2.1 Graph Data Structure (1 mark)

You must write a class `RoadGraph` that represents the roads in the city. The `__init__` method of `RoadGraph` would take as an input a list of roads `roads` represted 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 distance along that road, 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.
- You can assume that the maximum number of roads $|E|$ is significantly less than $|V|^2$.

Consider the following example for the roads in a city, are stored as a list of tuples:

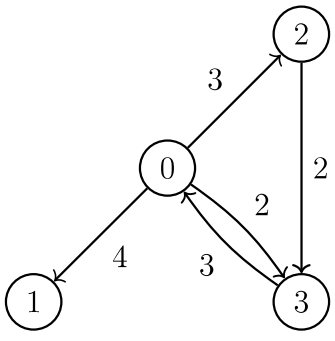
```
# The roads represented as a list of tuples
roads = [(0,1,4), (0,3,2), (0,2,3), (2,3,2), (3,0,3)]
```

- 4 locations in the city with the id of 0 to 3.
- 5 roads total in the city connecting the locations.

- There is a road from location 0 to location 1 with a distance of 4.
- There is a road from location 0 to location 3 with a distance of 2.
- There is a road from location 0 to location 2 with a distance of 3.
- There is a road from location 2 to location 3 with a distance of 2.
- There is a road from location 3 to location 0 with a distance of 3.
- 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 distance.

Running the following code would create a `RoadGraph` object called `mygraph`, which is then visualized below:

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



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.

2.2 Optimal Route Function (3 marks)

You would now proceed to implement `routing(self, start, end, chores_location)` as a function within the `RoadGraph` class. The functions accepts 3 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.
- `chores_location` is a non-empty list of non-negative integers that stores all of the location where your chores could be performed.
- Do note that it is possible for the locations in `chores_location` to include the `start` and/or `end` as well.
- Do note that the locations in `chores_location` is not sorted.
- Do note that all locations in `chores_location` are valid locations in the `roads` list.

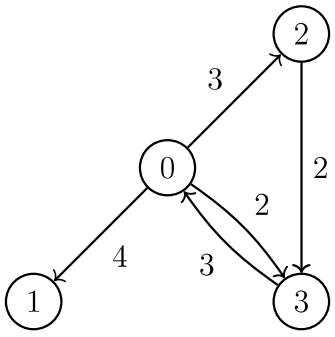
The function would then return the shortest route from the `start` location to the `end` location, going through at least 1 of the locations listed in `chores_location`.

- If such route exist, it would return the shortest route as a list of integers. If there are multiple such routes, return any one of these shortest routes.
- If no such route going from the `start` location to one of the locations in `chores_location` and proceeding next to the `end` location is possible, then the function would return `None`.

Several examples are provided in Section 2.3.

2.3 Examples

Consider the following road network:




```
# Example 1
# The roads represented as a list of tuples
roads = [(0,1,4), (0,3,2), (0,2,3), (2,3,2), (3,0,3)]
# Creating a RoadGraph object based on the given roads
mygraph = RoadGraph(roads)
```

Starting from location 0, you are looking to end at location 1. Your chores can be completed at both location 2 and location 3. The optimal route for you is to travel going through location 3 to complete the chore, then going back to location 0 before ending at location 1. This route would only travel a total distance of 9. The alternative route of going through locations 0, 2, 3, 0 and then 1 would give you a total distance of 10.

```
# Example 1.1
# the route inputs
start = 0
end = 1
chores_location = [2,3]

>>> mygraph.routing(start, end, chores_location)
[0, 3, 0, 1]
```

In another example below, you now start at location 0 but look to end in location 3. Your chore can only be completed in location 2 only. Thus the optimal route is to go from location 0 to location 2 to complete your chore and directly to location 3 that is your destination. Your total travel distance is 5.

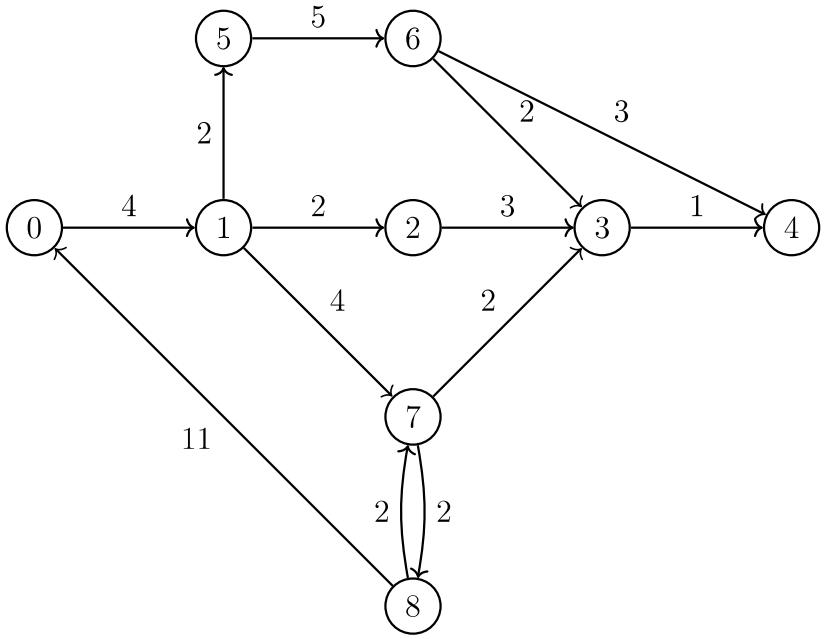
```
# Example 1.2
# the route inputs
start = 0
end = 3
chores_location = [2]

>>> mygraph.routing(start, end, chores_location)
[0, 2, 3]
```

The next 2 examples below shows when routes are not possible; with the function returning `None`.

```
# Example 1.3 Chores can't be done
start = 2
end = 0
chores_location = [1]

>>> mygraph.routing(start, end, chores_location)
None
```

```
# Example 1.4 Can't reach end
start = 1
end = 2
chores_location = [0,3]

>>> mygraph.routing(start, end, chores_location)
None
```

In the following graph example, we look at a bigger, longer and more complex graph.



The graph above is generated through the following code.

```
# Example 2
# 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)]
# Creating a RoadGraph object based on the given roads
mygraph = RoadGraph(roads)
```

In the example below, we start at location 0, end at location 3, and your chores can be completed at locations 5, 7, 8 and 4. There are a few possible routes possible such as $0, 1, 5, 6, 3$ with a distance of 13 or route $0, 1, 7, 3$ with a distance of 10. We would choose the smallest distance.

```
# Example 2.1 Possible Route
start = 0
end = 3
chores_location = [5,7,8,4]

>>> mygraph.routing(start, end, chores_location)
[0, 1, 7, 3]
```

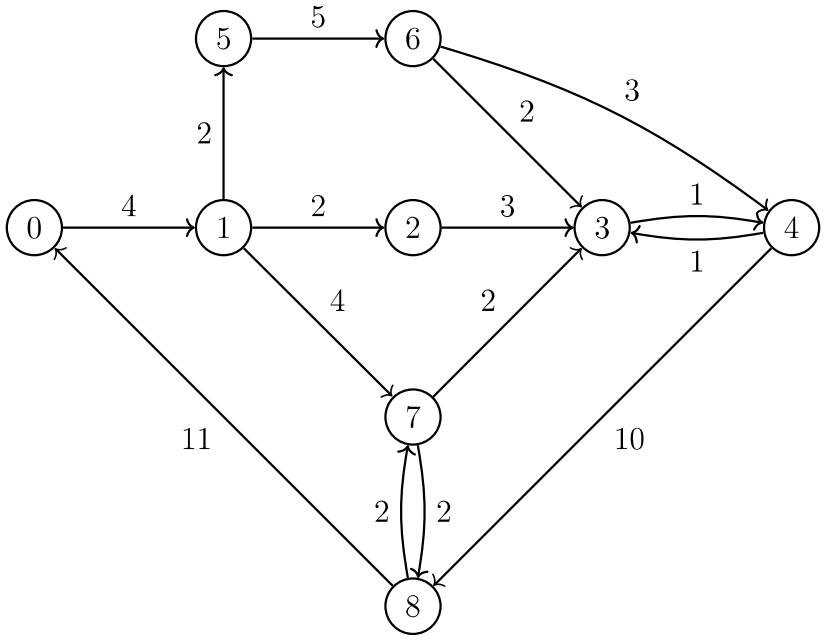
Similarly, if we were to start at location 1 we observe that despite the location 5 would be the closer location to us to perform the chore at a distance of 2 compared to location 7 with a distance of 4; the route after that would be further. Thus, we cannot make be short sighted in our path selection based on the nearest chore.

```
# Example 2.2 Possible Route, despite nearest chore
start = 1
end = 3
chores_location = [5,7,8,4]

>>> mygraph.routing(start, end, chores_location)
[1, 7, 3]
```

In the next example, we saw multiple routes with the same shortest distance – with route $0, 1, 5, 6, 4$ for a distance of 14 and route $0, 1, 5, 6, 3, 4$ for a distance of 14. Returning either would be accepted. Do note that route $0, 1, 7, 8, 7, 3, 4$ has a longer distance of 15 and should not be returned.

```
# Example 2.2 Possible Route, despite nearest chore
start = 0
end = 4
chores_location = [6,8]

>>> mygraph.routing(start, end, chores_location)
[0, 1, 5, 6, 4]
```

For the final example, we have an even more complicated network of roads.




```
# Example 3
# 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)]
# Creating a RoadGraph object based on the given roads
mygraph = RoadGraph(roads)
```

The following are 4 cases that were not highlighted by the earlier examples.

```
# Example 3.1 Best chore location is after end location
start = 1
end = 3
chores_location = [4,5,6,8]

>>> mygraph.routing(start, end, chores_location)
[1, 2, 3, 4, 3]
```

```
# Example 3.2 Best chore is before start location
start = 7
end = 4
chores_location = [5,6,8]

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

```
# Example 3.3 Best chore is alone the shortest route already
start = 0
end = 4
chores_location = [1,3,4,5,6,8]

>>> mygraph.routing(start, end, chores_location)
[0, 1, 2, 3, 4]
```

```
# Example 3.4 There is chore at the start and/ or end location
start = 0
end = 4
chores_location = [0,4,5,6,8]

>>> mygraph.routing(start, end, chores_location)
[0, 1, 2, 3, 4]
```

There could be other cases not covered in the examples provided. Do have a think about what are the other possible scenarios related to the question.

2.4 Complexity

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

The `__init__(raods)` 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`.
- You can make an assumption that the number of roads $|E|$ is significantly less than $|V|^2$. Thus, you should not make the assumption that $|E| = \Theta(|V|^2)$.

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

- Note that the `chores_location` is not stated in the complexity. At worst, the size of `chores_location` is $|V|$.

2.5 Hint(s)

In order to do so, you would utilize algorithms and concepts that you have learnt from FIT2004, namely:

- Graph data structure.
- Graph algorithms.

Firstly, ensure a suitable graph representation is chosen to meet the complexity requirement for `__init__(roads)` constructor of `RoadGraph` class. A suitable representation would also ensure the complexity requirements for `routing(self, start, end, chores_location)` is met as well.

The complexity for `routing(self, start, end, chores_location)` can be met by running a certain algorithm once (approach 1) or twice (approach 2) at most [1]. Do take note that this complexity remains the same irregardless of the number of locations in `chores_location`; thus, the solution shouldn't scale in complexity to that.

You are allowed to process the graph data structure/representation in `__init__(roads)` as part of the pre-processing for `routing(self, start, end, chores_location)`. Doing so as you seen in some of the studio questions could greatly improve the complexity for `routing(self, start, end, chores_location)`.

[1] There can be other approaches that would meet the complexity requirements as well.