Ex. 4	**SOCIAL NETWORKING RECOMMENDATION USING BFS ALGORITHM**	**MAINAK CHAUDHURI RA1911027010039 CSE - BD, N1**

AIM : Finding friends/connections using heuristic breadth first search algorithm.

Breadth First Search, being a level order traversal, would be quite efficient over Depth First Search for many business based insights, like the following:

1. Finding all the friends of all the people in the network
2. Finding all the mutual friends for a node in the network
3. Finding the shortest path between two people in the network
4. Finding the nth level friends for a person in the network

We can find the path between two people by running a BFS algorithm, starting the traversal from one person in level order until we reach the other person or in a much optimised way we can run a bi-directional BFS from both the nodes until our search meet at some point and hence we conclude the path , whereas DFS being a depth wise traversal may run through many unnecessary sub-trees, unknowing of the fact that the friend could be on the first level itself.
Moreover, in order to find friends at nth level, using BFS this could be done in much less time, as this traversal keeps account of all the nodes in each level.

Input Format
The first line of the input denotes the total number of users (n) in the social media network.
space.
Output Format
The output consists of space separated friend-ids which are present at level k from the source friend-id s. If no friend present at level k then print 0

CODE:

```
#social network bfs algorithm
from collections import deque
graph = {}
queries = []
def accept_values():
    #accept number of vertices and edges
    vertex_edge = [int(i) for  i in input().strip().split(" ")]
    #accept the edges for the undirected graph
    for i in range(vertex_edge[1]):
        edge = [int(i) for i in input().strip().split(" ")]
        try:
            graph[edge[0]].append(edge[1])
        except:
            graph[edge[0]] = [edge[1]]
        try:
            graph[edge[1]].append(edge[0])
        except:
            graph[edge[1]] = [edge[0]]
    #accepting the number of queries and the queries themselves
    number_of_queries = int(input())
    for i in range(number_of_queries):
        queries.append([int(i) for i in input().strip().split(" ")])
    #applying bfs on the (source, required distance) queries
    for query in queries:
        bfs(query)

def bfs(query):
    counter = 0
    q = deque()
    q.append(query[0])
    visited = {query[0] : True}
    distance = {query[0] : 0}
    while q:
        popped = q.popleft()
        for neighbour in graph[popped]:
            if neighbour not in visited:
                visited[neighbour] = True
                #stop looking further because we have all the nodes at the required distance,
i think
                if distance[popped] + 1 > query[1]:
                    for key in distance:
                        if distance[key] == query[1]:
                          counter +=1
                    print(counter)
                    return
                #keep going until we reach the required query distance
                else:
                    distance[neighbour] = distance[popped] + 1
                    q.append(neighbour)
    #prints 0 if there are no such nodes
    print(counter)
#calling function to accept values as per required format
accept_values()
```

OUTPUT:

```
0 61 21 32 43 64 5-1 -11 2
```

```
4 6
```