Experiment 7

Student Name: Gaganjot Singh **UID:** 22BCS14843
Branch: BE-CSE **Section/Group:** 22BCS-JT-802-B
Semester: 5th **Date of Performance:** 03/09/2024
Subject Name: Advanced Programming Lab-1 **Subject Code:** 22CSP-314

1. **Title (1a):** Breadth First Search: Shortest 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 *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.

3. **Objective:** To implement a breadth-first search (BFS) algorithm that determines the shortest path from a given starting node to all other nodes in an undirected graph where each edge has a uniform weight. The solution must return distances in node order, with unreachable nodes indicated by -1.

4. **Methodology:** Construct the graph using an adjacency list, then perform BFS from the starting node to explore all reachable nodes. Track distances during traversal, updating each node's distance if it is shorter than the previously known distance. Return a list of distances in node order, excluding the start node.

5. **Code:**

```
def bfs(n, m, edges, s):
    # Write your code here
    graph=defaultdict(list)
    for u, v in edges:
```

```
    graph[u].append(v)
    graph[v].append(u)

distances=[-1]*(n+1)
distances[s]=0

queue=deque([s])

while queue:
    node=queue.popleft()
    current_distance=distances[node]

    for neighbor in graph[node]:
        if distances[neighbor]==-1:
            distances[neighbor]=current_distance+6
            queue.append(neighbor)

result=[]
for i in range(1,n+1):
    if i !=s:
        result.append(distances[i])
return result
```

6. Output:

7. **Complexity:**

   Time Complexity: O(n+m)

   Space Complexity: O(n+m)

8. **Learning Outcomes:**

- Understanding of the BFS algorithm for finding the shortest path in unweighted graphs.
- Ability to implement adjacency lists for graph representation.
- Proficiency in using queues for breadth-first traversal of graph structures.
- Skill in handling graph traversal edge cases, such as unreachable nodes.
- Insight into optimizing algorithms for both time and space efficiency in large graphs.

1. **Title (1b):** Snakes and Ladders: The 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.
   - 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.
   - If a player lands at the base of a ladder, the player must climb the ladder. Ladders go up only.
   - 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:** To develop an efficient algorithm that determines the minimum number of rolls required to reach the final square in a Snakes and Ladders game, considering ladders and snakes that alter the player's position on the board. The solution aims to minimize the number of rolls using graph traversal techniques.

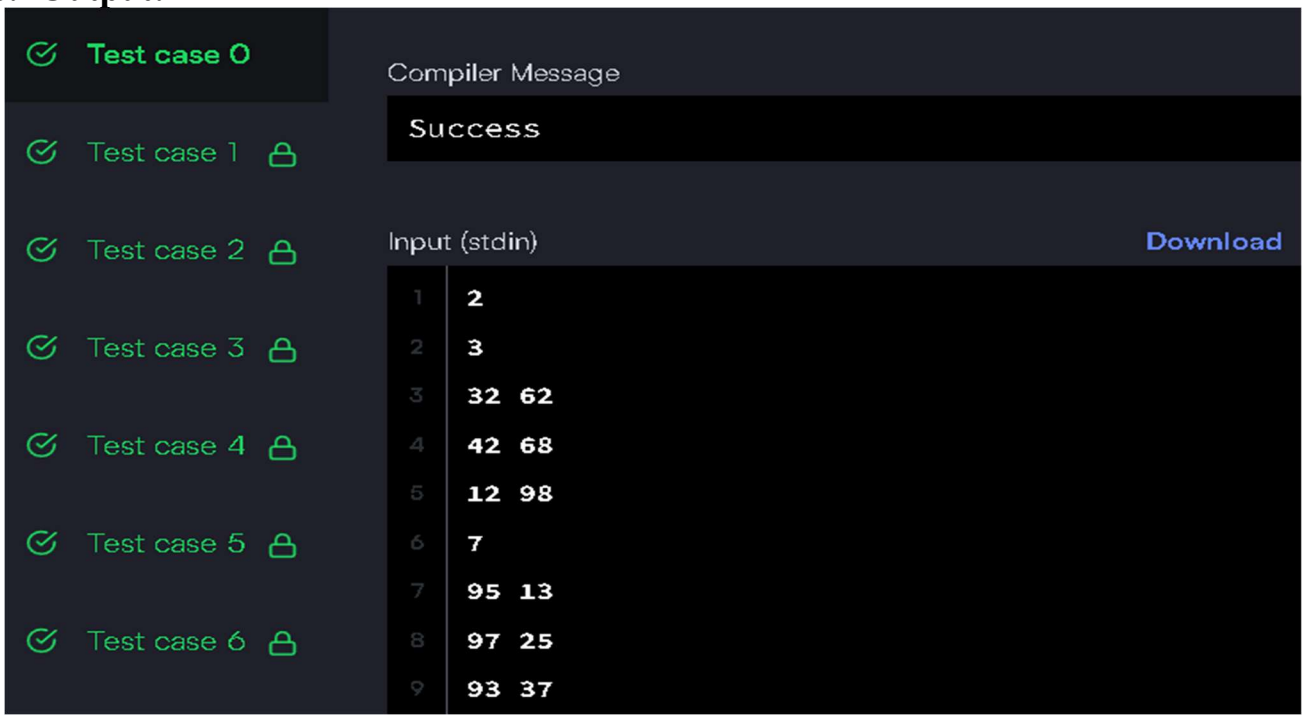
4. **Methodology:** The problem is modeled as a graph where each square represents a node and valid dice rolls represent edges. A Breadth-First Search (BFS) algorithm is implemented to explore the shortest path from the start to the destination square, accounting for the effects of ladders and snakes on the board.

5. **Code:**

```
def quickestWayUp(ladders, snakes):
    # Write your code here
    board=list(range(101))

    for start,end in ladders:
        board[start]=end
    for start, end in snakes:
        board[start]=end
```

```
queue=deque([(1,0)])
visited=set()

while queue:
    current_square, rolls=queue.popleft()

    if current_square==100:
        return rolls

    for dice_rolls in range(1,7):
        next_square=current_square+dice_rolls

        if next_square<=100:
            final_square=board[next_square]

            if final_square not in visited:
                visited.add(final_square)
                queue.append((final_square,rolls+1))
return -1
```

6. **Output:**



7. **Complexity:**
   Time Complexity: O(V+E)

   Space Complexity: O(V)

8. **Learning Outcomes:**

- Understanding how to model real-world problems as graph traversal tasks.
- Gaining proficiency in implementing the BFS algorithm to find the shortest path in unweighted graphs.
- Learning to handle edge cases, such as scenarios where specific board elements (ladders, snakes) modify the path.
- Analyzing and optimizing time and space complexity in graph-related problems.
- Applying computational thinking to design algorithms that solve board game-related challenges effectively.