Task 0

The Graph class represents the Graph and its methods and attributes.
The graph is a simple undirected graph
The graph is such that each vertex represents a word, and there is an edge between two vertices if and only if those two words differ by exactly one character.

def __init__(self,vertices_filename, edges_filename):

The attributes of the graph are initialized.
The graph is given as two files, the first containing information about the vertices, and the second containing information about the edges.

This function takes two file-names as inputs, and reads the data from them.
The first line of vertices_filename has a number, n, which is the number of vertices in the graph. The next n lines each start with a unique integer between 0 and n − 1 inclusive (the vertex ID) and then a space, and then a word (the word which that vertex represents).
The words will contain only lowercase English alphabet characters.
We assume that all words are constant size.

Each line of edges_filename consists of two numbers, separated by a space.
Each of these pairs of numbers represents an undirected edge connecting the two vertices with those IDs.
You are guaranteed that all the numbers in edges_filename are valid vertex IDs (i.e. are integers between 0 and n − 1 inclusive).
The edges are in no particular order.
This innit method will correctly populate an instance of your Graph class, taking into consideration each vertice and all the vertices linked to that vertice via an edge.

We go about populating the graph class by iterating over each line of the vertice file, and since the first line contains the number of vertices (n) in the graph we create an attribute of the graph class self.vertexArray which is a None-type array of size n.

Each line after the first line of the file contains the vertex data, in the format:
Verted ID   (space)   Word
Where the next n lines each start with a unique integer between 0 and n − 1 inclusive (the vertex ID) and then a space, and then a word (the word which that vertex represents).
In each line of vertex data read we create an instance of the Vertex Class [O(1)] and complete the attributes of the Vertex with vertex ID and word, the Vertex object is the added to the vertexArray at its correct position based on the VertexID [done in O(1)].
Eg: vertexArray[VertexID] = Vertex (object)

This process continues for each vertex data read in from the vertices file and the loop essentially executes for O(V+1) where V is the number of vertices which is the number of lines (after the first line) with vertices data. The +1 comes from the first line containing an integer representing the number of vertices in the graph. Therefore the time complexity of this operation evaluates to O(V+1) == O(V) where V is the number of lines in the vertices file.

In the next step of the innit function we iterate over each line in the edges file.
In each iteration we create an instance of the edge class [O(1)] and add the data of that instance of the edge with the data in the line being read.
This edge object is then added to the 'list of edges' of the correct vertex of the vertexArray. This is done by calling a method in the correct instance of the Vertex object in the vertexArray of the Grapgh class
Eg: for edge (u,v) the edge object is added to self.vertexArray[u].add_edge(edge)
This operation is done in O(1)

We also create a second instance of the edge object (to account for the reverse edge since edges are bi-directional). Eg: edge (v,u)
This second instance the edge object is then added to the correct vertex of the vertexArray in the same way as described above.
Eg: for edge (v,u) the edge object is added to self.vertexArray[v].add_edge(edge)
This operation is done in O(1)

This process continues for each edge data read in from the edges file and the loop essentially executes for O(E) where E is the number of edges, which is also the number of lines in the edges file.

Overall the innit function of the graph class executes in O(V+E)
Where V is the number of lines in the vertices file and E is the number if lines in the edges file.

Task 1:

The solve ladder function finds the shortest list of intermediate words which solves it (or reports that no such list exists)

The function receives an input start_vertex and target_vertex. These are integers between 0 and V − 1 inclusive, where V is the number of vertices in the graph.

We start off by creating a 'discovered' Queue (implementation of Queue obtained from content learned in FIT2085).
This 'discovered' Queue will contain the vertices being discovered as you explore the edges connected to the visting vertex, starting from the source vertex which is the first vertex added to the queue at the very beginning.

We then call the function bfs (breadth first search) providing the queue and start vertex as parameters.

In the breadth first search function we start off by removing the first vertex object from the queue (vertex at the front of the queue) and set the 'visited' attribute of the vertex as True, and we iterate over all the edges connected to that vertex.

In an edge,
if the "destination vertex" of that edge is not discovered nor visited {

> The destination vertex is added to the discovered Queue and the discovered attribute of the vertex is set to True.
> We also update the distance of that destination vertex based on a calculation considering the distance of the current visiting vertex (from source), and the predecessor of the destination vertex is set to the current visiting vertex [O(1)].

If the "destination vertex" is already discovered {

> We calculate the "current distance" from the source vertex, by taking into consideration the distance of the current visiting vertex (from source).
> We then proceed to check if the "current distance" is less than the distance mentioned in the distance attribute of the "destination vertex". If so, the distance attribute of the "destination vertex" is now updated with the "current distance" [O(1)].

This process now continues for all the edges that the vertices in the discovered Queue have. The BFS function executes in O(V+E). Where E is the total number of edges in the graph, and V is the total number of vertices in the graph

The solve_ladders function then calls a function findWords providing the target vertex and start vertex as parameters.
This function is now called recursively to essentially ‘backtrack’ from the target Vertex to the start Vertex using the predecessor as the new “target Vertex” at each iteration of the recursive call.
At each iteration of the recursive call the word contained in the predecessor is appended to the list of words, and the list of words is returned when the target vertex == start vertex and the backtracking stops.

Backtracking from Target Vertex to End Vertex is O(V) In the worst case.

Therefore, overall the solve_ladder function executes in O(V+E) Where E is the total number of edges in the graph, and V is the total number of vertices in the graph

Task 2:

In task 2 we want to find the cheapest word ladder. The cost of a word ladder is found by summing the the costs of the changes made to characters during that ladder. The cost of changing a character from given by squaring the difference in alphabet position of the two characters.

However, there is an additional requirement. In this task, the word ladder must also contain at least one word which starts with a given letter. This letter will be one of our input parameters.

start_vertex, target_vertex and req_char. start_vertex and target_vertex are integers between 0 and V − 1 inclusive, where V is the number of vertices in the graph. req_char is a lowercase english alphabet character.
cheapest_ladder will return two things as a tuple (or False). The first is the cost of the cheapest word ladder. The second is a list of strings, which are the intermediate words in the cheapest word ladder starting at start_vertex and ending at target_vertex, such that at least one word in the ladder has req_char as its first character (This can be the the word associated with start_vertex or target_vertex).
In some cases, it may not be possible to find a chain from the start_vertex to the target_vertex with at least one word in the ladder has req_char as its first character . In these cases, the function returns False.

To carry out task 2 we have implemented dijkstras algorithm which can be used to find the cheapest path from start vertex to all the vertexes in the connected graph.
Because we have to also ensure that the final path contains the character being seeked we have coded a modified version of dijkstras algorithm.

The Dijkstra algorithm now does a check whether we come across a vertex whose word has the desired first character, if not the cheapest_ladder fucnntion returns False.
If there is indeed a vertex with word starting with the desired char, the function returns the vertex ID of first vertex we come across with the desired char in its word, since this is the cheapest option.

The rest of the functionality of Dijkstras Algorithm is as standard and finds the cheapest path from the start vertex to all the vertices in the connected graph.

```
def cheapest_ladder(self, startVertexIndex, targetVertexIndex, reqChar):
```

In the chapest_ladder function, we call dijkstras Algorithm once giving the start index as startVertexIndex. This initial call of Dijkstras [O(ElogV)] returns the 'pointer' variable containing the vertex ID of the first Vertex we come across with the char as the first name of vertex.word, this is also the cheapest vertex with char in word (as first character).

We then make a call to findWords2(self, vertex, startVertexID,list) giving the 'vertex' param as vertexArray[pointer] and the second param as startVertex which in this instance the source vertex of the graph as determined by user.
This function will then essentially backtrack using recursion and find the path from the pointer to the start vertex, by using the predecessor of the 'vertex' at each recursive call. Prior to each recrursive call we append the vertex.word to the 'list' which is returned after the final recursive call.
We now have the cheapest path from the start vertex to the 'pointer' (first vertex we come across in Dijkstras with char as first character of word).

We now run a second Dijkstras algorithm giving the startVertex as the 'pointer' (vertex ID of cheapest vertex with char as the first character of word) we obtained from the first Dijkstra call.
This second Dijkstra call all of the cheapest paths from the cheapest vertex with char (as first character of word) to all other vertices in the connected grapgh.
Afterward we make another call to findWords2(self, vertex, startVertexID, list) giving the 'vertex' param as vertexArray[targetVertexIndex] and the startVertexID as the 'pointer' returned from the first execution of Dijkstra.

This function will then essentially backtrack using recursion and find the path from the pointer to the start vertex (pointer), by using the predecessor of the 'vertex' at each recursive call. Prior to each recrursive call we append the vertex.word to the 'list' which is returned after the final recursive call. We now have the cheapest path from the 'pointer' to the targetVertex (final target vertex as specified by the user in second param of cheapest_ladder func).

We now combine the 2 paths and we get the full path from the startVertex to targetVertex, containing the cheapest vertex containing 'char' as its first character of the vertex.word (if such a vertex exists).

This whole cheapest_ladder function excutes in O(E log(V)) time, where
V is the number of vertices in the graph
E is the number of edges in the graph