Full Name:	Remi Chartier
Email:	remipr.chartier@gmail.com
Test Name:	**Mock Test**
Taken On:	31 Oct 2021 12:55:38 IST
Time Taken:	0 min 23 sec/ 28 min
Contact Number:	+14084751573
Linkedin:	http://www.linkedin.com/in/remichartier
Invited by:	Ankush
Invited on:	31 Oct 2021 12:55:30 IST
Skills Score:	
Tags Score:	Algorithms 0/100 Core CS 0/100 Graph Theory 0/100 Medium 0/100 problem-solving 0/100

0%

0/100

scored in **Mock Test** in 0 min 23 sec on 31 Oct 2021 12:55:38 IST

Recruiter/Team Comments:

No Comments.

	Question Description	Time Taken	Score	Status
Q1	Breadth First Search: Shortest Reach > **Coding**	16 sec	0/ 100	⊗

QUESTION 1

⊗ Wrong Answer

Score 0

Breadth First Search: Shortest Reach > Coding

Graph Theory | Algorithms | Medium | problem-solving | Core CS

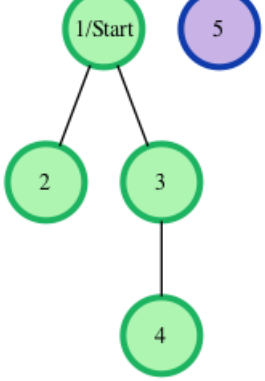
QUESTION DESCRIPTION

Consider an undirected graph where each edge weighs 6 units. Each of the nodes is labeled consecutively from 1 to n.

You will be given a number of queries. For each query, you will be given a list of edges describing an undirected graph. After you create a representation of the graph, you must determine and report the shortest distance to each of the other nodes from a given starting position using the *breadth-first search* algorithm (BFS). Return an array of distances from the start node in node number order. If a node is unreachable, return -1 for that node.

Example

The following graph is based on the listed inputs:



$n = 5$ // number of nodes
$m = 3$ // number of edges
$edges = [1, 2], [1, 3], [3, 4]$
$s = 1$ // starting node

All distances are from the start node 1. Outputs are calculated for distances to nodes 2 through 5: $[6, 6, 12, -1]$. Each edge is 6 units, and the unreachable node 5 has the required return distance of -1.

Function Description

Complete the *bfs* function in the editor below. If a node is unreachable, its distance is -1.

bfs has the following parameter(s):

- *int n*: the number of nodes
- *int m*: the number of edges
- *int edges[m][2]*: start and end nodes for edges
- *int s*: the node to start traversals from

Returns
int[n-1]: the distances to nodes in increasing node number order, not including the start node (-1 if a node is not reachable)

Input Format

The first line contains an integer q, the number of queries. Each of the following q sets of lines has the following format:

- The first line contains two space-separated integers n and m, the number of nodes and edges in the graph.
- Each line i of the m subsequent lines contains two space-separated integers, u and v, that describe an edge between nodes u and v.
- The last line contains a single integer, s, the node number to start from.

Constraints

- $1 \leq q \leq 10$
- $2 \leq n \leq 1000$
- $1 \leq m \leq \frac{n \cdot (n-1)}{2}$
- $1 \leq u, v, s \leq n$

Sample Input

```
2
4 2
1 2
1 3
1
3 1
2 3
2
```

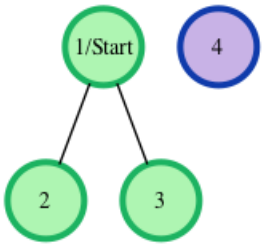
Sample Output

```
6 6 -1
-1 6
```

Explanation

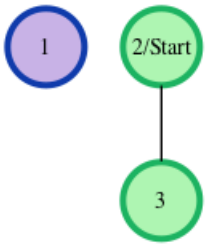
We perform the following two queries:

1. The given graph can be represented as:



where our *start* node, s, is node 1. The shortest distances from s to the other nodes are one edge to node 2, one edge to node 3, and an infinite distance to node 4 (which it is not connected to). We then return an array of distances from node 1 to nodes 2, 3, and 4 (respectively): $[6, 6, -1]$.

2. The given graph can be represented as:



where our *start* node, s, is node 2. There is only one edge here, so node 1 is unreachable from node 2 and node 3 has one edge connecting it to node 2. We then return an array of distances from node 2 to nodes 1, and 3 (respectively): $[-1, 6]$.

Note: Recall that the actual length of each edge is 6, and we return -1 as the distance to any node that is unreachable from s.

CANDIDATE ANSWER

Language used: **Python 3**

```
#
# Complete the 'bfs' function below.
#
# The function is expected to return an INTEGER_ARRAY.
# The function accepts following parameters:
#  1. INTEGER n
#  2. INTEGER m
#  3. 2D_INTEGER_ARRAY edges
#  4. INTEGER s
#


class Node:
    def __init__(self, value):
        self.value = value
        self.edges = []
        self.seen = False
        self.dist = 0

    def add_edge(self, number):
        self.edges.append(number)

    def reset_seen_dist(self):
```

```
        self.seen = False
        self.dist = 0

    def set_seen(self):
        self.seen = True

    def set_dist(self,a):
        self.dist = a

class Graph:
    def __init__(self):
        self.nodes = [Node(0)]

    def reset_seen_dist(self):
        for n in self.nodes:
            n.reset_seen_dist()


# start = start node number (1 to n)
# target = searched node number (1 to n)
def find_bfs(start, target, g):
    curr = g.nodes[start]
    links = []
    dist = 0
    if target == curr.value:
        return 0
    curr.set_seen()
    #print(f'node {start} current edges {curr.edges}')
    for e in curr.edges:
        if not g.nodes[e].seen:
            g.nodes[e].set_seen()
            g.nodes[e].dist = g.nodes[e].dist + 6
            #print(f"1st loop : g.nodes[{e}].dist = {g.nodes[e].dist}")

        links.append(e)

    while len(links) != 0:
        node_number = links.pop(0)
        if target == node_number:
            return g.nodes[node_number].dist
        curr = g.nodes[node_number]
        for e in curr.edges:
            if not g.nodes[e].seen:
                g.nodes[e].set_seen()
                g.nodes[e].dist = g.nodes[node_number].dist + 6
                #print(f" 2nd loops : g.nodes[{e}].dist = {g.nodes[e].dist}")
                links.append(e)
    return -1


def bfs(n, m, edges, s):
    # Write your code here
    #returns int[n-1] distances to nodes in increasing node number order,
    # not including the start node
    g = Graph()
    # create nodes, add them to the graph
    for i in range(1,n+1):
        g.nodes.append(Node(i))
    # fill list of edges on the nodes
    for e in edges:
        g.nodes[e[0]].add_edge(e[1])
        #g.nodes[e[1]].add_edge(e[0])
    # Now the Graph is filled.
    # We should now be able to implement BFS ...
```

```
    # build a list of distances from node s
    output = []
    for i in range(1, n+1):
        if i != s:
            g.reset_seen_dist()
            #print(f"find_bfs({s}, {i}, g)")
            output.append(find_bfs(s, i, g))
    #print(output)
    return output


```

TESTCASE	DIFFICULTY	TYPE	STATUS	SCORE	TIME TAKEN	MEMORY USED
Testcase 1	Easy	Sample case	Success	0	0.0591 sec	9.55 KB
Testcase 2	Medium	Hidden case	Wrong Answer	0	0.0563 sec	9.84 KB
Testcase 3	Medium	Hidden case	Wrong Answer	0	0.4681 sec	12.8 KB
Testcase 4	Hard	Hidden case	Wrong Answer	0	0.0492 sec	9.46 KB
Testcase 5	Hard	Hidden case	Wrong Answer	0	0.0704 sec	9.89 KB
Testcase 6	Hard	Hidden case	Wrong Answer	0	6.9075 sec	26 KB
Testcase 7	Hard	Hidden case	Wrong Answer	0	0.3464 sec	10.7 KB
Testcase 8	Easy	Sample case	Success	0	0.0484 sec	9.59 KB

No Comments