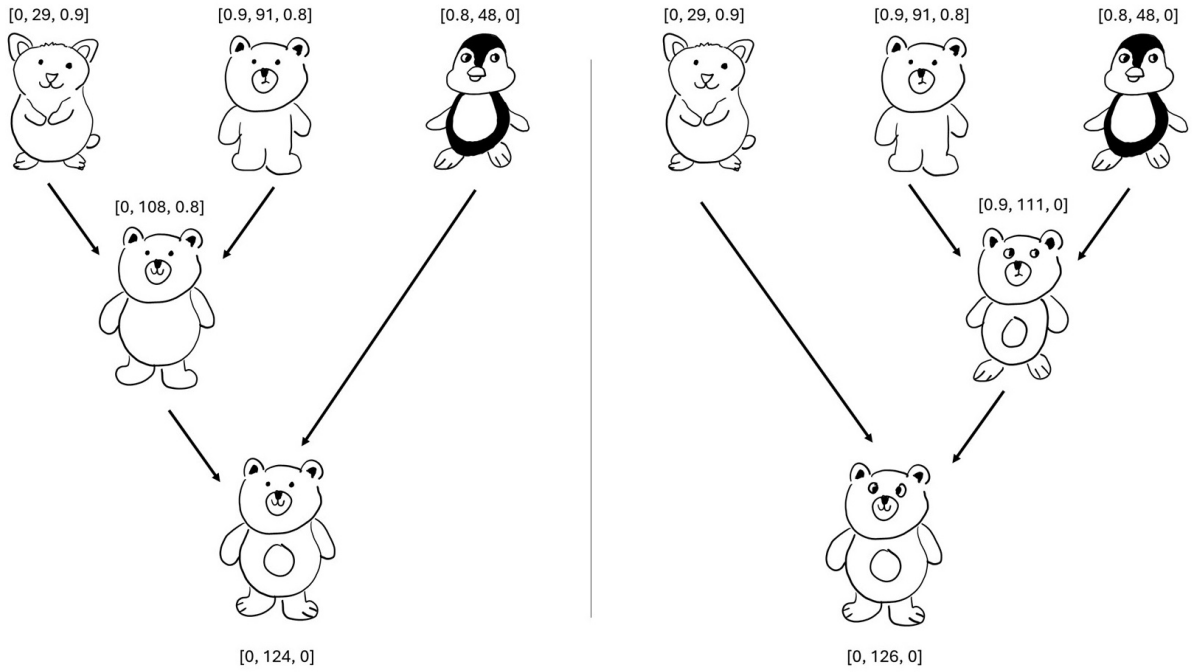


Figure 5: Side-by-side comparison of the fusing orders described earlier. Left has a fusion order of $((0, 1), 2)$ and right has a fusion order of $(0, (1, 2))$.

In summary, the example above can be run using the following code snippet with the resulting returned values.

```
>>> fuse([[0, 29, 0.9], [0.9, 91, 0.8], [0.8, 48, 0]])
126
```

To aid your exploration, consider the following additional examples:

In the example below, the same `fitmons` have been swapped around while retaining their `cuteness_score` and `affinity_score`. This different order will still produce the same optimal fused `fitmon` with the highest `cuteness_score` of 126.

```
>>> fuse([[0, 48, 0.8], [0.8, 91, 0.9], [0.9, 29, 0]])
126
```

```
>>> fuse([[0, 50, 0.3], [0.3, 50, 0.3], [0.3, 50, 0]])
24
```

For the above example, there are several observations to be made. Firstly, all of the `fitmons` have the same `cuteness_score` and the same `affinity_score`. Thus, the order of their fusing does not matter.

Fusing them all together will produce a final `fitmon` with a `cuteness_score` of 24. This value is less than the original `cuteness_score` but as you need to always fuse all of the `fitmons` together, you have no choice but to accept a less cute final outcome. The same can be observed with the next 2 examples below:

```
>>> fuse([[0, 50, 0.6], [0.6, 50, 0.3], [0.3, 50, 0]])
48
```

```
>>> fuse([[0, 50, 0.3], [0.3, 50, 0.3], [0.3, 80, 0]])
33
```

As for the example below, it is just an example of a larger input. Do note that your solution would be tested against very large input sizes in the range of thousands.

```
>>> fuse([[0, 50, 0.6], [0.6, 98, 0.4], [0.4, 54, 0.9], [0.9, 6, 0.3],
        [0.3, 34, 0.5], [0.5, 66, 0.3], [0.3, 63, 0.2], [0.2, 52, 0.5],
        [0.5, 39, 0.9], [0.9, 62, 0]] )
132
```

It is important to note that the examples provided are not exhaustive.There are many other possible cases, with specific edge cases which you would need to validate for your solution to ensure correctness. Thus, do ensure your solution is sufficiently tested using techniques you have learnt in prerequisite units such as unit-testing.

2 Question 2: Delulu is not the Solulu (10 marks, including 2 marks for documentation)

You are a bear stuck in the forest and would like to escape the forest. Unfortunately, the forest itself is known as the Delulu Forest where it is easy to get lost. Your ancestors have left markings on various large trees in the forest to help you escape the forest, where each of the large tree will provide a road to one or more other large tree. This is drawn onto a `treemap` that is given to you:

- There are a total of $|T|$ `trees` in the forest, from t_0 all the way to $t_{|T|-1}$.
- One of the trees would be `start` which is where you begin from.
- One or more trees would be `exits` which is where you can exit from. These trees would be marked in the `treemap` given to you.
- There are $|R|$ roads in total connecting the trees, from r_0 all the way to $r_{|R|-1}$.
- You can go from tree-u to tree-v if a road $r = (u, v)$ exist. However, you can't go from tree-v to tree-u unless the opposite road $r' = (v, u)$ also exist in the `treemap`.
- It takes w-minutes to travel along the road $r = (u, v, w)$. The travel time could differ between, and all of the time to travel along the road is stated in the `treemap` itself.

You then find out the reason why it is called the Delulu Forest – the exit is nothing but a delusion and even after reaching the exit tree, you are still stuck in the forest due to the seal of the forest. However, your ancestor left a hint – certain trees are Solulu trees. You can claw at the tree to destroy that Solulu tree. Doing so will undo the seal of the forest, and then you will be able to exit the forest at an exit tree.

- There are $|S|$ Solulu trees in the forest, from s_0 all the way to $s_{|S|-1}$.
- You would require y-minutes to claw at a Solulu tree-s in order to destroy it.
- Some Solulu trees will teleport you to another tree-t upon destruction. This teleportation might bring you closer to, or further from your exit.

You can't bear to be in the forest anymore and would want to escape as soon as possible. Thus, you use your pawsome computer science knowledge of Graphs to find figure out the quickest way to exit the Delulu forest. You would model the `treemap` using the graph ADT as follow:

```
class TreeMap:
    def __init__(self, roads, solulus):
        # ToDo: Initialize the graph data structure here.
                More details to be described in Section 2.1
    def escape(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 `TreeMap` that represents the trees in the forest and the roads between them.

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

- u is the starting tree ID for the road. This is a non-negative integer, in range of $0...|T|-1$.
- v is the ending tree ID for the road. This is a non-negative integer, in range of $0...|T|-1$.
- w is the amount of time needed to travel down the road from tree-u to tree-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 roads are 2-way roads.
- You can assume that all trees are connected by at least 1 road.
- The total number of roads $|R|$ can be significantly smaller than $|T|^2$ and therefore you should not assume that $|R| = \Theta(|T|^2)$.

The `__init__` method of the `TreeMap` also takes as an input a list of `solulus` represented as a list of tuples (x, y, z) where:

- x is the tree ID where that tree is a Solulu tree in the forest. This is a non-negative integer, in range of $0...|T|-1$.
- y is the amount of time needed to clay and destroy the Solulu tree. This is a non-negative integer.
- z is the tree ID where you will be teleported to if the Solulu tree is destroyed. This is a non-negative integer, in range of $0...|T|-1$. If $x == z$, then it means you will not be teleported.
- You cannot assume that the list of tuples is in any specific order.
- You can assume that each of the x values in the list `solulus` list is unique.

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

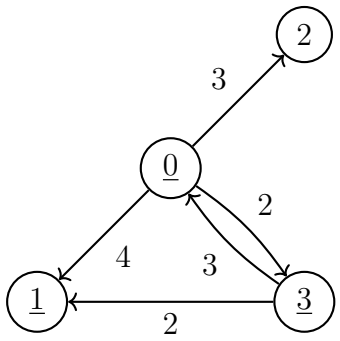
```
# The roads represented as a list of tuples
roads = [(0,1,4), (0,3,2), (0,2,3), (3,1,2), (3,0,3)]
# The solulu represented as a list of tuples
solulus = [(0,5,1), (3,2,0), (1,3,1)]
```

There are a total of 5 `roads`, connecting a total of 4 trees from ID 0 to 3 in the forest. The first tuple $(0, 1, 4)$ can be read as – there is a road from tree-0 to tree-1 and traversing this road takes 4 minutes.

There are a total of 3 `solulus` trees in the floor. Tree-0 is a Solulu tree that requires 5-minutes to destroy; and upon destroying it, you will be teleported to tree-1.

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

```
# Creating a TreeMap object based on the given roads and solulus
myforest = TreeMap(roads, solulus)
```



Of course, you can implement the `TreeMap` class to best solve the problem – adding any additional vertices and edges as you deemed to be appropriate. For example, you might need to represent the teleportation caused by the destruction of Solulu tree-0.

2.2 Optimal Escape Function

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

- `start` is a non-negative integer that represents a tree in the forest. You begin from here and there is only a single starting tree.
- `exits` is a non-empty list of non-negative integers that represents the exit trees in the forest. Arriving on this tree will allow you to escape the forest, as long as you have destroyed a Solulu tree prior.
- Do note that it is possible for the `solulus` to be the same tree as the `start` and/or `exits`. As stated earlier, you can choose: (1) to spend time to claw the tree and destroy it; or (2) to not waste any time and travel along another road.
- You must destroy one of the `solulu` tree in order to be able to exit the forest.
- You can only destroy exactly one `solulu` tree, as your claw will need to rest after.

This function would return one fastest route from `start` tree to one of the `exits` trees. This route would need to include destroying one Solulu tree, in order to break the seal of the forest. Thus the function would return a tuple of `(total_time, route)`:

- `total_time` is the time taken to exit the forest.
- `route` is the shortest route as a list of integers that represent the tree IDs along the road. If there are multiple routes that satisfy the constraints stated, return any one of those routes. In order words, the route is not a unique solution.

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

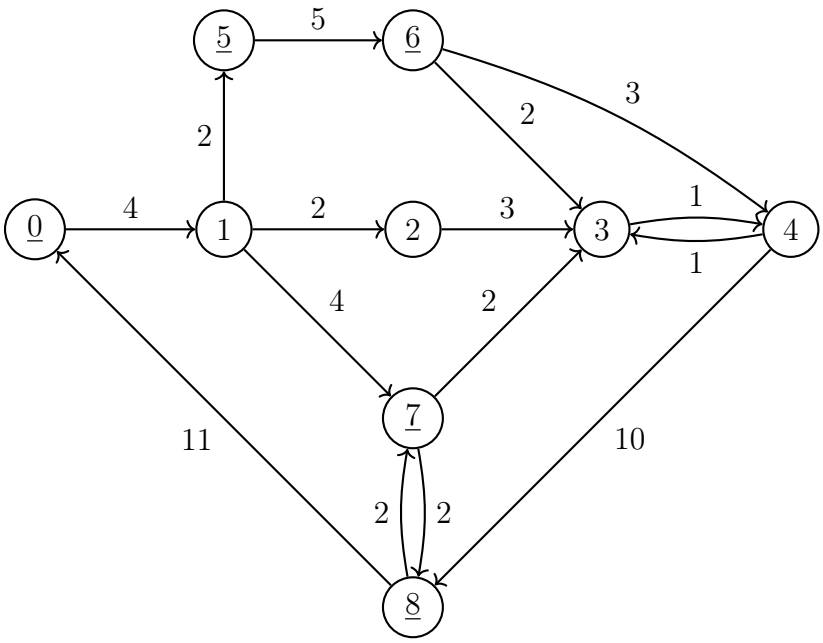
Several examples are provided in Section 2.3.

2.3 Examples

Consider the example floor map below:

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

# Creating a TreeMap object based on the given roads
myforest = TreeMap(roads, solulus)
```



Running the following functions will yield:

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

>>> myforest.escape(start, exits)
(9, [1, 7])
```

The simple Example 1.1 above is just going from tree-1 to tree-7, collecting the destroying tree-7 which requires an additional 5 minutes. Destroying tree-7 does not teleport you anywhere else, thus the total escape takes 9 minutes total.

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

>>> myforest.escape(start, exits)
(6, [7, 8])
```

On the other hand in Example 1.2, going from tree-7 to tree-8, we would destroy tree-8 because we would only need to spend 4 minutes instead of 5 minutes to destroy tree-7. Thus the total time taken is 6 minutes. This example also highlight how it is possible for the `start` and/or `exits` to be a Solulu tree.

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

>>> myforest.escape(start, exits)
(10, [1, 5, 6, 3])
```

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

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

>>> myforest.escape(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 destroy tree-6. For such scenario, you can return any of the 2 routes. Any route that exits at tree-0 would take longer, even if destroying tree-5 would teleport you directly to tree-0 which is an exit as it takes 10 minutes to do so for a total time of 12 minutes.

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

>>> myforest.escape(start, exits)
(20, [3, 4, 8, 7, 3, 4])
```

In Example 1.5 above, we could reach tree-4 from tree-3 but unfortunately we would need to take a detour in order to destroy-8, adding 4 minutes. This showcase an example where a Solulu tree to destroy is after one of the `exits` for the optimal escape.

```
# Example 1.6
start = 8
exits = [2]

>>> myforest.escape(start, exits)
(16, [8, 0, 2])
```

Last example present us with 2 possible options. The first option is $[8, 0, 1, 2]$, taking 4 minutes to destroy tree-8 giving us the total time of 21 minutes. The second option is $[8, 0, 2]$ where we go from tree-8 to tree-0 first, and then spend 5 minutes to destroy tree-0. While destroying tree-0 takes more time than destroying tree-8, by destroying tree-0, we teleport directly to tree-2. This resulted in a total escape time of 16 minutes only. Do note how the escape route is returned in this example.

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__(roads, solulus)` constructor of `TreeMap` class would run in $O(|T| + |R|)$ time and space where:

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

The `escape(self, start, exits)` of the `TreeMap` class would run in $O(|R| \log |T|)$ time and $O(|T| + |R|)$ 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!**