1. BACKGROUND

Plantix network is home to a wide variety of agricultural professionals. Each one of them is associated with a list of crops or *topics* they can provide advice on to help fellow experts. The *plantix.sdk* Python module can query such network, and can be extended to extract valuable information from it.

2. SCOPE OF THE CHANGE

Within this context, the development of a new feature is proposed. Given a user ID, the goal is to compile the list of topics that the given user is able to access via its network, regardless of the depth of connectivity (direct connections and connections of connections as well). To complete the returned information, the number of occurrences for each topic will also be included.

The *plantix.sdk* module shall include a method with the following signature:

```
def find_topics(self, start: str, n: int) -> List:
'''Implement logic here'''
```

Inputs:
- *start:* user in the network of experts to start the search from
- *n:* most common n topics to be displayed in the result

Output:
- a list of tuples, where each tuple contains the *topic* and the number of times it appeared in the search. For instance, if user 5's network contains 10 experts in cucumber, 6 in tomato and 4 in wheat, and given that the parameter *n* equals 2 (the two most common topics), the result shall be:

```
[
        ('cucumber', 10),
        ('tomato',    6)
   ]
```

3. SOLUTION

3.1. Description
Conceptually, such network of crop experts can be modeled as a graph, in which nodes represent experts, and edges are connections among them. A slight simplified (in the sense that not all nodes may not need to be visited) version of the Depth First Search (DFS) algorithm may be fit to extract this information.

A solution has been provided which is based on a recursive implementation of the above mentioned algorithm. Its characteristics, advantages and disadvantages that will be analyzed in the following section.

3.2. Analysis
- Advantages
  - Simplicity. The code is simple, easy to read and understand
  - Complexity. The algorithm's work can be divided into two steps.
    - In the first one, some amount of fixed work is done; essentially marking a vertex (expert) as visited. At worst, the time it will take to complete this work is proportional to the number of vertices (experts).

    - In the second one, a loop is executed in which connection between two users will be checked twice, one for each user. At worst case, its time complexity is proportional to the number of edges, or connections between users.
      Overall, the algorithm has a time linear time complexity, O(V + E), where V and E denote the number of vertices and edges between them.
    - Since the algorithm has been implemented using recursion, its memory footprint depends on the size of the stack created when doing recursive calls. Since only vertices that have not been visited yet are visited, the stack size will be proportional to this number, that is, O(V).

- Disadvantages
  - Querying the Plantix API is coupled with the algorithm that traverses the graph, which makes it more difficult to test it without using mock objects or artifacts of the like.

4. IMPROVEMENT

To solve the issue about the coupling between the API query and the graph traversal, one could write another version of the same algorithm in which the Plantix network is queried first so that the whole connectivity graph is built.
After that, the DFS algorithm will traverse the graph and extract the desired information.

- Advantages.
  - As mentioned before, decoupling between querying the Plantix API and graph traversal, which makes easier the task of writing tests.

- Disadvantages.
  - The memory complexity will be higher. If the graph is represented using a matrix, in which columns and rows represent all nodes that may be connected together, complexity will be O(V^2). This technique would be wasteful if the graph doesn't have many edges (users connected).
    Representing the graph using adjacent lists may be an advantage if the graph is dense. Following this technique, each node *n* will point to a list containing all nodes *n* is pointing to. The memory complexity will be in this case O(E).