Experiment: 7(a)

Student Name: Zatch **UID:**
Branch: BE CSE **Section:**
Semester: 5th **Date of Performance:**
Subject Name: AP Lab-1 **Subject Code: 22CSP-314**

1. **Title:** BFS Short Reach.

2. **Aim:** Consider an undirected graph where each edge weighs 6 units. Each of the nodes is labelled 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 *breadthfirst 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.

3. **Objective:** The objective of this experiment is to implement an efficient method for calculating the shortest paths in an undirected graph using Breadth-First Search (BFS). This involves representing the graph with an adjacency list, executing BFS from a given start node, and returning the shortest distances to all other nodes, with `-1` indicating unreachable nodes.

4. **Algorithm:**

Step1: Graph Representation:

- Create an adjacency list to represent the undirected graph from the given list of edges.

Step 2: Initialize BFS:

- Create a distance array of size n + 1 (where n is the number of nodes) initialized to -1. Set distance[startNode] to 0 to indicate the start node.
- Create a queue and enqueue the startNode.

Step 3: BFS Traversal:

- While the queue is not empty:

- Dequeue the front node currentNode.
- For each neighbor neighbor of currentNode:
  - If distance[neighbor] is -1 (indicating it has not been visited):
    - Update distance[neighbor] to distance[currentNode] + 6 (distance from the start node via the edge).
    - Enqueue neighbor.

Step 3: Return Distances:

- After BFS completes, return the distance array, which contains the shortest distances from the start node to each node.

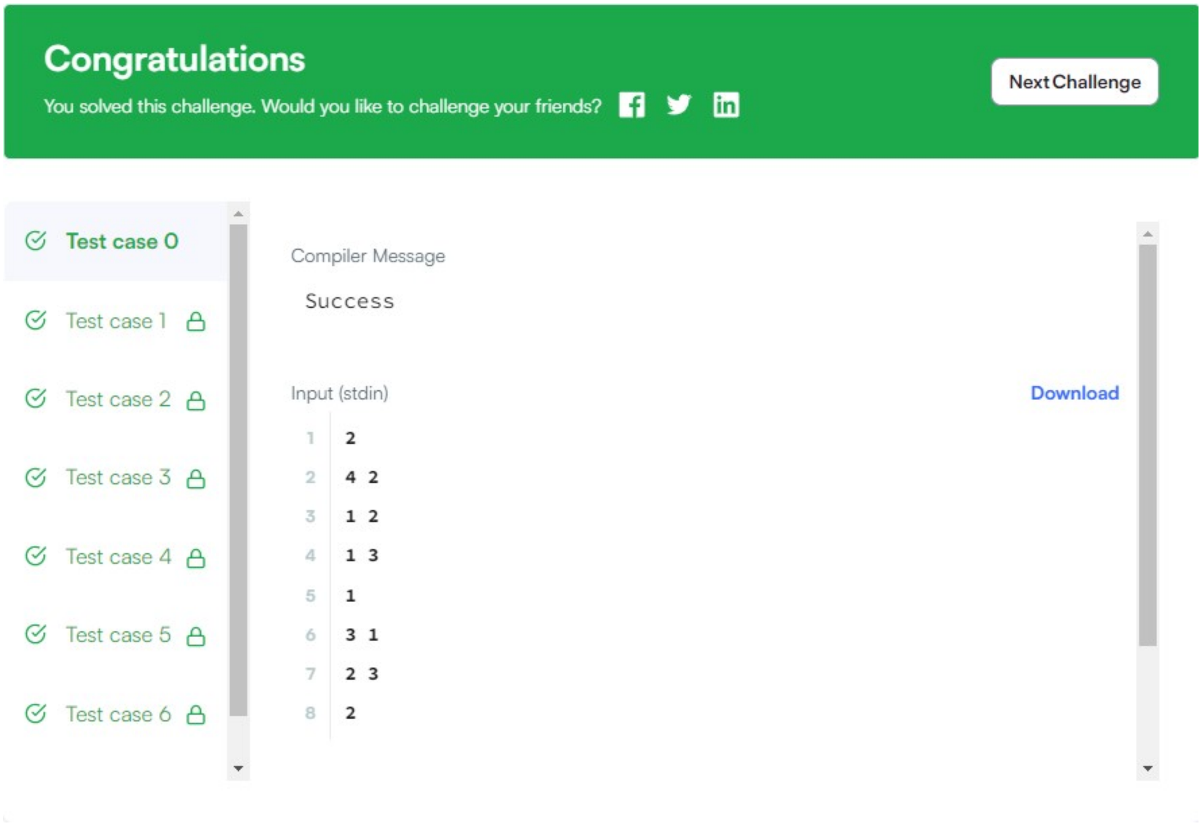
5. Code:

```
#include <bits/stdc++.h>
using namespace std;
string ltrim(const string &);
string rtrim(const string &);
vector<string> split(const string &);

vector<int> bfs(int n, int m, vector<vector<int>> edges, int s) {
   // Create an adjacency list for the graph
   vector<vector<int>> adj(n + 1); // Nodes are 1-indexed
   for (const auto &edge : edges) {
      int u = edge[0], v = edge[1];
      adj[u].push_back(v);
      adj[v].push_back(u); // As the graph is undirected
   }
   vector<int> distance(n + 1, -1);
   queue<int> q;
   distance[s] = 0;
   q.push(s);

   while (!q.empty()) {
      int node = q.front();
      q.pop();

      for (int neighbor : adj[node]) {
         if (distance[neighbor] == -1) { // If not visited
            distance[neighbor] = distance[node] + 6; // Unweighted graph: edge
 weight is 6
            q.push(neighbor);
         }
      }
   }
```

```
    vector<int> result;
    for (int i = 1; i <= n; i++) {
        if (i != s) {
            result.push_back(distance[i]);
        }
    }

    return result;
}

int main()
{
    ofstream fout(getenv("OUTPUT_PATH"));

    string q_temp;
    getline(cin, q_temp);

    int q = stoi(ltrim(rtrim(q_temp)));

    for (int q_itr = 0; q_itr < q; q_itr++) {
        string first_multiple_input_temp;
        getline(cin, first_multiple_input_temp);

        vector<string> first_multiple_input = split(rtrim(first_multiple_input_temp));

        int n = stoi(first_multiple_input[0]);
        int m = stoi(first_multiple_input[1]);

        vector<vector<int>> edges(m);

        for (int i = 0; i < m; i++) {
            edges[i].resize(2);

            string edges_row_temp_temp;
            getline(cin, edges_row_temp_temp);

            vector<string> edges_row_temp = split(rtrim(edges_row_temp_temp));

            for (int j = 0; j < 2; j++) {
                int edges_row_item = stoi(edges_row_temp[j]);

                edges[i][j] = edges_row_item;
            }
        }
        string s_temp;
        getline(cin, s_temp);
        int s = stoi(ltrim(rtrim(s_temp)));
        vector<int> result = bfs(n, m, edges, s);

        for (size_t i = 0; i < result.size(); i++) {
            fout << result[i];
            if (i != result.size() - 1) {
                fout << " ";
            }
        }
        fout << "\n";
    }
    fout.close();
    return 0;
}

string ltrim(const string &str) {
    string s(str);
    s.erase(
        s.begin(),
        find_if(s.begin(), s.end(), not1(ptr_fun<int, int>(isspace)))
    );
    return s;
}

string rtrim(const string &str) {
    string s(str);
    s.erase(
        find_if(s.rbegin(), s.rend(), not1(ptr_fun<int, int>(isspace))).base(),
        s.end()
    );
    return s;
}

vector<string> split(const string &str) {
    vector<string> tokens;

    string::size_type start = 0;
    string::size_type end = 0;

    while ((end = str.find(" ", start)) != string::npos) {
        tokens.push_back(str.substr(start, end - start));
        start = end + 1;
    }
```

```
        tokens.push_back(str.substr(start));
        return tokens;
         }
```

6. **Output:**



Congratulations
You solved this challenge. Would you like to challenge your friends?
Next Challenge

Test case 0
Test case 1
Test case 2
Test case 3
Test case 4
Test case 5
Test case 6

Compiler Message
```
Success
```
Input (stdin)
Download
```
2
4 2
1 2
1 3
1
3 1
2 3
2
```

7. **Learning outcomes:**
   1. **Graph Representation**: Understand how to represent graphs using adjacency lists for efficient traversal.
   2. **BFS Algorithm**: Learn to implement Breadth-First Search to compute shortest paths in unweighted graphs.
   3. **Queue Utilization**: Use queues to explore nodes level by level in BFS.
   4. **Edge Case Handling**: Handle unreachable nodes by marking them with a default value (e.g., `-1`).

8. **Time Complexity:** O(n+m)
9. **Space Complexity:** O(n+m)

Experiment: 7(b)

Student Name: Zatch **UID:**
Branch: BE-CSE **Section:**
Semester: 5th **Date of Performance:**
Subject Name: Advanced Lab-1 **Subject Code: 22CSP-314**

1. **Title:** Quickest Way Up.

2. **Aim**: Markov takes out his Snakes and Ladders game, stares at the board and wonders: "If I can always roll the die to whatever number I want, what would be the least number of rolls to reach the destination?"
   Rules The game is played with a cubic die of 6 faces numbered 1 to 6.
   1. Starting from square 1 , land on square 100 with the exact roll of the die. If moving the number rolled would place the player beyond square 100 , no move is made.
   2. If a player lands at the base of a ladder, the player must climb the ladder. Ladders go up only.
   3. If a player lands at the mouth of a snake, the player must go down the snake and come out through the tail. Snakes go down only.

3. **Objective**: The objective is to determine the minimum number of dice rolls needed to reach square 100, accounting for ladders that advance the player and snakes that send the player backward, while ensuring the player lands exactly on square 100.

4. **Algorithm:**

   1. **Graph Representation**: Treat the board as a graph with 100 nodes (each square representing a node). Each node is connected to the next 6 nodes (dice rolls), with additional edges for ladders and snakes.
   2. **Initialize**: Create an array `moves[]` where each index represents a square. For ladders and snakes, update `moves[i]` to point to the destination square (either the top of the ladder or the tail of the snake).
   3. **BFS Setup**: Use Breadth-First Search (BFS) to explore the shortest path. Initialize a queue and start from square 1. Maintain a distance array `dist[]` to track the minimum number of dice rolls to reach each square.
   4. **BFS Execution**:
      1. For each square, check all possible dice rolls (1 to 6).
      2. For each roll, move the player to the corresponding square and check for ladders or snakes to adjust the position.

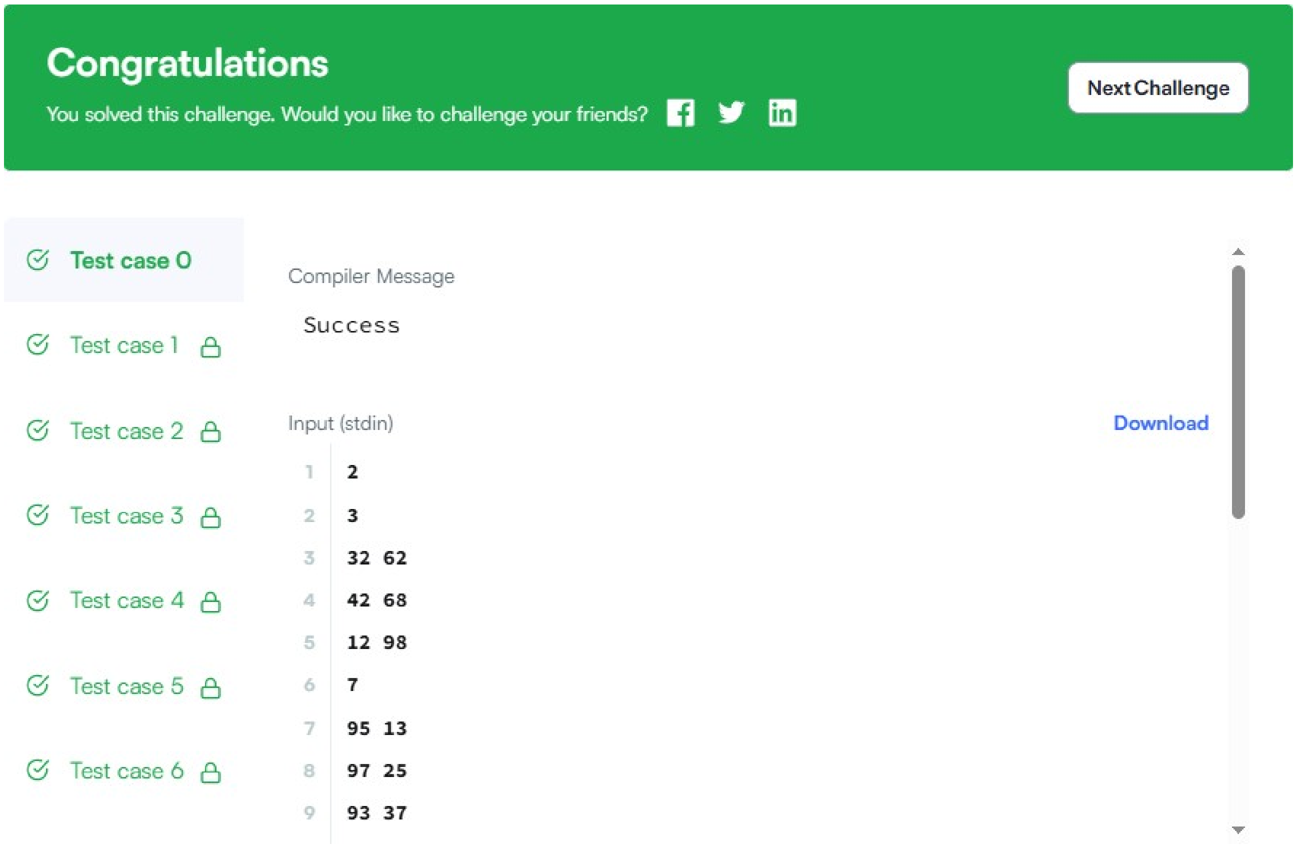
        3. If the new square hasn't been visited, update its distance and enqueue it.
    5. **Terminate**: The BFS terminates when you reach square 100, and the result is the minimum number of rolls stored in the distance array.

5. Code:

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

const int N=104;
const int INF=100000000;
int main()
{
   int t;
   cin>>t;
   for(int k=1;k<=t;++k)

   {
      vector<int> graph(N,0);
      vector<bool> mark(N,false);
      int n,m;
      cin>>n;
      for(int i=0;i<n;++i)
      {
         int a,b;
         cin>>a>>b;
         graph[a]=b;
      }

      cin>>m;
      for(int i=0;i<m;++i)
      {
         int a,b;
         cin>>a>>b;
         graph[a]=b;
      }

      queue< pair<int,int> > q;
      int ans=INF;
      q.push(make_pair(1,0));
```

```
        mark[1]=true;
        while(!q.empty())
        {
            pair<int,int> p=q.front();
            if(p.first==100)
            {
                ans=p.second;
                break;
            }
            q.pop();

            for(int i=1;i<=6;++i)
            {
                int x=p.first+i;
                if(x>100)
                    continue;

                if(mark[x]==false)
                {
                    mark[x]=true;
                    if(graph[x]==0)
                        q.push(make_pair(x,p.second+1));
                    else
                    {
                        x=graph[x];
                        mark[x]=true;
                        q.push(make_pair(x,p.second+1));

                    }
                }
            }
        }

        if(ans==INF)
            cout<<-1<<endl;
        else
            cout<<ans<<endl;
    }
     }
```

6. **Output:**



Compiler Message

```
Success
```

Input (stdin)

```
2
3
32 62
42 68
12 98
7
95 13
97 25
93 37
```

7. **Learning outcomes:**

1. Graph Representation: Learn to model a game board as a graph, where each node represents a square and edges represent possible dice rolls and game mechanics like ladders and snakes.

2. BFS Algorithm: Understand how Breadth-First Search (BFS) can be used to find the shortest path in an unweighted graph, which is essential for solving problems involving minimum steps or moves.

3. Handling Game Mechanics: Gain experience in incorporating specific game rules (e.g., ladders and snakes) into the graph traversal to adapt general algorithms to problem-specific constraints.

8. **Time Complexity:** O(1)
9. **Space Complexity:** O(1)