perplexity

CODING INTERVIEW PREPARATION

Day 6: Graphs (Part 1)

Complete Study Guide with Multiple Solutions

TABLE OF CONTENTS

Page

DAY 6: GRAPHS (PART 1)

Key Concepts for Today:

- **Graph Representation** using adjacency lists and matrices
- **Depth-First Search (DFS)** for graph traversal
- **Breadth-First Search (BFS)** for shortest paths and level exploration
- **Connected Components** identification and counting
- **Cycle Detection** in directed and undirected graphs
- **Topological Sorting** for dependency resolution
- **Graph Validation** and structural properties

Problems Index:

1. **Number of Islands** (Medium)
2. **Clone Graph** (Medium)
3. **Pacific Atlantic Water Flow** (Medium)
4. **Course Schedule** (Medium)
5. **Graph Valid Tree** (Medium)
6. **Number of Connected Components in an Undirected Graph** (Medium)
7. **Longest Consecutive Sequence** (Medium)

CORE CONCEPTS EXPLANATION

Graph Fundamentals

Definition: A graph G = (V, E) consists of a set of vertices V and a set of edges E that connect pairs of vertices.

Graph Types:

- **Directed vs Undirected**: Edges have direction or are bidirectional
- **Weighted vs Unweighted**: Edges have associated costs/weights or not
- **Connected vs Disconnected**: All vertices reachable vs separate components
- **Cyclic vs Acyclic**: Contains cycles or is cycle-free (DAG)

Graph Representation

1. Adjacency Matrix:

```
int[,] adjMatrix = new int[n, n];
// adjMatrix[i][j] = 1 if edge exists from i to j
```

- **Space:** O(V^2)
- **Edge lookup:** O(1)
- **Add/remove edge:** O(1)
- **Best for:** Dense graphs, frequent edge queries

2. Adjacency List:

```
Dictionary<int, List<int>> adjList = new Dictionary<int, List<int>>();
// or
List<List<int>> adjList = new List<List<int>>();
```

- **Space:** O(V + E)
- **Edge lookup:** O(degree of vertex)
- **Add edge:** O(1)
- **Best for:** Sparse graphs, memory efficiency

3. Edge List:

```
List<(int from, int to)> edges = new List<(int, int)>();
```

- **Space:** O(E)
- **Best for:** Algorithms that process all edges

Graph Traversal Algorithms

Depth-First Search (DFS):

```
void DFS(int node, HashSet<int> visited, Dictionary<int, List<int>> graph)
{
    if (visited.Contains(node)) return;

    visited.Add(node);
    // Process current node

    foreach (int neighbor in graph[node])
    {
        DFS(neighbor, visited, graph);
    }
}
```

Breadth-First Search (BFS):

```
void BFS(int start, Dictionary<int, List<int>> graph)
{
    var visited = new HashSet<int>();
    var queue = new Queue<int>();

    queue.Enqueue(start);
    visited.Add(start);

    while (queue.Count > 0)
    {
        int node = queue.Dequeue();
        // Process current node

        foreach (int neighbor in graph[node])
        {
            if (!visited.Contains(neighbor))
            {
                visited.Add(neighbor);
                queue.Enqueue(neighbor);
            }
        }
    }
}
```

Common Graph Patterns

1. Connected Components:

- Use DFS/BFS to find all nodes reachable from a starting point
- Count separate components in disconnected graph

2. Cycle Detection:

- **Undirected:** Use DFS, cycle exists if we visit a node that's not parent
- **Directed:** Use DFS with recursion stack, cycle exists if we visit a node in current path

3. Topological Sorting:

- Order vertices such that for edge (u,v), u appears before v
- Use DFS with finish times or BFS with in-degree counting (Kahn's algorithm)

4. Shortest Path:

- **Unweighted:** BFS gives shortest path
- **Weighted:** Dijkstra's algorithm or Bellman-Ford

Why These Concepts Matter in Interviews

Problem-Solving Versatility:

- Graphs model many real-world relationships and dependencies
- Many problems can be reduced to graph traversal or analysis
- Tests understanding of fundamental algorithms and data structures

Algorithm Design Skills:

- **State space exploration**: DFS/BFS for finding solutions
- **Optimization problems**: Finding shortest paths, minimum spanning trees
- **Constraint satisfaction**: Topological sorting, bipartite matching

Real-world Applications:

- **Social Networks**: Friend relationships, influence propagation
- **Transportation**: Route planning, network optimization
- **Dependencies**: Build systems, course prerequisites, task scheduling
- **Web Crawling**: Page discovery, link analysis
- **Distributed Systems**: Network topology, failure detection

PROBLEM 1: NUMBER OF ISLANDS

Problem Statement:

Given an `m x n` 2D binary grid `grid` which represents a map of `'1'`s (land) and `'0'`s (water), return the number of islands.

An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Test Cases:

```
Example 1:
Input: grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
Output: 1
Explanation: All connected '1's form one island

Example 2:
Input: grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
Output: 3
Explanation: Three separate islands exist

Example 3:
Input: grid = [
  ["0","0","0","0","0"],
  ["0","0","0","0","0"],
  ["0","0","0","0","0"],
  ["0","0","0","0","0"]
]
Output: 0
Explanation: No land exists, so no islands
```

Interview Questions & Answers:

1. **Q:** "What defines 'adjacent' cells in this problem?"
   A: Adjacent means horizontally or vertically connected (4-directional). Diagonal connections don't count unless specified.
2. **Q:** "Can you solve this without modifying the original grid?"
   A: Yes, use a separate visited array or set to track visited cells, but modifying the grid in-place is more space-efficient.
3. **Q:** "How do you handle edge cases like empty grid or all water?"
   A: Empty grid returns 0, all water returns 0. Always validate grid dimensions before processing.
4. **Q:** "What if the problem allowed diagonal connections (8-directional)?"
   A: Same algorithm, just add diagonal directions to the directions array: `[(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)]`.
5. **Q:** "Could you use Union-Find for this problem?"
   A: Yes, Union-Find (Disjoint Set Union) is another approach where you union adjacent land cells and count distinct components.

Approach 1: Brute Force - Check All Cells Multiple Times

Idea: For each unvisited land cell, perform exhaustive search to mark all connected land cells, leading to redundant work.

Time Complexity: O((m×n)²) - potentially scan entire grid for each cell
Space Complexity: O(m×n) - visited array

```
using System;
using System.Collections.Generic;

public class Solution
{
    public int NumIslands(char[][] grid)
    {
        if (grid == null || grid.Length == 0) return 0;

        int m = grid.Length, n = grid[0].Length;
        var visited = new bool[m, n];
        int islands = 0;

        // Check each cell in the grid
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < n; j++)
            {
                if (grid[i][j] == '1' && !visited[i, j])
                {
                    // Found new island, explore all connected land
                    ExploreIsland(grid, visited, i, j, m, n);
                    islands++;
                }
            }
        }

        return islands;
    }

    private void ExploreIsland(char[][] grid, bool[,] visited, int row, int col, int m, i
    {
        // Inefficient: checking bounds and conditions multiple times
        if (row < 0 || row >= m || col < 0 || col >= n ||
            visited[row, col] || grid[row][col] == '0')
        {
            return;
        }

        visited[row, col] = true;

        // Explore all 4 directions
        ExploreIsland(grid, visited, row - 1, col, m, n); // up
        ExploreIsland(grid, visited, row + 1, col, m, n); // down
        ExploreIsland(grid, visited, row, col - 1, m, n); // left
        ExploreIsland(grid, visited, row, col + 1, m, n); // right
    }
}
```

Approach 2: DFS with In-Place Modification

Explanation: Use DFS to explore islands but modify the original grid to mark visited cells, eliminating need for separate visited array.

Time Complexity: O(m×n) - visit each cell at most once
Space Complexity: O(min(m,n)) - recursion stack for DFS

```
using System;

public class Solution
{
    public int NumIslands(char[][] grid)
    {
        if (grid == null || grid.Length == 0) return 0;

        int m = grid.Length, n = grid[0].Length;
        int islands = 0;

        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < n; j++)
            {
                if (grid[i][j] == '1')
                {
                    islands++;
                    DFS(grid, i, j, m, n);
                }
            }
        }

        return islands;
    }

    private void DFS(char[][] grid, int row, int col, int m, int n)
    {
        // Base case: out of bounds or water/visited
        if (row < 0 || row >= m || col < 0 || col >= n || grid[row][col] != '1')
        {
            return;
        }

        // Mark current cell as visited by changing it to '0'
        grid[row][col] = '0';

        // Explore all 4 directions
        DFS(grid, row - 1, col, m, n);
        DFS(grid, row + 1, col, m, n);
        DFS(grid, row, col - 1, m, n);
        DFS(grid, row, col + 1, m, n);
    }
}
```

Approach 3: Optimal - BFS with Queue

Detailed Reasoning: Use BFS to explore islands level by level. This approach is optimal for cases where recursion depth might be an issue (very large connected components), and it's easier to understand the exploration pattern.

Time Complexity: O(m×n) - visit each cell at most once
Space Complexity: O(min(m,n)) - queue size in worst case

```
using System;
using System.Collections.Generic;

public class Solution
{
    /// <summary>
    /// Counts number of islands using BFS traversal
    /// Time: O(m*n), Space: O(min(m,n))
    /// </summary>
    public int NumIslands(char[][] grid)
    {
        // Input validation
        if (grid == null || grid.Length == 0 || grid[0].Length == 0) return 0;

        int m = grid.Length;
        int n = grid[0].Length;
        int islands = 0;

        // Directions for 4-way movement (up, down, left, right)
        int[,] directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

        // Scan entire grid
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < n; j++)
            {
                // Found unvisited land - start BFS to explore entire island
                if (grid[i][j] == '1')
                {
                    islands++;
                    BFS(grid, i, j, m, n, directions);
```

```
                }
            }
        }

        return islands;
    }

    /// <summary>
    /// BFS to mark all connected land cells as visited
    /// </summary>
    private void BFS(char[][] grid, int startRow, int startCol, int m, int n, int[,] dire
    {
        var queue = new Queue<(int row, int col)>();
        queue.Enqueue((startRow, startCol));
        grid[startRow][startCol] = '0'; // Mark as visited

        while (queue.Count > 0)
        {
            var (row, col) = queue.Dequeue();

            // Explore all 4 adjacent cells
            for (int d = 0; d < 4; d++)
            {
                int newRow = row + directions[d, 0];
                int newCol = col + directions[d, 1];

                // Check bounds and if cell is unvisited land
                if (newRow >= 0 && newRow < m && newCol >= 0 && newCol < n &&
                    grid[newRow][newCol] == '1')
                {
                    grid[newRow][newCol] = '0'; // Mark as visited
                    queue.Enqueue((newRow, newCol));
                }
            }
        }
    }
}
```

Key Takeaways:

- **Connected components**: Each island is a connected component in graph
- **In-place modification**: Mark visited cells by changing grid values
- **4-directional traversal**: Use direction arrays for cleaner neighbor exploration

PROBLEM 2: CLONE GRAPH

Problem Statement:

Given a reference of a node in a connected undirected graph, return a deep copy (clone) of the graph. Each node in the graph contains a value (`int`) and a list (`List[Node]`) of its neighbors.

Test Cases:

```
Example 1:
Input: adjList = [[2,4],[1,3],[2,4],[1,3]]
Output: [[2,4],[1,3],[2,4],[1,3]]
Explanation: 4-node graph where:
- Node 1 connects to nodes 2 and 4
- Node 2 connects to nodes 1 and 3
- Node 3 connects to nodes 2 and 4
- Node 4 connects to nodes 1 and 3

Example 2:
Input: adjList = [[]]
Output: [[]]
Explanation: Single node with no neighbors

Example 3:
Input: adjList = []
Output: []
Explanation: Empty graph (null input)
```

Interview Questions & Answers:

1. **Q:** "What does 'deep copy' mean for a graph?"
   A: Create entirely new node objects with same values and connections, but no shared references with original graph.
2. **Q:** "How do you handle cycles in the graph during cloning?"
   A: Use a HashMap to track already cloned nodes. If we encounter a node we've already cloned, return the existing clone instead of creating new one.
3. **Q:** "Can you use both DFS and BFS for this problem?"
   A: Yes, both work. DFS is more intuitive with recursion, BFS uses explicit queue. Choice depends on graph structure and stack depth concerns.
4. **Q:** "What if the graph is disconnected?"
   A: Problem states we're given a reference to a node in a connected graph, so all reachable nodes will be cloned.
5. **Q:** "How do you ensure the cloned graph has the same structure?"
   A: Maintain mapping from original nodes to cloned nodes, and use this mapping to set up neighbor relationships in cloned graph.

Approach 1: DFS with HashMap (Recursive)

Idea: Use DFS recursion with HashMap to track original→clone mapping and handle cycles.

Time Complexity: O(V + E) - visit each node and edge once
Space Complexity: O(V) - HashMap and recursion stack

```
using System;
using System.Collections.Generic;
```

```
// Definition for a Node
public class Node
{
    public int val;
    public IList<Node> neighbors;

    public Node()
    {
        val = 0;
        neighbors = new List<Node>();
    }

    public Node(int _val)
    {
        val = _val;
        neighbors = new List<Node>();
    }

    public Node(int _val, List<Node> _neighbors)
    {
        val = _val;
        neighbors = _neighbors;
    }
}

public class Solution
{
    private Dictionary<Node, Node> cloneMap = new Dictionary<Node, Node>();

    public Node CloneGraph(Node node)
    {
        if (node == null) return null;

        cloneMap.Clear(); // Reset for each call
        return DFS(node);
    }

    private Node DFS(Node node)
    {
        // If already cloned, return existing clone
        if (cloneMap.ContainsKey(node))
        {
            return cloneMap[node];
        }

        // Create new clone node
        Node clone = new Node(node.val);
        cloneMap[node] = clone;

        // Clone all neighbors recursively
        foreach (Node neighbor in node.neighbors)
        {
            clone.neighbors.Add(DFS(neighbor));
        }
```

```
        return clone;
    }
}
```

Approach 2: BFS with HashMap

Explanation: Use BFS with queue to clone graph level by level, maintaining HashMap for original→clone mapping.

Time Complexity: O(V + E) - visit each node and edge once
Space Complexity: O(V) - HashMap and queue storage

```
using System;
using System.Collections.Generic;

public class Solution
{
    public Node CloneGraph(Node node)
    {
        if (node == null) return null;

        var cloneMap = new Dictionary<Node, Node>();
        var queue = new Queue<Node>();

        // Start with the given node
        Node clone = new Node(node.val);
        cloneMap[node] = clone;
        queue.Enqueue(node);

        while (queue.Count > 0)
        {
            Node current = queue.Dequeue();

            // Process each neighbor
            foreach (Node neighbor in current.neighbors)
            {
                if (!cloneMap.ContainsKey(neighbor))
                {
                    // Create clone for new neighbor
                    cloneMap[neighbor] = new Node(neighbor.val);
                    queue.Enqueue(neighbor);
                }

                // Add cloned neighbor to current clone's neighbor list
                cloneMap[current].neighbors.Add(cloneMap[neighbor]);
            }
        }

        return clone;
    }
}
```

Approach 3: Optimal - DFS with Cleaner Structure

Detailed Reasoning: Use DFS but with cleaner separation of concerns - one method handles cloning logic, recursive calls handle graph traversal. This approach is optimal because it's intuitive, handles cycles correctly, and has minimal overhead.

Time Complexity: O(V + E) - visit each node and edge exactly once
Space Complexity: O(V) - HashMap for node mapping plus recursion stack

```
using System;
using System.Collections.Generic;

public class Solution
{
    /// <summary>
    /// Clones undirected graph using DFS with HashMap for cycle handling
    /// Time: O(V + E), Space: O(V)
    /// </summary>
    public Node CloneGraph(Node node)
    {
        // Handle null input
        if (node == null) return null;

        // Map to store original -> clone mapping
        var visited = new Dictionary<Node, Node>();

        return CloneHelper(node, visited);
    }

    /// <summary>
    /// Recursive helper to clone graph nodes and establish connections
    /// </summary>
    private Node CloneHelper(Node original, Dictionary<Node, Node> visited)
    {
        // If we've already cloned this node, return the clone
        if (visited.ContainsKey(original))
        {
            return visited[original];
        }

        // Create new clone node with same value
        Node clone = new Node(original.val);
        visited[original] = clone;

        // Recursively clone all neighbors and add to clone's neighbor list
        foreach (Node neighbor in original.neighbors)
        {
            clone.neighbors.Add(CloneHelper(neighbor, visited));
        }

        return clone;
    }
}
```

Key Takeaways:

- **Cycle handling**: Use HashMap to avoid infinite recursion in cyclic graphs
- **Deep copy principle**: Create new objects, don't share references
- **Graph traversal**: DFS/BFS both work, choice depends on preference and constraints

PROBLEM 3: PACIFIC ATLANTIC WATER FLOW

Problem Statement:

There is an `m x n` rectangular island that borders both the Pacific Ocean and Atlantic Ocean. The Pacific Ocean touches the island's left and top edges, and the Atlantic Ocean touches the island's right and bottom edges.

The island is partitioned into a grid of square cells. You are given an `m x n` integer matrix `heights` where `heights[r][c]` represents the height above sea level of the cell at coordinate `(r, c)`.

The island receives a lot of rain, and the rain water can flow to neighboring cells directly north, south, east, and west if the neighboring cell's height is less than or equal to the current cell's height. Water can flow from any cell adjacent to an ocean into that ocean.

Return a 2D list of grid coordinates where water can flow to both the Pacific and Atlantic oceans.

Test Cases:

```
Example 1:
Input: heights = [[1,2,2,3,5],[3,2,3,4,4],[2,4,5,3,1],[6,7,1,4,5],[5,1,1,2,4]]
Output: [[0,4],[1,3],[1,4],[2,2],[3,0],[3,1],[4,0]]
Explanation: Water from these cells can reach both oceans

Example 2:
Input: heights = [[1]]
Output: [[0,0]]
Explanation: Single cell touches both oceans

Example 3:
Input: heights = [[1,2,3],[4,5,6],[7,8,9]]
Output: [[2,2]]
Explanation: Only bottom-right cell can reach both oceans
```

Interview Questions & Answers:

1. **Q:** "Why start from ocean edges instead of checking each cell individually?"
   A: Starting from edges and working inward is more efficient. We can mark all cells reachable from each ocean in two passes, then find intersection.
2. **Q:** "How do you handle the water flow direction?"
   A: Water flows from high to low, but we traverse in reverse - from oceans upward to higher elevations. This way we find all cells that can eventually drain to each ocean.
3. **Q:** "What if we used BFS instead of DFS?"
   A: BFS works equally well. DFS is often chosen for its simplicity in implementation, but BFS can be better for very deep recursions.
4. **Q:** "How do you optimize for memory usage?"
   A: Use bit manipulation to store both ocean reachability in single integer per cell, or process one ocean at a time and store only the intersection.
5. **Q:** "What if the problem had diagonal flow?"
   A: Same approach, just add diagonal directions to the directions array and modify the traversal accordingly.

Approach 1: Brute Force - Check Each Cell

Idea: For each cell, perform DFS/BFS to see if it can reach both Pacific and Atlantic oceans.

Time Complexity: O(m^2n^2) - potentially explore entire grid for each cell
Space Complexity: O(mn) - visited arrays for each exploration

```
using System;
using System.Collections.Generic;

public class Solution
{
    public IList<IList<int>> PacificAtlantic(int[][] heights)
    {
        var result = new List<IList<int>>();
        if (heights == null || heights.Length == 0) return result;

        int m = heights.Length, n = heights[0].Length;
        int[,] directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

        // Check each cell individually
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < n; j++)
            {
                if (CanReachPacific(heights, i, j, directions) &&
                    CanReachAtlantic(heights, i, j, directions))
                {
                    result.Add(new List<int> {i, j});
                }
            }
        }

        return result;
    }

    private bool CanReachPacific(int[][] heights, int row, int col, int[,] directions)
    {
        int m = heights.Length, n = heights[0].Length;
        var visited = new bool[m, n];
        return DFS(heights, row, col, visited, directions, "pacific");
    }

    private bool CanReachAtlantic(int[][] heights, int row, int col, int[,] directions)
    {
        int m = heights.Length, n = heights[0].Length;
        var visited = new bool[m, n];
        return DFS(heights, row, col, visited, directions, "atlantic");
    }

    private bool DFS(int[][] heights, int row, int col, bool[,] visited, int[,] direction
    {
        int m = heights.Length, n = heights[0].Length;

        // Check if reached ocean
        if (ocean == "pacific" && (row == 0 || col == 0)) return true;
        if (ocean == "atlantic" && (row == m - 1 || col == n - 1)) return true;

        visited[row, col] = true;

        for (int d = 0; d < 4; d++)
        {
            int newRow = row + directions[d, 0];
            int newCol = col + directions[d, 1];

            if (newRow >= 0 && newRow < m && newCol >= 0 && newCol < n &&
                !visited[newRow, newCol] && heights[newRow][newCol] <= heights[row][col])
            {
                if (DFS(heights, newRow, newCol, visited, directions, ocean))
                {
                    return true;
                }
            }
        }

        return false;
    }
}
```

Approach 2: Reverse DFS from Ocean Edges

Explanation: Start DFS from all ocean edges simultaneously, mark cells reachable from each ocean, then find intersection.

Time Complexity: O(mn) - each cell visited at most twice
Space Complexity: O(mn) - two reachability arrays

```
using System;
using System.Collections.Generic;

public class Solution
{
    public IList<IList<int>> PacificAtlantic(int[][] heights)
    {
        var result = new List<IList<int>>();
        if (heights == null || heights.Length == 0) return result;
```

```
        int m = heights.Length, n = heights[0].Length;
        bool[,] pacificReach = new bool[m, n];
        bool[,] atlanticReach = new bool[m, n];

        // Start DFS from Pacific edges (top and left)
        for (int i = 0; i < m; i++)
        {
            DFS(heights, i, 0, pacificReach);  // Left edge
        }
        for (int j = 0; j < n; j++)
        {
            DFS(heights, 0, j, pacificReach);  // Top edge
        }

        // Start DFS from Atlantic edges (bottom and right)
        for (int i = 0; i < m; i++)
        {
            DFS(heights, i, n - 1, atlanticReach);  // Right edge
        }
        for (int j = 0; j < n; j++)
        {
            DFS(heights, m - 1, j, atlanticReach);  // Bottom edge
        }

        // Find cells reachable from both oceans
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < n; j++)
            {
                if (pacificReach[i, j] && atlanticReach[i, j])
                {
                    result.Add(new List<int> {i, j});
                }
            }
        }

        return result;
    }

    private void DFS(int[][] heights, int row, int col, bool[,] reachable)
    {
        reachable[row, col] = true;
        int m = heights.Length, n = heights[0].Length;
        int[,] directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

        for (int d = 0; d < 4; d++)
        {
            int newRow = row + directions[d, 0];
            int newCol = col + directions[d, 1];

            if (newRow >= 0 && newRow < m && newCol >= 0 && newCol < n &&
                !reachable[newRow, newCol] && heights[newRow][newCol] >= heights[row][col
            {
                DFS(heights, newRow, newCol, reachable);
            }
```

```
            }
        }
    }
```

Approach 3: Optimal - BFS from Ocean Edges

Detailed Reasoning: Use BFS instead of DFS to traverse from ocean edges. This approach is optimal because it processes cells level by level, which can be more predictable for memory usage and avoids deep recursion stack issues.

Time Complexity: O(mn) - each cell visited at most twice (once for each ocean)
Space Complexity: O(mn) - reachability arrays and queue storage

```
using System;
using System.Collections.Generic;

public class Solution
{
    /// <summary>
    /// Finds cells that can drain to both oceans using BFS from ocean edges
    /// Time: O(mn), Space: O(mn)
    /// </summary>
    public IList<IList<int>> PacificAtlantic(int[][] heights)
    {
        var result = new List<IList<int>>();
        if (heights == null || heights.Length == 0 || heights[0].Length == 0) return resu

        int m = heights.Length, n = heights[0].Length;
        bool[,] pacificReach = new bool[m, n];
        bool[,] atlanticReach = new bool[m, n];

        var pacificQueue = new Queue<(int, int)>();
        var atlanticQueue = new Queue<(int, int)>();

        // Add Pacific ocean edges to queue
        for (int i = 0; i < m; i++)
        {
            pacificQueue.Enqueue((i, 0));      // Left edge
            pacificReach[i, 0] = true;

            atlanticQueue.Enqueue((i, n - 1)); // Right edge
            atlanticReach[i, n - 1] = true;
        }

        for (int j = 0; j < n; j++)
        {
            pacificQueue.Enqueue((0, j));      // Top edge
            pacificReach[0, j] = true;

            atlanticQueue.Enqueue((m - 1, j)); // Bottom edge
            atlanticReach[m - 1, j] = true;
        }

        // BFS from Pacific edges
```

```
        BFS(heights, pacificQueue, pacificReach);

        // BFS from Atlantic edges
        BFS(heights, atlanticQueue, atlanticReach);

        // Find intersection - cells reachable from both oceans
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < n; j++)
            {
                if (pacificReach[i, j] && atlanticReach[i, j])
                {
                    result.Add(new List<int> {i, j});
                }
            }
        }

        return result;
    }

    /// <summary>
    /// BFS to mark all cells reachable from starting ocean edges
    /// </summary>
    private void BFS(int[][] heights, Queue<(int, int)> queue, bool[,] reachable)
    {
        int m = heights.Length, n = heights[0].Length;
        int[,] directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

        while (queue.Count > 0)
        {
            var (row, col) = queue.Dequeue();

            // Check all 4 directions
            for (int d = 0; d < 4; d++)
            {
                int newRow = row + directions[d, 0];
                int newCol = col + directions[d, 1];

                // Check bounds, not visited, and water can flow (height >= current)
                if (newRow >= 0 && newRow < m && newCol >= 0 && newCol < n &&
                    !reachable[newRow, newCol] &&
                    heights[newRow][newCol] >= heights[row][col])
                {
                    reachable[newRow, newCol] = true;
                    queue.Enqueue((newRow, newCol));
                }
            }
        }
    }
}
```

Key Takeaways:

- **Reverse thinking**: Start from destination (oceans) and work backward to sources
- **Two-pass approach**: Mark reachability for each ocean separately, then find intersection
- **Flow direction**: Water flows downhill, but we traverse uphill from ocean edges

PROBLEM 4: COURSE SCHEDULE

Problem Statement:

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [ai, bi]` indicates that you must take course `bi` first if you want to take course `ai`.

For example, the pair ``, indicates that to take course `0` you have to first take course `1`.

Return `true` if you can finish all courses. Otherwise, return `false`.

Test Cases:

```
Example 1:
Input: numCourses = 2, prerequisites = [[1,0]]
Output: true
Explanation: Take course 0 first, then course 1

Example 2:
Input: numCourses = 2, prerequisites = [[1,0],[0,1]]
Output: false
Explanation: Circular dependency - course 0 needs 1, course 1 needs 0

Example 3:
Input: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]
Output: true
Explanation: Valid order exists: 0 → 1 → 3, 0 → 2 → 3
```

Interview Questions & Answers:

1. **Q:** "What makes this a graph problem?"
   A: Courses are vertices, prerequisites are directed edges. The problem asks if we can find a topological ordering of all vertices (no cycles).
2. **Q:** "How do you detect cycles in a directed graph?"
   A: Use DFS with three states: unvisited, visiting (in current path), visited. If we encounter a 'visiting' node, we found a cycle.
3. **Q:** "Can you solve this without building the full adjacency list?"
   A: For cycle detection, we need the graph structure. But for simple cases, you could use Union-Find, though it's less efficient for directed graphs.

4. **Q:** "What if some courses have no prerequisites?"
   A: They can be taken anytime and don't affect the topological ordering. All isolated nodes are valid in any topological sort.
5. **Q:** "How would you return the actual course order instead of just true/false?"
   A: That's "Course Schedule II" - use Kahn's algorithm or DFS to build the topological order while checking for cycles.

Approach 1: DFS with Simple Cycle Detection

Idea: Use DFS for each course, maintaining visited set to detect cycles during traversal.

Time Complexity: O(V + E)² - potentially restart DFS for each vertex
Space Complexity: O(V + E) - graph storage and recursion stack

```
using System;
using System.Collections.Generic;

public class Solution
{
    public bool CanFinish(int numCourses, int[][] prerequisites)
    {
        // Build adjacency list
        var graph = new Dictionary<int, List<int>>();
        for (int i = 0; i < numCourses; i++)
        {
            graph[i] = new List<int>();
        }

        foreach (var prerequisite in prerequisites)
        {
            int course = prerequisite[0];
            int prereq = prerequisite[1];
            graph[course].Add(prereq);
        }

        // Check each course for cycles
        for (int i = 0; i < numCourses; i++)
        {
            var visited = new HashSet<int>();
            if (HasCycle(graph, i, visited))
            {
                return false;
            }
        }

        return true;
    }

    private bool HasCycle(Dictionary<int, List<int>> graph, int course, HashSet<int> visi
    {
        if (visited.Contains(course)) return true; // Cycle detected

        visited.Add(course);

        foreach (int prereq in graph[course])
        {
            if (HasCycle(graph, prereq, visited))
            {
                return true;
            }
        }

        visited.Remove(course); // Backtrack
        return false;
    }
}
```

Approach 2: Kahn's Algorithm (BFS with In-degree)

Explanation: Use Kahn's algorithm for topological sorting. If we can process all vertices, no cycles exist.

Time Complexity: O(V + E) - process each vertex and edge once
Space Complexity: O(V + E) - graph storage and queue

```
using System;
using System.Collections.Generic;

public class Solution
{
    public bool CanFinish(int numCourses, int[][] prerequisites)
    {
        // Build graph and calculate in-degrees
        var graph = new List<List<int>>(numCourses);
        var inDegree = new int[numCourses];

        for (int i = 0; i < numCourses; i++)
        {
            graph.Add(new List<int>());
        }

        foreach (var prerequisite in prerequisites)
        {
            int course = prerequisite[0];
            int prereq = prerequisite[1];
            graph[prereq].Add(course);
            inDegree[course]++;
        }

        // Start with courses that have no prerequisites
        var queue = new Queue<int>();
        for (int i = 0; i < numCourses; i++)
        {
            if (inDegree[i] == 0)
            {
                queue.Enqueue(i);
            }
        }

        int processedCourses = 0;

        while (queue.Count > 0)
        {
            int course = queue.Dequeue();
            processedCourses++;

            // Remove this course and update in-degrees
            foreach (int dependent in graph[course])
            {
                inDegree[dependent]--;
                if (inDegree[dependent] == 0)
                {
                    queue.Enqueue(dependent);
                }
            }
        }

        return processedCourses == numCourses;
    }
}
```

Approach 3: Optimal - DFS with Three States

Detailed Reasoning: Use DFS with three states (unvisited, visiting, visited) for optimal cycle detection. This approach correctly handles all graph structures and is the most intuitive for directed graph cycle detection.

Time Complexity: O(V + E) - visit each vertex and edge at most once
Space Complexity: O(V + E) - graph storage and recursion stack

```
using System;
using System.Collections.Generic;

public class Solution
{
    /// <summary>
    /// Determines if all courses can be finished using DFS cycle detection
    /// Time: O(V + E), Space: O(V + E)
    /// </summary>
    public bool CanFinish(int numCourses, int[][] prerequisites)
    {
        // Build adjacency list representation of the graph
        var graph = new List<List<int>>(numCourses);
        for (int i = 0; i < numCourses; i++)
        {
            graph.Add(new List<int>());
        }

        // Add edges: course → prerequisite
        foreach (var prerequisite in prerequisites)
        {
            int course = prerequisite[0];
            int prereq = prerequisite[1];
            graph[course].Add(prereq);
        }

        // Three states: 0 = unvisited, 1 = visiting, 2 = visited
        var state = new int[numCourses];

        // Check each course for cycles
        for (int i = 0; i < numCourses; i++)
        {
            if (state[i] == 0 && HasCycleDFS(graph, i, state))
            {
                return false; // Cycle detected
            }
        }

        return true; // No cycles found
    }

    /// <summary>
    /// DFS to detect cycles using three-state approach
    /// </summary>
    /// <param name="graph">Adjacency list representation</param>
    /// <param name="course">Current course being processed</param>
    /// <param name="state">State array: 0=unvisited, 1=visiting, 2=visited</param>
    /// <returns>True if cycle is detected</returns>
    private bool HasCycleDFS(List<List<int>> graph, int course, int[] state)
    {
        if (state[course] == 1) return true;  // Cycle: found node in current path
        if (state[course] == 2) return false; // Already fully processed

        // Mark as visiting (in current DFS path)
        state[course] = 1;

        // Check all prerequisites recursively
        foreach (int prereq in graph[course])
        {
            if (HasCycleDFS(graph, prereq, state))
            {
                return true;
            }
        }

        // Mark as visited (fully processed)
        state[course] = 2;
        return false;
    }
}
```

Key Takeaways:

- **Cycle detection**: Course scheduling is equivalent to detecting cycles in directed graph
- **Three-state DFS**: Most robust method for directed graph cycle detection
- **Topological sorting**: Kahn's algorithm provides alternative approach with in-degree counting

PROBLEM 5: GRAPH VALID TREE

Problem Statement:

You have a graph of `n` nodes labeled from `0` to `n - 1`. You are given an integer `n` and a list of `edges` where `edges[i] = [ai, bi]` indicates that there is an undirected edge between nodes `ai` and `bi` in the graph.

Return `true` if the edges of the given graph make up a valid tree, and `false` otherwise.

Test Cases:

```
Example 1:
Input: n = 5, edges = [[0,1],[0,2],[0,3],[1,4]]
Output: true
Explanation: Valid tree - connected and no cycles
    0
   /|\
  1 2 3
  |
  4

Example 2:
Input: n = 5, edges = [[0,1],[1,2],[2,3],[1,3],[3,4]]
Output: false
Explanation: Contains cycle: 1-2-3-1
    0
    |
    1---3
   /|   |\
  2-+   | 4
        |
        4

Example 3:
Input: n = 5, edges = [[0,1],[0,4]]
Output: false
Explanation: Not connected - nodes 2 and 3 are isolated
```

Interview Questions & Answers:

1. **Q:** "What are the properties of a valid tree?"
   A: A valid tree must be: 1) Connected (all nodes reachable), 2) Acyclic (no cycles), 3) Have exactly n-1 edges for n nodes.
2. **Q:** "Can you use the edge count as a quick check?"
   A: Yes! A tree with n nodes must have exactly n-1 edges. If edge count is wrong, return false immediately.
3. **Q:** "How do you detect cycles in an undirected graph?"
   A: During DFS, if we visit a node that's already visited and it's not the immediate parent, we found a cycle.
4. **Q:** "Could you use Union-Find for this problem?"
   A: Yes, Union-Find is excellent for this. If union operation fails (nodes already connected), there's a cycle.
5. **Q:** "What if the graph has multiple connected components?"
   A: Not a valid tree. A tree must be connected - DFS/BFS should visit all n nodes starting from any node.

Approach 1: DFS with Visited Array

Idea: Check edge count, then use DFS to verify connectivity and detect cycles.

Time Complexity: O(V + E) - DFS traversal
Space Complexity: O(V + E) - graph storage and visited array

```
using System;
using System.Collections.Generic;

public class Solution
{
    public bool ValidTree(int n, int[][] edges)
    {
        // Tree must have exactly n-1 edges
        if (edges.Length != n - 1) return false;

        // Build adjacency list
        var graph = new List<List<int>>(n);
        for (int i = 0; i < n; i++)
        {
            graph.Add(new List<int>());
        }

        foreach (var edge in edges)
        {
            graph[edge[0]].Add(edge[1]);
            graph[edge[1]].Add(edge[0]);
        }

        var visited = new bool[n];

        // Check for cycles and connectivity
        if (HasCycle(graph, 0, -1, visited)) return false;

        // Check if all nodes are visited (connected)
        for (int i = 0; i < n; i++)
        {
            if (!visited[i]) return false;
        }

        return true;
    }

    private bool HasCycle(List<List<int>> graph, int node, int parent, bool[] visited)
    {
        visited[node] = true;

        foreach (int neighbor in graph[node])
        {
            if (neighbor == parent) continue; // Skip parent

            if (visited[neighbor] || HasCycle(graph, neighbor, node, visited))
            {
                return true; // Cycle detected
            }
        }

        return false;
    }
}
```

Approach 2: BFS with Connectivity Check

Explanation: Use BFS to check connectivity and detect cycles by counting visited nodes.

Time Complexity: O(V + E) - BFS traversal
Space Complexity: O(V + E) - graph storage and queue

```
using System;
using System.Collections.Generic;

public class Solution
{
    public bool ValidTree(int n, int[][] edges)
    {
        // Tree must have exactly n-1 edges
        if (edges.Length != n - 1) return false;

        // Build adjacency list
        var graph = new List<List<int>>(n);
        for (int i = 0; i < n; i++)
        {
            graph.Add(new List<int>());
        }

        foreach (var edge in edges)
        {
            graph[edge[0]].Add(edge[1]);
            graph[edge[1]].Add(edge[0]);
        }

        // BFS to check connectivity
        var visited = new bool[n];
        var queue = new Queue<int>();
        queue.Enqueue(0);
        visited[0] = true;
        int visitedCount = 1;

        while (queue.Count > 0)
        {
            int node = queue.Dequeue();

            foreach (int neighbor in graph[node])
            {
                if (!visited[neighbor])
                {
                    visited[neighbor] = true;
                    queue.Enqueue(neighbor);
                    visitedCount++;
                }
            }
        }

        // All nodes should be visited if graph is connected
        return visitedCount == n;
    }
}
```

Approach 3: Optimal - Union-Find (Disjoint Set Union)

Detailed Reasoning: Use Union-Find data structure for efficient cycle detection and connectivity checking. This approach is optimal because it naturally handles both requirements of a tree and has nearly constant time per operation.

Time Complexity: O(E × α(V)) - where α is inverse Ackermann function, nearly O(E)
Space Complexity: O(V) - Union-Find parent and rank arrays

```
using System;

public class Solution
{
    /// <summary>
    /// Validates if graph forms a tree using Union-Find data structure
    /// Time: O(E * α(V)), Space: O(V)
    /// </summary>
    public bool ValidTree(int n, int[][] edges)
    {
        // Tree must have exactly n-1 edges
        if (edges.Length != n - 1) return false;
```

```
        // Initialize Union-Find structure
        UnionFind uf = new UnionFind(n);

        // Process each edge
        foreach (var edge in edges)
        {
            int node1 = edge[0];
            int node2 = edge[1];

            // If nodes are already connected, adding this edge creates a cycle
            if (!uf.Union(node1, node2))
            {
                return false;
            }
        }

        // If we successfully added all edges without cycles,
        // and we have n-1 edges, the graph is a valid tree
        return true;
    }
}

/// <summary>
/// Union-Find data structure with path compression and union by rank
/// </summary>
public class UnionFind
{
    private int[] parent;
    private int[] rank;

    public UnionFind(int n)
    {
        parent = new int[n];
        rank = new int[n];

        // Initialize each node as its own parent
        for (int i = 0; i < n; i++)
        {
            parent[i] = i;
            rank[i] = 0;
        }
    }

    /// <summary>
    /// Find root of component with path compression
    /// </summary>
    public int Find(int x)
    {
        if (parent[x] != x)
        {
            parent[x] = Find(parent[x]); // Path compression
        }
        return parent[x];
    }
```

```
    /// <summary>
    /// Union two components if they're not already connected
    /// Returns true if union was successful, false if nodes were already connected
    /// </summary>
    public bool Union(int x, int y)
    {
        int rootX = Find(x);
        int rootY = Find(y);

        // Already in same component - would create cycle
        if (rootX == rootY) return false;

        // Union by rank for efficiency
        if (rank[rootX] < rank[rootY])
        {
            parent[rootX] = rootY;
        }
        else if (rank[rootX] > rank[rootY])
        {
            parent[rootY] = rootX;
        }
        else
        {
            parent[rootY] = rootX;
            rank[rootX]++;
        }

        return true;
    }
}
```

Key Takeaways:

- **Tree properties**: Exactly n-1 edges, connected, and acyclic
- **Quick edge count check**: Eliminates invalid cases immediately
- **Union-Find efficiency**: Optimal for connectivity and cycle detection problems

PROBLEM 6: NUMBER OF CONNECTED COMPONENTS IN AN UNDIRECTED GRAPH

Problem Statement:

You have a graph of `n` nodes. You are given an integer `n` and an array `edges` where `edges[i] = [ai, bi]` indicates that there is an edge between `ai` and `bi` in the graph.

Return the number of connected components in the graph.

Test Cases:

```
Example 1:
Input: n = 5, edges = [[0,1],[1,2],[3,4]]
Output: 2
Explanation: Two components: {0,1,2} and {3,4}
    0---1---2    3---4

Example 2:
Input: n = 5, edges = [[0,1],[1,2],[2,3],[3,4]]
Output: 1
Explanation: One component: {0,1,2,3,4}
    0---1---2---3---4

Example 3:
Input: n = 3, edges = []
Output: 3
Explanation: Three components: {0}, {1}, {2} (all isolated)
```

Interview Questions & Answers:

1. **Q:** "What defines a connected component?"
   A: A connected component is a maximal set of vertices where every pair is connected by some path. Maximal means we can't add any more vertices to the component.
2. **Q:** "How do you avoid counting the same component multiple times?"
   A: Mark nodes as visited during DFS/BFS traversal. Each time we start a new traversal from an unvisited node, we've found a new component.
3. **Q:** "Can you solve this without building the adjacency list?"
   A: Yes, using Union-Find. Process edges to union connected nodes, then count distinct root parents.
4. **Q:** "What if there are no edges?"
   A: Each node forms its own component, so answer is n.
5. **Q:** "How would you modify this to return the actual components, not just the count?"
   A: During DFS/BFS, collect all nodes visited in each traversal and store them as separate component lists.

Approach 1: DFS with Visited Tracking

Idea: Use DFS to explore each unvisited node and mark all nodes in its component as visited.

Time Complexity: O(V + E) - visit each node and edge once
Space Complexity: O(V + E) - adjacency list and visited array

```
using System;
using System.Collections.Generic;

public class Solution
{
```

```
    public int CountComponents(int n, int[][] edges)
    {
        // Build adjacency list
        var graph = new List<List<int>>(n);
        for (int i = 0; i < n; i++)
        {
            graph.Add(new List<int>());
        }

        foreach (var edge in edges)
        {
            graph[edge[0]].Add(edge[1]);
            graph[edge[1]].Add(edge[0]);
        }

        var visited = new bool[n];
        int components = 0;

        // Check each node
        for (int i = 0; i < n; i++)
        {
            if (!visited[i])
            {
                // Found new component
                components++;
                DFS(graph, i, visited);
            }
        }

        return components;
    }

    private void DFS(List<List<int>> graph, int node, bool[] visited)
    {
        visited[node] = true;

        foreach (int neighbor in graph[node])
        {
            if (!visited[neighbor])
            {
                DFS(graph, neighbor, visited);
            }
        }
    }
}
```

Approach 2: BFS with Queue

Explanation: Use BFS instead of DFS to explore connected components iteratively.

Time Complexity: O(V + E) - visit each node and edge once
Space Complexity: O(V + E) - adjacency list and queue

```
using System;
using System.Collections.Generic;

public class Solution
{
    public int CountComponents(int n, int[][] edges)
    {
        // Build adjacency list
        var graph = new List<List<int>>(n);
        for (int i = 0; i < n; i++)
        {
            graph.Add(new List<int>());
        }

        foreach (var edge in edges)
        {
            graph[edge[0]].Add(edge[1]);
            graph[edge[1]].Add(edge[0]);
        }

        var visited = new bool[n];
        int components = 0;

        for (int i = 0; i < n; i++)
        {
            if (!visited[i])
            {
                components++;
                BFS(graph, i, visited);
            }
        }

        return components;
    }

    private void BFS(List<List<int>> graph, int start, bool[] visited)
    {
        var queue = new Queue<int>();
        queue.Enqueue(start);
        visited[start] = true;

        while (queue.Count > 0)
        {
            int node = queue.Dequeue();

            foreach (int neighbor in graph[node])
            {
                if (!visited[neighbor])
                {
                    visited[neighbor] = true;
                    queue.Enqueue(neighbor);
                }
            }
        }
```

```
    }
}
```

Approach 3: Optimal - Union-Find

Detailed Reasoning: Use Union-Find to efficiently group connected nodes. Count distinct components by counting nodes that are their own parent (root nodes). This approach is optimal because it processes each edge exactly once and provides nearly constant-time operations.

Time Complexity: O(E × α(V)) - where α is inverse Ackermann, nearly O(E)
Space Complexity: O(V) - parent and rank arrays

```
using System;

public class Solution
{
    /// <summary>
    /// Counts connected components using Union-Find data structure
    /// Time: O(E * α(V)), Space: O(V)
    /// </summary>
    public int CountComponents(int n, int[][] edges)
    {
        UnionFind uf = new UnionFind(n);

        // Union all connected nodes
        foreach (var edge in edges)
        {
            uf.Union(edge[0], edge[1]);
        }

        // Count distinct components
        return uf.GetComponentCount();
    }
}

/// <summary>
/// Union-Find data structure optimized for component counting
/// </summary>
public class UnionFind
{
    private int[] parent;
    private int[] rank;
    private int components;

    public UnionFind(int n)
    {
        parent = new int[n];
        rank = new int[n];
        components = n; // Initially, each node is its own component

        for (int i = 0; i < n; i++)
        {
            parent[i] = i;
            rank[i] = 0;
```

```
        }
    }

    /// <summary>
    /// Find root with path compression
    /// </summary>
    public int Find(int x)
    {
        if (parent[x] != x)
        {
            parent[x] = Find(parent[x]); // Path compression
        }
        return parent[x];
    }

    /// <summary>
    /// Union two nodes and decrease component count if they were separate
    /// </summary>
    public void Union(int x, int y)
    {
        int rootX = Find(x);
        int rootY = Find(y);

        if (rootX == rootY) return; // Already in same component

        // Union by rank
        if (rank[rootX] < rank[rootY])
        {
            parent[rootX] = rootY;
        }
        else if (rank[rootX] > rank[rootY])
        {
            parent[rootY] = rootX;
        }
        else
        {
            parent[rootY] = rootX;
            rank[rootX]++;
        }

        components--; // Merged two components into one
    }

    /// <summary>
    /// Get current number of connected components
    /// </summary>
    public int GetComponentCount()
    {
        return components;
    }
}
```

Key Takeaways:

- **Component identification**: Each DFS/BFS starting from unvisited node finds one component
- **Union-Find optimization**: Tracks component count dynamically as edges are processed
- **Graph traversal**: Both DFS and BFS work equally well for component detection

PROBLEM 7: LONGEST CONSECUTIVE SEQUENCE

Problem Statement:

Given an unsorted array of integers `nums`, return the length of the longest consecutive elements sequence.

You must write an algorithm that runs in `O(n)` time complexity.

Test Cases:

```
Example 1:
Input: nums = [100,4,200,1,3,2]
Output: 4
Explanation: The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore its len

Example 2:
Input: nums = [0,3,7,2,5,8,4,6,0,1]
Output: 9
Explanation: The longest consecutive sequence is [0,1,2,3,4,5,6,7,8] with length 9.

Example 3:
Input: nums = []
Output: 0
Explanation: Empty array has no consecutive sequence.
```

Interview Questions & Answers:

1. **Q:** "How is this a graph problem?"
   A: Think of each number as a node, with edges to consecutive numbers (n-1 and n+1). We're finding the longest path in this implicit graph structure.
2. **Q:** "Why can't we just sort the array first?"
   A: Sorting would take O(n log n) time, but the problem requires O(n) solution. We need a more efficient approach.
3. **Q:** "How do you avoid checking the same sequence multiple times?"
   A: Only start checking from the beginning of a potential sequence - numbers that don't have a predecessor (n-1) in the set.
4. **Q:** "What if there are duplicate numbers?"
   A: Use HashSet to eliminate duplicates automatically. Duplicates don't extend consecutive sequences.

5. **Q:** "Could you use Union-Find for this problem?"
   A: Yes, you could union consecutive numbers and find the largest component, but HashSet approach is simpler and more efficient.

Approach 1: Sorting

Idea: Sort array and find longest consecutive sequence by linear scan.

Time Complexity: O(n log n) - sorting dominates
Space Complexity: O(1) - if sorting in-place, otherwise O(n)

```
using System;

public class Solution
{
    public int LongestConsecutive(int[] nums)
    {
        if (nums.Length == 0) return 0;

        Array.Sort(nums);
        int maxLength = 1;
        int currentLength = 1;

        for (int i = 1; i < nums.Length; i++)
        {
            if (nums[i] == nums[i - 1])
            {
                // Skip duplicates
                continue;
            }
            else if (nums[i] == nums[i - 1] + 1)
            {
                // Consecutive number found
                currentLength++;
            }
            else
            {
                // Sequence broken
                maxLength = Math.Max(maxLength, currentLength);
                currentLength = 1;
            }
        }

        return Math.Max(maxLength, currentLength);
    }
}
```

Approach 2: HashSet with Naive Approach

Explanation: Use HashSet for O(1) lookup, but check consecutive sequences from every number.

Time Complexity: O(n²) - for each number, potentially check entire sequence
Space Complexity: O(n) - HashSet storage

```
using System;
using System.Collections.Generic;

public class Solution
{
    public int LongestConsecutive(int[] nums)
    {
        if (nums.Length == 0) return 0;

        var numSet = new HashSet<int>(nums);
        int maxLength = 1;

        foreach (int num in numSet)
        {
            int currentLength = 1;
            int current = num;

            // Check consecutive numbers in positive direction
            while (numSet.Contains(current + 1))
            {
                current++;
                currentLength++;
            }

            maxLength = Math.Max(maxLength, currentLength);
        }

        return maxLength;
    }
}
```

Approach 3: Optimal - Smart HashSet

Detailed Reasoning: Use HashSet for O(1) lookups, but optimize by only starting sequence checks from the beginning of potential sequences. A number starts a sequence only if (number - 1) is not in the set. This ensures each number is visited at most twice.

Time Complexity: O(n) - each number visited at most twice
Space Complexity: O(n) - HashSet storage

```
using System;
using System.Collections.Generic;

public class Solution
{
    /// <summary>
    /// Finds longest consecutive sequence using optimized HashSet approach
    /// Time: O(n), Space: O(n)
    /// </summary>
    public int LongestConsecutive(int[] nums)
    {
        // Handle empty array
        if (nums.Length == 0) return 0;

        // Store all numbers in HashSet for O(1) lookup
        var numSet = new HashSet<int>(nums);
        int maxLength = 0;

        foreach (int num in numSet)
        {
            // Only start checking from the beginning of a potential sequence
            // If num-1 exists, then num is not the start of a sequence
            if (!numSet.Contains(num - 1))
            {
                int currentNum = num;
                int currentLength = 1;

                // Extend sequence as far as possible
                while (numSet.Contains(currentNum + 1))
                {
                    currentNum++;
                    currentLength++;
                }

                // Update maximum length found
                maxLength = Math.Max(maxLength, currentLength);
            }
        }

        return maxLength;
    }
}
```

Key Takeaways:

- **Implicit graph structure**: Numbers connected to consecutive neighbors
- **Sequence start optimization**: Only begin from numbers that don't have predecessors
- **HashSet efficiency**: O(1) lookups enable linear time complexity

DAY 6 SUMMARY

Summary of Concepts Covered

Graph Fundamentals:

- **Graph representation**: Adjacency lists vs matrices, space-time trade-offs
- **Graph types**: Directed/undirected, weighted/unweighted, connected/disconnected
- **Graph properties**: Cycles, connectivity, components

Graph Traversal Algorithms:

- **Depth-First Search (DFS)**: Recursive and iterative implementations
- **Breadth-First Search (BFS)**: Level-by-level exploration using queues
- **Traversal applications**: Component detection, cycle detection, path finding

Specialized Graph Algorithms:

- **Union-Find (Disjoint Set Union)**: Efficient component tracking and cycle detection
- **Topological Sorting**: Kahn's algorithm for dependency resolution
- **Connected Components**: Identification and counting in undirected graphs

Graph Problem Patterns:

- **Grid-based graphs**: 2D matrices as implicit graph structures
- **State space exploration**: DFS/BFS for finding valid configurations
- **Cycle detection**: Different approaches for directed vs undirected graphs
- **Connectivity analysis**: Reachability and component analysis

Time & Space Complexity Summary

Problem Type	Brute Force	Optimized	Key Technique
Islands Counting	**O((mn)³)** redundant	**O(mn)**	DFS/BFS with visited marking
Graph Cloning	**O(V+E)**	**O(V+E)**	HashMap for cycle handling
Water Flow	**O(mn×(mn))** each cell	**O(mn)**	Reverse DFS from edges
Course Schedule	**O(V+E)²**	**O(V+E)**	Topological sort/cycle detection
Valid Tree	**O(V+E)**	**O(V×α(V))**	Union-Find optimization
Connected Components	**O(V+E)**	**O(V×α(V))**	Union-Find for efficiency
Consecutive Sequence	**O(n log n)** sorting	**O(n)**	Smart HashSet optimization

Pattern Recognition Guide

Use DFS When:

- Need to explore all paths or complete subgraphs
- Working with tree-like structures or recursive relationships

- Cycle detection in directed graphs
- Memory usage proportional to depth is acceptable

Use BFS When:

- Finding shortest paths in unweighted graphs
- Level-by-level processing is natural
- Want to limit memory usage to graph width
- Processing nodes by distance from source

Use Union-Find When:

- Dynamic connectivity queries
- Cycle detection in undirected graphs
- Component counting with incremental updates
- Minimum spanning tree algorithms

Common Graph Problem Patterns

Pattern 1: Component Exploration

```
int ExploreComponents(graph) {
    var visited = new bool[n];
    int components = 0;

    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            components++;
            DFS(graph, i, visited); // or BFS
        }
    }
    return components;
}
```

Pattern 2: Cycle Detection (Undirected)

```
bool HasCycle(graph, node, parent, visited) {
    visited[node] = true;

    foreach (neighbor in graph[node]) {
        if (neighbor == parent) continue;
        if (visited[neighbor] || HasCycle(graph, neighbor, node, visited)) {
            return true;
        }
    }
    return false;
}
```

Pattern 3: Topological Sort

```
bool TopologicalSort(graph) {
    var inDegree = CalculateInDegree(graph);
    var queue = new Queue<int>();

    // Add nodes with no dependencies
    for (int i = 0; i < n; i++) {
        if (inDegree[i] == 0) queue.Enqueue(i);
    }

    int processed = 0;
    while (queue.Count > 0) {
        int node = queue.Dequeue();
        processed++;

        foreach (neighbor in graph[node]) {
            if (--inDegree[neighbor] == 0) {
                queue.Enqueue(neighbor);
            }
        }
    }

    return processed == n; // True if no cycles
}
```

Review Tips

- **Master graph representations**: Practice converting between adjacency list/matrix
- **Understand traversal differences**: Know when to use DFS vs BFS
- **Practice cycle detection**: Different techniques for directed vs undirected graphs
- **Visualize graph problems**: Draw diagrams to understand connectivity and flow
- **Recognize implicit graphs**: Grid problems, state space, sequence problems

How to Practice Mock Interviews

1. **Problem Classification** (3 min): Identify graph type and required operations
2. **Representation Choice** (2 min): Adjacency list vs matrix, justify choice
3. **Algorithm Selection** (5 min): DFS vs BFS vs Union-Find, explain trade-offs
4. **Implementation** (25 min): Code while explaining graph navigation logic
5. **Complexity Analysis** (5 min): Time/space complexity with justification
6. **Testing** (10 min): Edge cases like empty graphs, disconnected components

Common Mistakes to Avoid

- **Visited array management**: Forgetting to mark nodes as visited or incorrect reset
- **Infinite loops in cycles**: Not handling visited nodes properly in cycle detection
- **Parent tracking**: Incorrect parent handling in undirected graph cycle detection

- **Graph representation errors**: Mixing up adjacency list indices or edge directions
- **Component counting**: Double-counting components or missing isolated nodes
- **Base case handling**: Not handling empty graphs or single-node graphs
- **Memory optimization**: Using adjacency matrix for sparse graphs

Advanced Topics for Further Study

- **Strongly Connected Components**: Kosaraju's and Tarjan's algorithms
- **Minimum Spanning Trees**: Kruskal's and Prim's algorithms
- **Shortest Path Algorithms**: Dijkstra's, Bellman-Ford, Floyd-Warshall
- **Network Flow**: Maximum flow algorithms, min-cut max-flow theorem
- **Graph Coloring**: Bipartite graphs, vertex coloring problems
- **Advanced DFS**: Bridges, articulation points, strongly connected components

Real-world Applications

- **Social Networks**: Friend recommendations, community detection, influence analysis
- **Transportation**: Route planning, traffic optimization, network design
- **Web Technologies**: Page ranking, web crawling, link analysis
- **Dependencies**: Build systems, package management, task scheduling
- **Biology**: Protein interactions, gene regulatory networks, phylogenetic trees
- **Computer Networks**: Routing protocols, network topology, failure detection
- **Game Development**: Pathfinding, AI decision trees, procedural generation

Graph Algorithm Toolkit

Essential Algorithms to Master:

- [ ] DFS (recursive and iterative)
- [ ] BFS with queue
- [ ] Union-Find with path compression
- [ ] Topological sorting (Kahn's algorithm)
- [ ] Cycle detection (directed and undirected)
- [ ] Connected components identification
- [ ] Shortest path in unweighted graphs (BFS)
- [ ] Graph validation and property checking

Problem-Solving Checklist:

- [ ] Identify graph structure (explicit vs implicit)
- [ ] Choose appropriate representation
- [ ] Select optimal traversal algorithm
- [ ] Handle edge cases (empty, disconnected, cycles)
- [ ] Optimize for time/space constraints
- [ ] Verify connectivity and component requirements
- [ ] Test with various graph topologies

This completes Day 6: Graphs (Part 1) with comprehensive coverage of all 7 problems, following the same detailed structure and educational approach as previous days.