Problem Analysis (BFS)

1. Definition (BFS)

Breadth-First Search (BFS) is a graph traversal algorithm that explores nodes **level by level**. It starts from a source node, visits all its neighbors first, then moves to the neighbors' neighbors, and so on. BFS guarantees the **shortest path in an unweighted graph**.

2. Solution Approach

1. Maintain a **queue** to store nodes to visit next.
2. Keep a **distance array or visited array** to track which nodes are visited and their distances from the start.
3. Start from the source node:
    - Mark it as visited and push it into the queue.
4. While the queue is not empty:
    - Pop the front node.
    - For each unvisited neighbor, mark it visited, record the distance, and push it into the queue.
5. Stop when all nodes are visited or the target node is reached.

3. Hint (General BFS Tips)

- BFS is best for **shortest path problems** in unweighted graphs.
- Use a queue, not recursion (DFS uses recursion/stack).
- Always mark nodes as visited **when you push them into the queue**, not when you pop, to avoid revisiting.

4. Pseudocode (C++ Style)

```
vector<int> bfs(int start, int n, vector<vector<int>>& adj) {
    vector<int> dist(n, -1);
    queue<int> q;
    dist[start] = 0;
    q.push(start);

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v : adj[u]) {
            if (dist[v] == -1) {
                dist[v] = dist[u] + 1;
```

```
                q.push(v);
            }
        }
    }
    return dist;
}
```

5. Hint

- BFS is typically used for:
  - Shortest path in **unweighted graphs**.
  - Checking **connectivity**.
  - Level order traversal in **trees**.
- Always check for **graph boundaries** (0-indexed vs 1-indexed).

6. Time Complexity

- **O(V + E)** where:
  - V = number of vertices
  - E = number of edges
- Each node is visited **once**, and each edge is checked **once**.

Implementation Link:

algorithm-/BFS at main · Jannat651/algorithm-

problem1: BICOLORING

Problem Link:

Online Judge

Problem Statement

Are given an undirected graph.
Must determine whether it is **bicolorable**, meaning:

- You can color every node using **only two colors**
- No two connected (adjacent) nodes can have the **same color**

If such a coloring is possible → **BICOLORABLE.**
Otherwise → **NOT BICOLORABLE.**

This is exactly the same as checking if a graph is **bipartite**.

Pseudocode

```
read n
while n != 0:
    read e
    create graph with n empty lists

    repeat e times:
        read a, b
        add b to graph[a]
        add a to graph[b]

    create color array size n, fill with -1
    queue Q

    color[0] = 0
    push 0 into Q

    bipartite = true

    while Q not empty:
        node = Q.pop()
        for neighbor in graph[node]:
            if color[neighbor] == -1:
                color[neighbor] = 1 - color[node]
                Q.push(neighbor)
            else if color[neighbor] == color[node]:
                bipartite = false

    if bipartite:
        print "BICOLORABLE."
    else:
        print "NOT BICOLORABLE."

    read n
```

Hints for Solving the Problem

- Use **BFS** to color the graph level-by-level.
- Use an array color[]:
    - -1 → not colored yet
    - 0 and 1 → two colors
- When coloring neighbors, assign opposite color:
  color[neighbor] = 1 - color[node]
- If you ever find an edge where both ends have the **same color**, the graph is **not bicolorable**.

- The graph may be **disconnected** in general, but for this problem it always starts BFS from node 0.

Input-Output Analysis

Input

- First line: number of nodes n
- Second line: number of edges e
- Next e lines: pairs a b meaning edge between a and b
- Ends when n = 0

Output

- Print **BICOLORABLE.** if possible
- Print **NOT BICOLORABLE.** otherwise

Example

Input

3
3
0 1
1 2
2 0
0

Output

NOT BICOLORABLE.

Reason: Triangle graph → odd cycle → cannot use 2 colors.

Implementation Link:

algorithm-/BFS/bicolorlink/bicolor.cpp at main · Jannat651/algorithm-