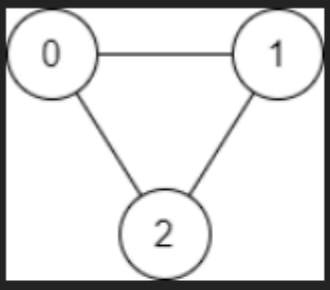
GRAPHS.

1) FIND IF PATH EXISTS (1971)

Example 1:



Input: n = 3, edges = [[0,1],[1,2],[2,0]], source = 0, destination = 2
Output: true
Explanation: There are two paths from vertex 0 to vertex 2:
- 0 → 1 → 2
- 0 → 2

Idea:

1. Build adjacency list of size `n`, add both directions because undirected.
2. Create `visited[n]`, mark `source` visited and push it into queue.
3. While queue not empty, remove a node `curr`.
4. If `curr == destination`, return true.
5. For every neighbor of `curr`, if not visited, mark visited and push into queue.
6. If BFS ends without finding destination, return false.

Traps:

1. Adj size must be `n`, not edges.length.
2. Must mark visited when adding to queue, not after removing.

3. Push neighbor value, not loop index.

4. Forgetting reverse edge breaks connectivity.

Code:

```
class Solution {

    public boolean validPath(int n, int[][] edges, int source, int destination) {

        List<List<Integer>> adjlist = new ArrayList<>();

        if(source==destination) return true;

        for (int i = 0; i < n; i++) {

            adjlist.add(new ArrayList<>());

        }

        for (int i = 0; i <edges.length; i++) {

            int u = edges[i][0];

            int v = edges[i][1];

            adjlist.get(u).add(v);

            adjlist.get(v).add(u);

        }

        boolean[] visited = new boolean[n];

        Queue<Integer> q = new LinkedList<>();

        q.add(source);

        visited[source] = true;

        while(!q.isEmpty())

        {

            int size = q.size();

            int curr = q.remove();

            for(int i=0; i<adjlist.get(curr).size(); i++)

            {

                if(adjlist.get(curr).get(i)==destination) return true;
```

```
                else if(!visited[adjlist.get(curr).get(i)])
                {
                    int nei = adjlist.get(curr).get(i);
                    visited[nei] = true;
                    q.add(nei);
                    continue;
                }
            }
        }
        return false;
    }
}
```

2) NUMBER OF PROVINCES

🌐 Number of Provinces (Find Circle Number)

🔹 Idea

- The matrix represents an **undirected graph**.
- `isConnected[i][j] == 1` means city `i` is directly connected to city `j`.
- A province = one **connected component** in the graph.

Approach:

1. Maintain a `visited[]` array.
2. Loop through every city `i`.
3. If city `i` is not visited:
    - It means we found a **new province**.
    - Increment `count`.
    - Run **BFS** starting from `i`.
4. BFS marks all cities connected (directly or indirectly) to `i` as visited.
5. Continue until all cities are processed.

👉 Number of BFS calls = Number of provinces.

⚠️ Trap Notes

- This is an **undirected graph** (matrix is symmetric).
- Don't start BFS from already visited nodes.
- Mark visited **when pushing into queue**, not after popping.
- No need to check `i != j` (diagonal values are 1 but already handled by visited).
- Works because matrix size is `n x n`.

```
class Solution {
    public int findCircleNum(int[][] isConnected) {
        Queue<Integer> q = new LinkedList<>();
        boolean[] visited = new boolean[isConnected.length];
        int count = 0;
        for(int i=0; i<isConnected.length; i++)
        {
            if(!visited[i])
            {
                q.add(i);
                visited[i] = true;
                count++;
                while(!q.isEmpty())
```

```
                {
                    int curr = q.remove();
                    for(int j=0; j<isConnected[curr].length; j++)
                    {
                        if(!visited[j] && isConnected[curr][j] == 1)
                        {
                            q.add(j);
                            visited[j] = true;
                        }
                    }
                }
            }
        }
        return count;
    }
}
```

3) NUMBER OF ISLANDS

🏝️ Number of Islands

🔹 Idea

- Traverse the grid.
- Whenever you find an unvisited `'1'`, it means a new island.
- Start BFS from that cell.
- Mark all connected `'1'` cells (4-directionally) as visited.
- Increment island count.
- Continue until full grid is processed.

Time: O(m × n)
Space: O(m × n) (visited + queue worst case)

⚠️ Trap Notes

- Grid is `char[][]` → compare with `'1'`, not `1`.
- Always check bounds first before accessing grid.
- Mark visited when adding to queue, not after removing.
- Only 4 directions (not diagonals).
- Don't forget to handle empty grid case (edge case).

```
class Solution {
    public int numIslands(char[][] grid) {
        Queue<int[]> q = new LinkedList<>();
        int m = grid.length;
        int n = grid[0].length;
        boolean [][] visited = new boolean[m][n];
        int count = 0;
        int[] dx = {-1,1,0,0};
        int[] dy = {0,0,-1,1};
        for(int i=0; i<m; i++)
        {
            for(int j=0; j<n; j++)
            {
                if(!visited[i][j] && grid[i][j]=='1')
                {
                    q.add(new int[]{i,j});
                    visited[i][j] = true;
                    count++;
                    while(!q.isEmpty())
                    {
                        int size = q.size();
                        int[] curr = q.remove();
                        for(int t=0; t<size; t++)
                        {
                            int x = curr[0];
                            int y = curr[1];
                            for(int k=0; k<4; k++)
                            {
                                int nx = x + dx[k];
                                int ny = y + dy[k];
                                if(nx < 0 || ny < 0 || nx >= m || ny >= n)
continue;
```

```
                                if(visited[nx][ny]) continue;
                                if(grid[nx][ny]!='1') continue;
                                q.add(new int[]{nx, ny});
                                visited[nx][ny] = true;
                            }
                        }
                    }
                }
            }
        }
        return count;
    }
}
```

4) FLOOD FILL

Idea:

1. Start from given cell `(sr, sc)`, note its original color.
2. If original color == new color, return image directly.
3. Use DFS or BFS to traverse 4 directions (up, down, left, right).
4. Only move to cells inside bounds AND having original color.
5. Mark cell with new color, mark as visited to avoid revisiting.

Traps:

1. Forgetting boundary check → index out of bounds.
2. Not checking original color → you spread to wrong cells.
3. Not handling `original == newColor` → infinite recursion/loop.
4. Marking visited too late → same cell added multiple times.
5. Accidentally allowing diagonal movement, only 4 directions allowed.

```java
class Solution {
    public int[][] floodFill(int[][] image, int sr, int sc, int color) {
        int original = image[sr][sc];
        image[sr][sc] = color;
        int m = image.length;
        int n = image[0].length;
        int[] dx = {0,0,1,-1};
        int[] dy = {1,-1,0,0};
        boolean[][] visited = new boolean[m][n];
        Queue<int[]> q = new LinkedList<>();
        q.add(new int[]{sr,sc});
        visited[sr][sc] = true;
        while(!q.isEmpty())
        {
            int size = q.size();
            for(int i=0; i<size; i++)
            {
                int[] curr = q.remove();
                int x = curr[0];
                int y = curr[1];
                for(int j=0; j<4;j++)
                {
                    int nx = x + dx[j];
                    int ny = y + dy[j];
                    if(nx < 0 || ny < 0 || nx >= m || ny >= n) continue;
                    if(visited[nx][ny]) continue;
                    if(image[nx][ny]!=original) continue;
                    if(image[nx][ny]==original && !visited[nx][ny])
                    {
                        image[nx][ny] = color;
                        visited[nx][ny] = true;
                        q.add(new int[]{nx,ny});
                    }
                }
            }
        }
        return image;
    }
}
```

5) SURROUNDED REGIONS - MOST IMPORTANT BOUNDARY CHECKING.

Surrounded Regions – Idea & Traps (based on your code)

✅ Idea

1. Only 'O' connected to boundary should NOT be flipped.
2. Start BFS from all boundary 'O' cells (top, bottom, left, right).
3. Mark all reachable 'O' from boundary as temporary value `'3'`.
4. After BFS, any remaining `'O'` is fully surrounded → convert to `'X'`.
5. Convert temporary `'3'` back to `'O'`.

Core thinking:
Don't try to find surrounded regions.
Instead, protect boundary-connected regions first.

⚠️ Traps

1. Forgetting to start BFS from all four boundaries.
2. Not marking before pushing to queue → duplicates in queue.
3. Forgetting boundary checks → index out of bounds.
4. Directly flipping 'O' to 'X' during BFS → breaks traversal.
5. Not restoring temporary marker ('3') back to 'O' at end.
6. Missing corner overlap handling (your visited avoids double adding, good).

```
class Solution {
    public void solve(char[][] board) {
        int m = board.length;
        int n = board[0].length;
        Queue<int[]> q = new LinkedList<>();
        int[] dx = { 0, 0, -1, 1 };
```

```
        int[] dy = { 1, -1, 0, 0 };
        boolean[][] visited = new boolean[m][n];
        //-------------------------------------------------------------

        // Top and Bottom rows
        for (int j = 0; j < n; j++) {

            // Top row (row = 0)
            if (!visited[0][j] && board[0][j] == 'O') {
                board[0][j] = '3';
                visited[0][j] = true;
                q.add(new int[] { 0, j });
            }

            // Bottom row (row = m - 1)
            if (!visited[m - 1][j] && board[m - 1][j] == 'O') {
                board[m - 1][j] = '3';
                visited[m - 1][j] = true;
                q.add(new int[] { m - 1, j });
            }
        }

        // Left and Right columns
        for (int i = 0; i < m; i++) {

            // Left column (col = 0)
            if (!visited[i][0] && board[i][0] == 'O') {
                board[i][0] = '3';
                visited[i][0] = true;
                q.add(new int[] { i, 0 });
            }

            // Right column (col = n - 1)
            if (!visited[i][n - 1] && board[i][n - 1] == 'O') {
                board[i][n - 1] = '3';
                visited[i][n - 1] = true;
                q.add(new int[] { i, n - 1 });
            }
        }

//-------------------------------------------------------------------------
---------------
        while (!q.isEmpty()) {
            int size = q.size();
            for (int i = 0; i < size; i++) {
```

```
                int[] curr = q.remove();
                int x = curr[0];
                int y = curr[1];
                for (int j = 0; j < 4; j++) {
                    int nx = x + dx[j];
                    int ny = y + dy[j];
                    if (nx < 0 || ny < 0 || nx >= m || ny >= n)
                        continue;
                    if (visited[nx][ny])
                        continue;
                    if (board[nx][ny] == 'X')
                        continue;
                    if (board[nx][ny] == 'O') {
                        board[nx][ny] = '3';
                        visited[nx][ny] = true;
                        q.add(new int[] { nx, ny });
                    }
                }
            }
        }
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < n; j++) {
                if (board[i][j] == '3') {
                    board[i][j] = 'O';
                } else if (board[i][j] == 'O') {
                    board[i][j] = 'X';
                }
            }
        }
    }
}
```

CYCLE DETECTION TYPES

LINKED LIST CYCLE

✅ Idea (your BFS + visited method)

1. Traverse the list starting from head.

2. Store every visited node reference in a collection.

3. For each node, check:
    - If `curr.next` is already in visited → cycle exists.
4. Otherwise, add `curr.next` to visited and continue.
5. If traversal reaches `null` → no cycle.

Core thinking:
Cycle means revisiting the same node reference again.

⚠️ Traps

1. Check `head == null` before adding to queue, otherwise you add null unnecessarily.
2. `visited.contains()` is $O(n)$, so total becomes $O(n^2)$ worst case.
3. Using `ArrayList` for visited is inefficient, `HashSet` is better for $O(1)$ lookup

CODE

```java
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public boolean hasCycle(ListNode head) {
        Queue<ListNode> q = new LinkedList<>();
        List <ListNode> visited = new ArrayList<>();
        q.add(head);
        visited.add(head);
        if(head==null) return false;
        while(!q.isEmpty())
        {
```

```
            int size = q.size();
            for(int i=0; i<size; i++)
            {
                ListNode curr = q.remove();
                if(curr.next==null) continue;
                if(visited.contains(curr.next)) return true;
                else
                {
                    if(curr.next == null) continue;
                    q.add(curr.next);
                    visited.add(curr.next);
                }
            }
        }
        return false;
    }
}
```

IS GRAPH BIPARTITE?

✅ Idea – Bipartite Check (Your BFS Coloring Approach)

1. A graph is bipartite if we can divide nodes into 2 groups such that no adjacent nodes have same color.
2. Use BFS and assign alternate colors (0 and 1).
3. For every unvisited node (important for disconnected graphs), start BFS.
4. Color starting node 0.
5. For each neighbor:
    - If uncolored → assign opposite color and push to queue.

- If already colored and same as current → return false.

6. If no conflict found after processing all components → return true.

Core idea:
Bipartite = no odd-length cycle = possible 2-coloring.

⚠️ Traps

1. You don't actually need `visited` separately, color array alone is enough.

Your condition

```
if(visited[i] && color[i]!=-1) continue;
```

2. is logically messy, checking `color[i] != -1` alone is enough.

3. Must handle disconnected graph, starting BFS from every uncolored node.

4. Forgetting to color before pushing to queue causes duplicate processing.

5. Conflict check must come after ensuring the neighbor is already colored.

```
class Solution {

    public boolean isBipartite(int[][] graph) {

        int n = graph.length;

        Queue<Integer> q = new LinkedList<>();
```

```
        int[] color = new int[n];
        boolean[] visited = new boolean[n];
        Arrays.fill(color, -1);
        color[0] = 0;
        for(int i=0; i<n; i++)
        {
            if(visited[i] && color[i]!=-1) continue;
            q.add(i);
            visited[i] = true;
            color[i] = 0;
            while(!q.isEmpty())
            {
                int curr = q.remove();
                for(int j=0; j<graph[curr].length; j++)
                {
                    int nei = graph[curr][j];
                    if(!visited[nei] && color[nei]==-1)
                    {
                        color[nei] = 1 - color[curr];
                        q.add(nei);
                        visited[nei] = true;
                    }
                    else if(color[nei]==color[curr]) return false;
                }
            }
        }
```

```
        return true;

    }

}
```

JUMP GAME AND JUMP GAME 2

JUMP GAME

```
class Solution {
    public boolean canJump(int[] nums) {
        Queue<Integer> q = new LinkedList<>();
        if(nums.length==1) return true;
        q.add(0);
        boolean[] visited = new boolean[nums.length];
        visited[0]=true;
        while(!q.isEmpty())
        {
            int curr = q.remove();
            for(int i=0; i<=nums[curr]; i++)
            {
                int jf = curr + i;
                if(jf >= nums.length) continue;
                if(jf==nums.length - 1) return true;
                if(!visited[jf])
                {
                    visited[jf] = true;
                    q.add(jf);
                }
            }
        }
        return false;
    }
}
```

JUMP GAME 2

```
import java.util.*;
```

```java
class Solution {
    public int jump(int[] nums) {
        int n = nums.length;
        if (n == 1) return 0;

        Queue<Integer> q = new LinkedList<>();
        boolean[] visited = new boolean[n];

        q.add(0);
        visited[0] = true;

        int jumps = 0;

        while (!q.isEmpty()) {
            int size = q.size();
            jumps++;

            for (int s = 0; s < size; s++) {
                int curr = q.remove();

                for (int i = 1; i <= nums[curr]; i++) {
                    int next = curr + i;

                    if (next >= n - 1) {
                        return jumps;
                    }

                    if (!visited[next]) {
                        visited[next] = true;
                        q.add(next);
                    }
                }
            }
        }

        return -1;
    }
}
```

🔹 Jump Game (LC 55)

✅ Idea (BFS Reachability)

1. Treat each index as a node, jumps as edges.
2. Start BFS from index 0.
3. For each index `curr`, try all jumps `curr + i` where `1 ≤ i ≤ nums[curr]`.
4. If you reach `n - 1`, return true.
5. Use `visited[]` to avoid revisiting indices.
6. If BFS ends without reaching last index → return false.

⚠️ Traps

1. Must check `next < n` before using it.
2. Mark visited before adding to queue.
3. Handle `n == 1` separately → already at last index.
4. Loop must allow full jump range.
5. Without visited → exponential reprocessing.

🔹 Jump Game II (LC 45)

✅ Idea (BFS Level = 1 Jump)

1. Each BFS level represents one jump.
2. Start from index 0, `jumps = 0`.

3. For all nodes in current level, explore reachable indices.

4. If any index reaches `n - 1`, return `jumps + 1`.

5. After finishing one level, increment `jumps`.

6. Use visited to avoid reprocessing.

Core thinking:
Minimum jumps = shortest path in unweighted graph.

⚠️ Traps

1. Must process level by level, not node by node.

2. Increment `jumps` once per level, not per node.

3. Return immediately when reaching last index.

4. Check bounds before accessing array.

5. Forgetting visited causes heavy repetition.

🧠 One-Line Difference

Jump Game → "Can I reach?"
Jump Game II → "Minimum steps to reach?"