Full Name:	Remi Chartier
Email:	remipr.chartier@gmail.com
Test Name:	**Mock Test**
Taken On:	2 Nov 2021 07:57:58 IST
Time Taken:	4 min 5 sec/ 28 min
Contact Number:	+14084751573
Linkedin:	http://www.linkedin.com/in/remichartier
Invited by:	Ankush
Invited on:	2 Nov 2021 07:57:52 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 4 min 5 sec on 2 Nov 2021 07:57:58 IST

Recruiter/Team Comments:

No Comments.

	Question Description	Time Taken	Score	Status
Q1	Breadth First Search: Shortest Reach > **Coding**	3 min 42 sec	0/ 100	⊗

QUESTION 1

⊗ Wrong Answer

Score 0

Breadth First Search: Shortest Reach > Coding

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

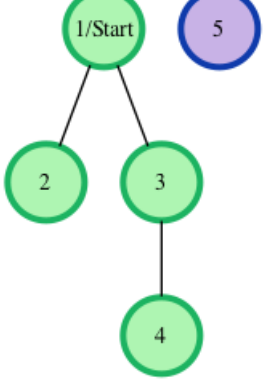
QUESTION DESCRIPTION

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
```

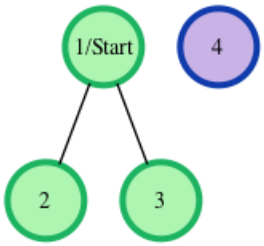
Sample Output

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

Explanation

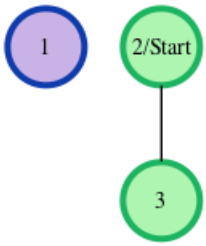
We perform the following two queries:

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: **C++**

```
/*
 * 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{
public:
    int value;
    vector<Node *> edges;
    bool seen;
    int dist;

    Node(int v) : value(v), seen(false), dist(0){
    }


    void add_edge(Node * node){
```

```
        edges.push_back(node);
    }

    void reset_seen_dist_attributes();
};

class Graph{
public:
    vector<Node *> nodes;

    void add_node(Node * node){
        nodes.push_back(node);
    }

    // constructors Destructors
    //Graph(){}
    ~Graph(){
        for(auto n: nodes){
            delete n;
        }
    }

    void reset_seen_dist_attributes();

    void print_nodes_dist();
};

void Graph::reset_seen_dist_attributes(){
    for(auto n: nodes){
        n->reset_seen_dist_attributes();
    }
}

void Node::reset_seen_dist_attributes(){
     seen = false;
     dist = 0;
}

void Graph::print_nodes_dist(){
    for(auto n: nodes){
        cout << n->dist << ", ";
    }
    cout << endl;
}

// n : number of nodes
// m : number of edges
// edges : start and end nodes for edges
// s : node to start traversals
vector<int> bfs(int n, int m, vector<vector<int>> edges, int s) {
    vector<int> output;
    Graph g;
    Node * curr(nullptr);

    // nodes creations in g.
    for(int i=1; i <= n; ++i){
        g.add_node(new Node(i));
    }
    // edges creations
    for(auto e: edges){
        g.nodes[e[0]-1]->edges.push_back(g.nodes[e[1]-1]);
    }

    // now, can implement BFS
```

```
    for(int i = 1; i <= n; ++i){
        if(i == s) continue;
        // reset nodes seen and dist attributes
        g.reset_seen_dist_attributes();
        // will start from node # s --> s-1 in indexes.
        curr = g.nodes[s-1];
        if(curr->value == i){
            output.push_back(curr->dist);
            continue;
        }
        // otherwise store all the edges in a queue
        // and set them as seen .... + increment dist
        vector<Node*> queue;
        for(auto& e: curr->edges){
            e->seen = true;
            e->dist = curr->dist + 6;
            queue.push_back(e);
        }
        //queue
        // Now, while queue not empty :
        //   now dequeue each one, check if target.
        //      if target, output.push_back(curr->dist) and continue
        //   if not target, queue edges if not seen yet
        //      (mark them as seen + increment dist)
        bool found = false;
        while(queue.size() != 0){
            //   now dequeue each one, check if target.
            curr = queue.front();
            // erase front
            queue.erase(queue.begin());
            //      if target, output.push_back(curr->dist) and continue
            if(curr->value == i){
                output.push_back(curr->dist);
                found = true;
                break;
            }
            //   if not target, queue edges if not seen yet
            for(auto& e: curr->edges){
                if(!e->seen){
                    e->seen = true;
                    e->dist = curr->dist + 6;
                    queue.push_back(e);
                }
            }
        }
        // if reached here, did not find target --> push back -1.
        if(found == false)
            output.push_back(-1);
    }  // end for
    return output;
}
```

TESTCASE	DIFFICULTY	TYPE	STATUS	SCORE	TIME TAKEN	MEMORY USED
Testcase 1	Easy	Sample case	Success	0	0.0159 sec	8.97 KB
Testcase 2	Medium	Hidden case	Wrong Answer	0	0.0194 sec	9.1 KB
Testcase 3	Medium	Hidden case	Wrong Answer	0	0.0714 sec	11.1 KB
Testcase 4	Hard	Hidden case	Wrong Answer	0	0.0176 sec	8.95 KB

Testcase 5	Hard	Hidden case	Wrong Answer	0	0.0186 sec	9.29 KB
Testcase 6	Hard	Hidden case	Wrong Answer	0	0.2303 sec	19 KB
Testcase 7	Hard	Hidden case	Wrong Answer	0	0.0325 sec	9.7 KB
Testcase 8	Easy	Sample case	Success	0	0.0198 sec	8.95 KB

No Comments

PDF generated at: 2 Nov 2021 02:33:22 UTC