	Topic	Subtopic	Rev-1	Rev-2	Rev-3
1	Basics	Types & Representation			
2	Traversals	BFS & DFS			
3		Problems			
4		Connected Components			
5		Provinces			
6		Islands			
7		Enclaves			
8		Surrounded Regions			
9		Flood Fill			
10		Rotten Oranges			
11		Nearest Cell 1			
12		Cycle in Undirected			
13		Bipartite Check			
14	Topo & DAG	Kahn's Algorithm			
15		Cycle in Directed			
16		Safe States			
17		Course Schedule I			
18		Course Schedule II			
19	Shortest Path	DAG Shortest Path			
20		Unweighted Graph			
21		Dijkstra			
22		Bellman-Ford			
23		Floyd-Warshall			
24		Johnson			
25		A* Search			
26		Print Path			
27		Flights K Stops			
28		Ways to Destination			
29		Word Ladder I			
30		Word Ladder II			
31		Min Multiplications			
32		Binary Maze			

	Topic	Subtopic	Rev-1	Rev-2	Rev-3
33		Min Effort Path			
34		City with Fewest Neighbors			
35	MST	Intro			
36		Disjoint Set			
37		Prim & Kruskal			
38	DSU Problems	Make Network Connected			
39		Accounts Merge			
40		Islands II			
41		Large Island			
42		Remove Stones			
43	SCC	Kosaraju			
44		Bridges / Tarjan			
45		Articulation Points			

Complete Guide to Graph Data Structures

Table of Contents

Data Structure Classification

Data structures can be broadly classified into two main categories:

Linear Data Structures

Data elements are arranged in a sequential (linear) manner.

- **Examples**: Arrays, Linked Lists, Stacks, Queues

Non-Linear Data Structures

Data elements are not arranged in a sequential manner. They form a hierarchical or interconnected structure.

- **Examples**: Trees, Graphs

What is a Graph?

A **Graph** is a non-linear data structure consisting of:

- **Nodes/Vertices/Points**: Individual data elements
- **Edges/Lines/Arcs**: Connections between pairs of nodes

Key Definitions:

- A graph is a finite set of nodes and edges used to connect pairs of nodes
- A pictorial representation of a set of objects where some pairs are connected by links
- Can be considered as a cyclic tree

Example Graph Structure:

```
Vertices: V = {A, B, C, D, E}
Edges: E = {(A,B), (B,C), (C,E), (E,D), (D,B), (D,A)}
```

Applications of Graphs

Graphs have numerous real-world applications:

Social Media & E-commerce

- **Recommendation Systems**: Used in social media platforms and e-commerce applications to suggest friends, products, or content

Computer Science

- **Flow of Computation**: Represent computational processes and data flow

Navigation Systems

- **Google Maps**: Uses graphs where road intersections are vertices and roads are edges
- **Shortest Path Algorithms**: Calculate optimal routes between locations

World Wide Web

- **Web Page Linking**: Pages connect to others through hyperlinks, forming a graph structure

Types of Graphs

1. Null Graph

- **Definition**: Contains vertices but no edges between them
- **Characteristics**: n nodes, zero edges

2. Finite Graph

- **Definition**: Contains a finite number of vertices and edges
- **Characteristics**: Limited, countable elements

3. Infinite Graph

- **Definition**: Contains infinite number of vertices and edges
- **Characteristics**: Unbounded structure

4. Cyclic Graph

- **Definition**: Contains at least one cycle
- **Characteristics**: Closed paths exist within the graph

5. Acyclic Graph

- **Definition**: Contains no cycles
- **Characteristics**: No closed paths, tree-like structure

6. Simple Graph

- **Definition**: Only one edge between any pair of vertices
- **Characteristics**: No multiple edges or self-loops

7. Multi Graph

- **Definition**: Multiple edges between pairs of vertices
- **Characteristics**: No self-loops but allows parallel edges

8. Complete Graph

- **Definition**: Edge exists between every pair of vertices
- **Characteristics**: Every complete graph is also a simple graph
- **Degree**: Each node has degree = (number of vertices - 1)

9. Pseudo Graph

- **Definition**: Contains at least one self-loop
- **Characteristics**: Vertices can connect to themselves

10. Regular Graph

- **Definition**: All vertices have equal degree
- **Note**: Complete graphs are regular, but not all regular graphs are complete

11. Connected Graph

- **Definition**: At least one path exists between every pair of vertices
- **Characteristics**: No isolated components

12. Disconnected Graph

- **Definition**: No path exists between at least one pair of vertices
- **Characteristics**: Contains isolated components

13. Directed Graph (Digraph)

- **Definition**: All edges have direction indicating traversal order
- **Representation**: G = (V, E) where edges map to ordered pairs (Vi, Vj)
- **Example**: Like following someone on social media (one-way relationship)

14. Undirected Graph

- **Definition**: No orientation/direction on edges
- **Characteristics**: All edges are bidirectional
- **Example**: Facebook friendship (mutual relationship)

15. Weighted Graph

- **Definition**: Edges have associated values/weights
- **Use Cases**: Represent cost, distance, or capacity

16. Unweighted Graph

- **Definition**: No values associated with edges
- **Default**: All graphs are unweighted unless specified

17. Dense & Sparse Graphs

- **Dense Graph**: Number of edges close to maximum possible
- **Sparse Graph**: Contains only a few edges

18. Bipartite Graph

- **Definition**: Vertex set can be split into two disjoint sets (V1, V2)
- **Characteristics**: Edges only connect vertices between different sets

19. Complete Bipartite Graph

- **Definition**: Every vertex in first set connects to every vertex in second set
- **Notation**: Kx,y (x vertices in first set, y in second set)

Graph Terminology

Basic Elements

- **Vertex (Node)**: Individual data element
- **Edge**: Path/connection between two vertices (endpoints)

Vertex Relationships

- **Adjacent Vertices**: Two vertices connected by an edge
- **Incident Edge**: Edge connected to a vertex as an endpoint

Edge Direction (for Directed Graphs)

- **Outgoing Edge**: Edge directed away from a vertex
- **Incoming Edge**: Edge directed toward a vertex

Degree Concepts

- **Degree**: Number of edges incident to a vertex
- **In-degree**: Number of incoming edges to a vertex
- **Out-degree**: Number of outgoing edges from a vertex

Special Elements

- **Parallel Edges**: Multiple edges between same pair of vertices
- **Self-loop**: Edge connecting a vertex to itself
- **Forest**: Graph with no cycles
- **Path**: Sequence of successive edges between nodes

- **Path Length**: Number of edges in a path
- **Simple Path**: Path with distinct vertices

Special Vertices

- **Source Vertex**: In-degree = 0
- **Sink Vertex**: Out-degree = 0

Connectivity

- **Strongly Connected**: Directed path exists between every pair of vertices (both directions)
- **Weakly Connected**: Becomes connected when all directed edges are replaced with undirected edges
- **Bridge**: Edge whose removal disconnects the graph

Graph Representation

1. Adjacency Matrix

Description: 2D matrix representation where matrix[i][j] indicates connection between vertex i and vertex j.

Matrix Size: V × V (where V = number of vertices)

Values:

- **Unweighted**: 1 (edge exists) or 0 (no edge)
- **Weighted**: Actual weight value or 0 (no edge)

For Directed Graphs: matrix[i][j] = 1 means edge from vertex i to vertex j

For Undirected Graphs: matrix[i][j] = matrix[j][i] = 1 (symmetric matrix)

Advantages:

- Easy to understand and implement
- O(1) time complexity for:
  - Checking edge existence
  - Adding an edge
  - Removing an edge

Disadvantages:

- $O(V^2)$ space complexity (inefficient for sparse graphs)
- $O(V^2)$ time complexity for adding/removing vertices

2. Adjacency List

Description: Array of linked lists where each vertex maintains a list of its neighbors.

Structure:

- Array indexed by vertex number
- Each array element points to a linked list of adjacent vertices

Implementation Options:

- Linked list representation
- Array-based representation

Advantages:

- Space efficient: O(V + E) where E = number of edges
- O(1) time for adding/deleting edges and vertices
- Clear visualization of adjacent nodes

Disadvantages:

- Slower for checking if two vertices are adjacent
- O(V) time complexity for adjacency testing

3. Incidence Matrix

Description: Matrix representation where rows represent vertices and columns represent edges.

Matrix Size: V × E (where V = vertices, E = edges)

Values:

- **0**: Vertex not connected to edge
- **1**: Vertex connected as outgoing endpoint
- **-1**: Vertex connected as incoming endpoint

Use Case: Particularly useful for directed graphs where edge direction matters

Advantages:

- Clear representation of vertex-edge relationships
- Useful for certain graph algorithms

Disadvantages:

- Space inefficient: O(V × E)
- More complex than other representations

4. Incidence List

Description: Each vertex maintains a list of edges incident to it.

Structure: Similar to adjacency list but stores edge information instead of just neighboring vertices.

Graph Data Structures and Algorithms Guide

Table of Contents

Introduction

A graph is a collection of nodes (vertices) connected by edges. Graphs can be:

- **Directed**: Edges have a direction (one-way)
- **Undirected**: Edges are bidirectional (two-way)
- **Weighted**: Edges have associated weights/costs
- **Unweighted**: All edges have equal weight

Graph Representations

Adjacency List

An adjacency list represents a graph as a dictionary where each key is a node and the value is a collection of its neighbors.

1. Undirected Graph (Unweighted)

```
# Input format: n (nodes), e (edges), followed by e lines of edge pairs
n, e = map(int, input().split())
edges = [input().split() for i in range(e)]
edges_conv = [(int(i), int(j)) for i, j in edges]

# Initialize graph with empty sets for each node
graph = {i: set() for i in range(1, n+1)}

# Add edges in both directions (undirected)
for i, j in edges_conv:
    graph[i].add(j)
    graph[j].add(i)

print(graph)
```

Example Input:

```
4 4
1 2
2 3
3 4
4 1
```

Output:

```
{1: {2, 4}, 2: {1, 3}, 3: {2, 4}, 4: {1, 3}}
```

2. Directed Graph (Unweighted)

```
n, e = map(int, input().split())
edges = [input().split() for i in range(e)]
edges_conv = [(int(i), int(j)) for i, j in edges]

# Initialize graph with empty sets
graph = {i: set() for i in range(1, n+1)}

# Add edges in one direction only (directed)
for i, j in edges_conv:
    graph[i].add(j)

print(graph)
```

3. Undirected Weighted Graph

```
n, e = map(int, input().split())
edges = [input().split() for i in range(e)]
edges_conv = [(int(i), int(j), int(w)) for i, j, w in edges]

# Initialize graph with empty dictionaries for weights
graph = {i: dict() for i in range(1, n+1)}

# Add weighted edges in both directions
for i, j, w in edges_conv:
    graph[i][j] = w
    graph[j][i] = w

print(graph)
```

Example Input:

```
3 3
1 2 5
2 3 3
3 1 2
```

Output:

```
{1: {2: 5, 3: 2}, 2: {1: 5, 3: 3}, 3: {2: 3, 1: 2}}
```

4. Directed Weighted Graph

```
n, e = map(int, input().split())
edges = [input().split() for i in range(e)]
edges_conv = [(int(i), int(j), int(w)) for i, j, w in edges]

# Initialize graph with empty dictionaries
graph = {i: dict() for i in range(1, n+1)}

# Add weighted edges in one direction only
for i, j, w in edges_conv:
    graph[i][j] = w

print(graph)
```

Adjacency Matrix

An adjacency matrix represents a graph as a 2D array where matrix[i][j] indicates the presence (and weight) of an edge between nodes i and j.

1. Undirected Graph (Unweighted)

```
n, e = map(int, input().split())
edges = [input().split() for i in range(e)]
# Convert to 0-indexed for matrix
edges_conv = [(int(i)-1, int(j)-1) for i, j in edges]

# Initialize n×n matrix with zeros
graph = [[0]*n for i in range(n)]

# Set matrix[i][j] = 1 for both directions
for i, j in edges_conv:
    graph[i][j] = 1
    graph[j][i] = 1

# Print matrix row by row
print(*graph, sep="\n")
```

2. Directed Graph (Unweighted)

```
n, e = map(int, input().split())
edges = [input().split() for i in range(e)]
edges_conv = [(int(i)-1, int(j)-1) for i, j in edges]

graph = [[0]*n for i in range(n)]

# Set matrix[i][j] = 1 for one direction only
for i, j in edges_conv:
    graph[i][j] = 1

print(*graph, sep="\n")
```

3. Weighted Undirected Graph

```
n, e = map(int, input().split())
edges = [input().split() for i in range(e)]
edges_conv = [(int(i)-1, int(j)-1, int(w)) for i, j, w in edges]

graph = [[0]*n for i in range(n)]

# Set matrix[i][j] = weight for both directions
for i, j, w in edges_conv:
    graph[i][j] = w
    graph[j][i] = w

print(*graph, sep="\n")
```

4. Weighted Directed Graph

```
n, e = map(int, input().split())
edges = [input().split() for i in range(e)]
edges_conv = [(int(i)-1, int(j)-1, int(w)) for i, j, w in edges]

graph = [[0]*n for i in range(n)]

# Set matrix[i][j] = weight for one direction only
for i, j, w in edges_conv:
    graph[i][j] = w

print(*graph, sep="\n")
```

Graph Traversal Algorithms

Breadth-First Search (BFS)

BFS explores nodes level by level, visiting all neighbors before moving to the next level. It uses a **queue** (FIFO - First In, First Out).

```
def bfs(graph, start_node, n):
    queue = []
    visited = {i: False for i in range(1, n+1)}

    # Start with the initial node
    queue.append(start_node)
    visited[start_node] = True

    while queue:
        # Remove from front of queue
        current = queue.pop(0)
        print(current, end="->")

        # Visit all unvisited neighbors
        for neighbor in graph[current]:
            if not visited[neighbor]:
                visited[neighbor] = True
                queue.append(neighbor)
```

Key Points:

- Uses a queue for node processing
- Guarantees shortest path in unweighted graphs
- Time complexity: O(V + E)
- Space complexity: O(V)

Depth-First Search (DFS)

DFS explores as far as possible along each branch before backtracking. It uses a **stack** (LIFO - Last In, First Out).

Iterative Implementation

```
def dfs_iterative(graph, start_node, n):
    stack = []
    visited = {i: False for i in range(1, n+1)}

    visited[start_node] = True
    stack.append(start_node)

    while stack:
        # Remove from top of stack
        current = stack.pop()
        print(current, end="->")

        # Visit all unvisited neighbors
        for neighbor in graph[current]:
            if not visited[neighbor]:
                visited[neighbor] = True
                stack.append(neighbor)
```

Recursive Implementation

```
def dfs_recursive(graph, node, visited):
    visited[node] = True
    print(node, end="->")

    # Recursively visit all unvisited neighbors
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs_recursive(graph, neighbor, visited)

# Usage for all nodes
visited = {i: False for i in range(1, n+1)}
for i in range(1, n+1):
    if not visited[i]:
        dfs_recursive(graph, i, visited)
```

Key Points:

- Uses a stack (or recursion stack) for node processing
- Good for detecting cycles and topological sorting
- Time complexity: O(V + E)
- Space complexity: O(V)

Complete Examples

Example 1: Creating and Traversing an Undirected Graph

```
# Create undirected graph
n, e = 4, 4
edges = [("1", "2"), ("2", "3"), ("3", "4"), ("4", "1")]
edges_conv = [(int(i), int(j)) for i, j in edges]

graph = {i: set() for i in range(1, n+1)}
for i, j in edges_conv:
    graph[i].add(j)
    graph[j].add(i)

print("Graph:", graph)

# BFS traversal
print("\nBFS from node 1:")
bfs(graph, 1, n)

# DFS traversal
print("\nDFS from node 1:")
dfs_recursive(graph, 1, {i: False for i in range(1, n+1)})
```

Example 2: Working with Weighted Graphs

```
# Create weighted directed graph
n, e = 3, 3
edges = [("1", "2", "5"), ("2", "3", "3"), ("1", "3", "2")]
edges_conv = [(int(i), int(j), int(w)) for i, j, w in edges]

graph = {i: dict() for i in range(1, n+1)}
for i, j, w in edges_conv:
    graph[i][j] = w

print("Weighted Graph:", graph)

# Find shortest path using BFS (modified for weights)
def shortest_path(graph, start, end):
    # Implementation depends on algorithm (Dijkstra's, etc.)
    pass
```

Time and Space Complexity

Adjacency List vs Adjacency Matrix

Operation	Adjacency List	Adjacency Matrix
Storage	O(V + E)	$O(V^2)$
Add Edge	O(1)	O(1)
Remove Edge	O(V)	O(1)
Check Edge	O(V)	O(1)

Operation	Adjacency List	Adjacency Matrix
BFS/DFS	O(V + E)	$O(V^2)$

Connected Components

Problem Statement

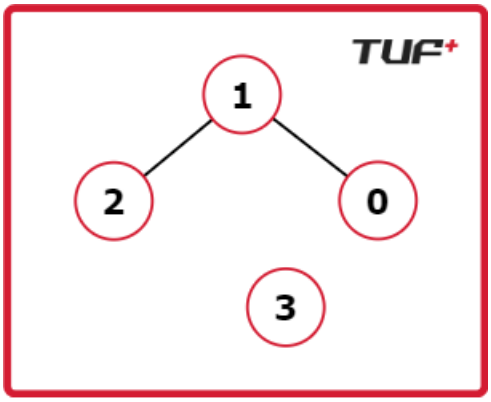
Given an undirected graph consisting of V vertices numbered from 0 to V-1 and E edges, find the number of connected components in the graph.

The ith edge is represented by [ai, bi], denoting an edge between vertex ai and bi. Two vertices u and v belong to the same component if there is a path from u to v or v to u.

Definition: A connected component is a subgraph of a graph in which there exists a path between any two vertices, and no vertex of the subgraph shares an edge with a vertex outside of the subgraph.

Examples

Example 1




```
Input: V = 4, edges = [[0,1], [1,2]]
Output: 2
```

Explanation: Vertices {0, 1, 2} form the first component and vertex 3 forms the second component.

Example 2

```
Input: V = 7, edges = [[0, 1], [1, 2], [2, 3], [4, 5]]
Output: 3
```

Explanation:

- The edges [0, 1], [1, 2], [2, 3] form a connected component with vertices {0, 1, 2, 3}
- The edge [4, 5] forms another connected component with vertices {4, 5}
- Vertex 6 is isolated and forms its own component
- Therefore, the graph has 3 connected components: {0, 1, 2, 3}, {4, 5}, and {6}

Approach

1. **Convert to Adjacency List**: Transform the given edges into an adjacency list representation for easy traversal

2. **Initialize Data Structures**:
    - Create a visited array to mark nodes as visited
    - Initialize a counter to count the number of components
3. **Traverse Components**:
    - For each unvisited node, increment the counter (new component found)
    - Perform BFS/DFS to visit all nodes in the current component
4. **Return Result**: The counter gives us the total number of connected components

Implementation

```
def solution(n, edges):
    # Create adjacency list
    nodes = [i for i in range(n)]
    graph = {i: set() for i in nodes}

    # Build the graph
    for i, j in edges:
        graph[i].add(j)
        graph[j].add(i)

    # Initialize visited array
    visited = {i: False for i in nodes}

    def bfs(graph, node):
        """Perform BFS to visit all nodes in current component"""
        visited[node] = True
        queue = [node]

        while queue:
            temp = queue.pop(0)
            for adj in graph[temp]:
                if not visited[adj]:
                    queue.append(adj)
                    visited[adj] = True

    # Count connected components
    count = 0
    for node in nodes:
        if not visited[node]:
            count += 1
            bfs(graph, node)

    return count

# Test the solution
v = 7
edges = [[0, 1], [1, 2], [2, 3], [4, 5]]
print(solution(v, edges))  # Output: 3
```

Algorithm Details

- **Time Complexity**: O(V + E) where V is the number of vertices and E is the number of edges
- **Space Complexity**: O(V + E) for the adjacency list and visited array

Number of Provinces

Problem Statement

Given an undirected graph with V vertices represented as an n x n adjacency matrix, find the number of provinces in the graph.

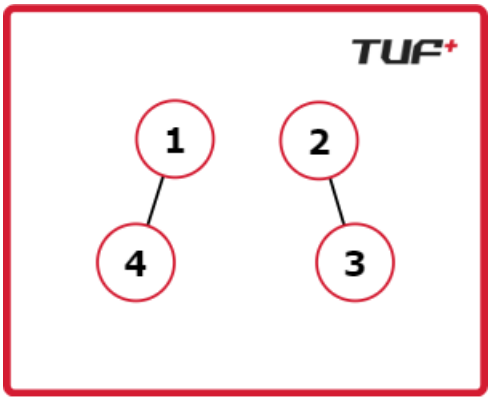
Two vertices u and v belong to a single province if there is a path from u to v or v to u. The graph is given as an n x n matrix `adj` where `adj[i][j] = 1` if the ith city and the jth city are directly connected, and `adj[i][j] = 0` otherwise.

Definition: A province is a group of directly or indirectly connected cities and no other cities outside of the group.

Examples

Example 1

```
Input: adj = [
    [1, 0, 0, 1],
    [0, 1, 1, 0],
    [0, 1, 1, 0],
    [1, 0, 0, 1]
]
Output: 2
```



Explanation: - Province 1: Cities [0, 3] - City 0 and city 3 have a path between them - Province 2: Cities [1, 2] - City 1 and city 2 have a path between them - There is no path between any city in province 1 and any city in province 2

Example 2

```
Input: adj = [
    [1, 0, 1],
    [0, 1, 0],
    [1, 0, 1]
]
Output: 2
```

Explanation:

- Province 1: Cities [0, 2] - City 0 and city 2 have a path between them
- Province 2: City [1] - City 1 has no path to city 0 or city 2, so it forms its own province

Approach

1. **Convert Adjacency Matrix to List**: Transform the given adjacency matrix into an adjacency list representation for easier traversal
2. **Initialize Data Structures**:
   - Create a visited array to mark nodes as visited
   - Initialize a counter to count the number of provinces
3. **Traverse Provinces**:
   - For each unvisited node, increment the counter (new province found)
   - Perform BFS to visit all nodes in the current province
4. **Return Result**: The counter gives us the total number of provinces

Implementation

```
def solution(adj_matrix):
    n = len(adj_matrix)
    nodes = [i for i in range(n)]

    # Convert adjacency matrix to adjacency list
    graph = {i: set() for i in nodes}
    for i in range(n):
        for j in range(n):
            if adj_matrix[i][j] == 1:
                graph[i].add(j)

    # Initialize visited array
    visited = {i: False for i in nodes}

    def bfs(graph, node):
        """Perform BFS to visit all nodes in current province"""
        visited[node] = True
        queue = [node]

        while queue:
            temp = queue.pop(0)
            for adj in graph[temp]:
                if not visited[adj]:
                    queue.append(adj)
                    visited[adj] = True

    # Count provinces
    count = 0
    for node in nodes:
        if not visited[node]:
            count += 1
            bfs(graph, node)

    return count

# Test the solution
adj_matrix = [
    [1, 0, 0, 1],
```

```
    [0, 1, 1, 0],
    [0, 1, 1, 0],
    [1, 0, 0, 1]
]
print(solution(adj_matrix))  # Output: 2
```

Optimized Implementation (Direct Matrix Traversal)

```
def solution_optimized(adj_matrix):
    n = len(adj_matrix)
    visited = [False] * n

    def dfs(node):
        """Perform DFS directly on adjacency matrix"""
        visited[node] = True
        for neighbor in range(n):
            if adj_matrix[node][neighbor] == 1 and not visited[neighbor]:
                dfs(neighbor)

    count = 0
    for i in range(n):
        if not visited[i]:
            count += 1
            dfs(i)

    return count
```

Algorithm Details

- **Time Complexity**: $O(V^2)$ where V is the number of vertices (due to adjacency matrix traversal)
- **Space Complexity**: $O(V^2)$ for the adjacency list and visited array

Number of Islands

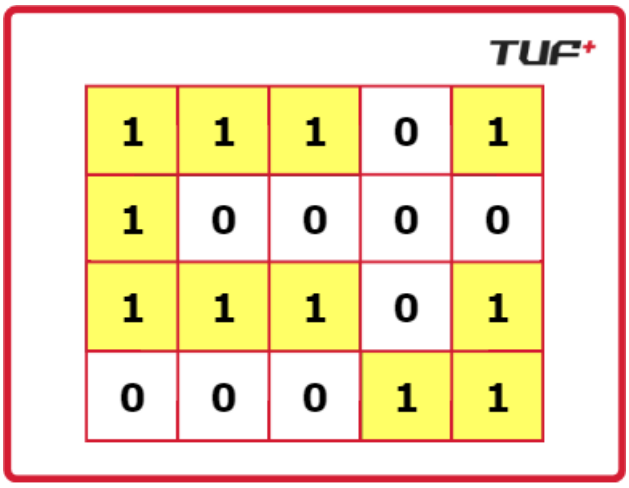
Problem Statement

Given a grid of size N x M (N is the number of rows and M is the number of columns in the grid) consisting of '0's (Water) and '1's (Land), find the number of islands.

An island is surrounded by water and is formed by connecting adjacent lands horizontally, vertically, or diagonally (i.e., in all 8 directions).

Examples

Example 1




```
Input: grid = [
    ["1", "1", "1", "0", "1"],
    ["1", "0", "0", "0", "0"],
    ["1", "1", "1", "0", "1"],
    ["0", "0", "0", "1", "1"]
]
Output: 2
```

Explanation: This grid contains 2 islands. Each '1' represents a piece of land, and the islands are formed by connecting adjacent lands in all 8 directions. Despite some islands having a common edge, they are considered separate islands because there is no land connectivity between them.

Example 2

```
Input: grid = [
    ["1", "0", "0", "0", "1"],
    ["0", "1", "0", "1", "0"],
    ["0", "0", "1", "0", "0"],
    ["0", "1", "0", "1", "0"]
]
Output: 1
```

Explanation: In the given grid, there's only one island as all the '1's are connected either horizontally, vertically, or diagonally, forming a single contiguous landmass surrounded by water on all sides.

Intuition

Graph Representation

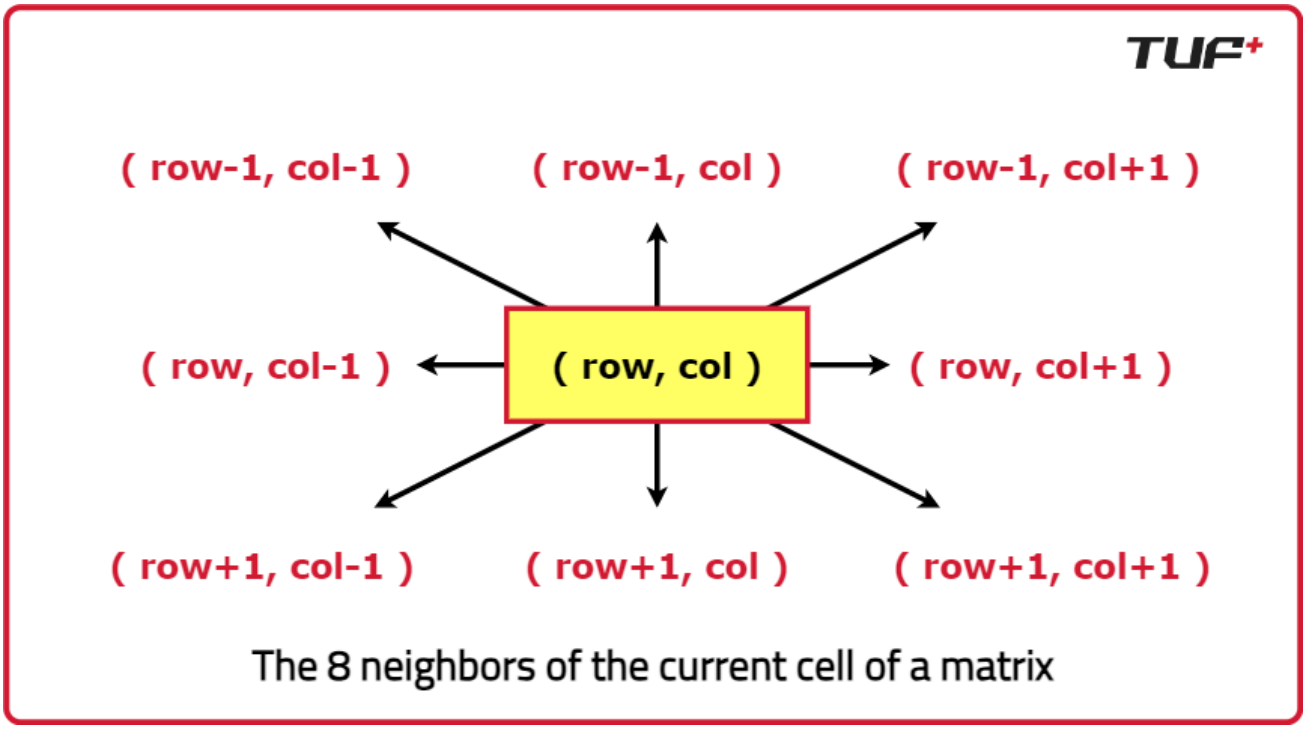
- Think of all cells in the grid as nodes/vertices
- Each cell is connected to its 8 neighbors through edges
- Land cells ('1') that are connected form a connected component (island)

8-Directional Traversal

The 8 neighbors of any cell (row, col) can be accessed using:

- **Row variations**: row-1, row, row+1 (deltaRow: -1, 0, 1)

- **Column variations**: col-1, col, col+1 (deltaCol: -1, 0, 1)



Approach

1. **Initialize**: Determine grid dimensions and create a 2D visited array
2. **Traverse Grid**: Loop through each cell in the grid
3. **Find New Islands**: If cell is land ('1') and not visited, start BFS/DFS
4. **Explore Island**: Use BFS to visit all connected land cells in 8 directions
5. **Count Islands**: Increment counter for each new island found

Implementation

```
def solution(grid):
    n, m = len(grid), len(grid[0])
    visited = [[False] * m for _ in range(n)]

    def bfs(row, col):
        """Perform BFS to explore all connected land cells"""
        visited[row][col] = True
        queue = [(row, col)]

        while queue:
            r, c = queue.pop(0)

            # Check all 8 directions
            for i in range(-1, 2):
                for j in range(-1, 2):
                    adj_row = r + i
                    adj_col = c + j

                    # Check bounds
                    if (adj_row >= 0 and adj_col >= 0 and
                        adj_row < n and adj_col < m):
```

```
                        # If it's land and not visited
                        if (grid[adj_row][adj_col] == '1' and
                            not visited[adj_row][adj_col]):
                            visited[adj_row][adj_col] = True
                            queue.append((adj_row, adj_col))

        count = 0
        for i in range(n):
            for j in range(m):
                if grid[i][j] == '1' and not visited[i][j]:
                    count += 1
                    bfs(i, j)

        return count

# Test the solution
grid = [
    ["1", "0", "0", "0", "1"],
    ["0", "1", "0", "1", "0"],
    ["0", "0", "1", "0", "0"],
    ["0", "1", "0", "1", "0"]
]
print(solution(grid))  # Output: 1
```

Algorithm Details

- **Time Complexity**: O(N × M) where N is rows and M is columns
- **Space Complexity**: O(N × M) for the visited array and queue

Number of Enclaves

Problem Statement

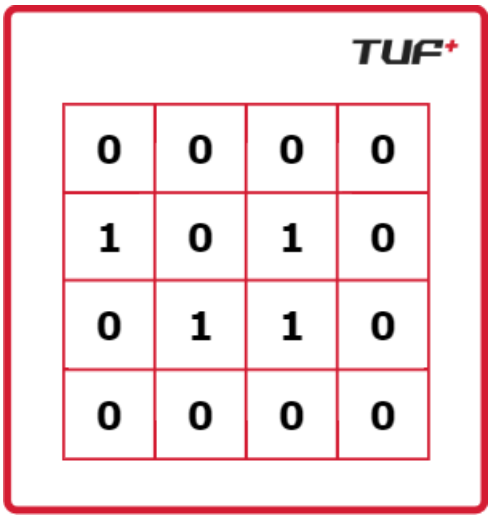
Given an N x M binary matrix grid, where 0 represents a sea cell and 1 represents a land cell. A move consists of walking from one land cell to another adjacent (4-directionally) land cell or walking off the boundary of the grid.

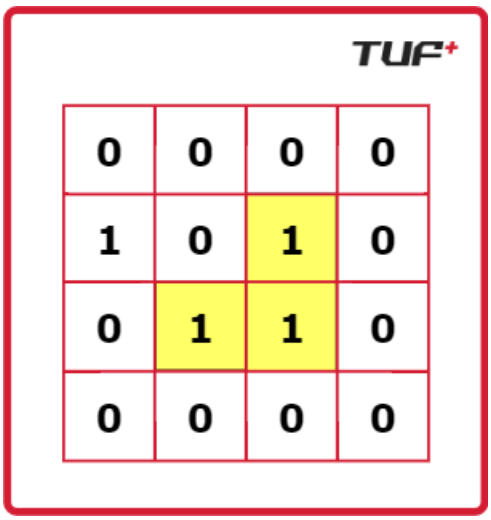
Find the number of land cells in the grid for which we cannot walk off the boundary of the grid in any number of moves.

Examples

Example 1

```
Input: grid = [
    [0, 0, 0, 0],
    [1, 0, 1, 0],
    [0, 1, 1, 0],
    [0, 0, 0, 0]
]
Output: 3
```

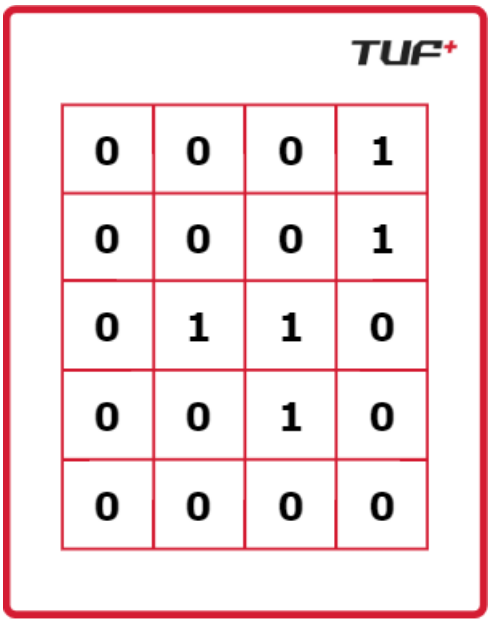


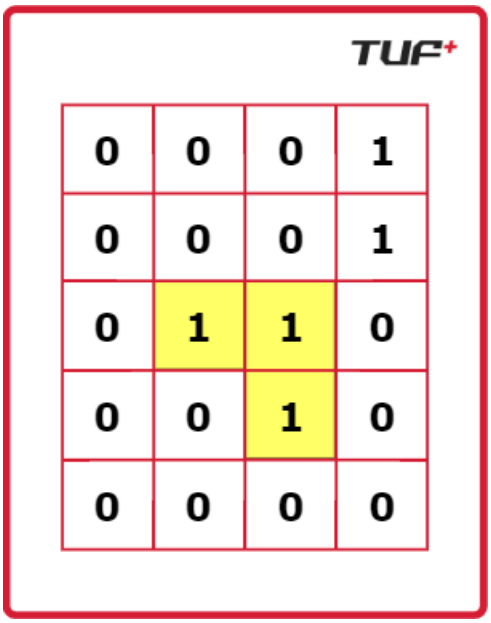


Explanation: The land cells at positions (1,2), (2,1), and (2,2) are completely surrounded and cannot reach the boundary. Therefore, there are 3 enclaves.

Example 2

```
Input: grid = [
    [0, 0, 0, 1],
    [0, 0, 0, 1],
    [0, 1, 1, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 0]
]
Output: 3
```





Explanation: The land cells that cannot reach the boundary form the enclaves. The highlighted cells represent these enclosed land cells.

Intuition

The problem requires finding land cells that are completely surrounded by other land cells and cannot connect to the boundary of the grid by moving in 4 directions (up, down, left, right).

Key Insight: Any land cell directly connected to the boundary or indirectly connected via other land cells should not be counted as an enclave. Therefore, we can:

1. Identify and mark all land cells reachable from the boundary
2. Count the remaining unmarked land cells as enclaves

Approach

1. **Setup**: Create arrays for 4-directional movement and boundary checking

2. **BFS Implementation**: Create a function to traverse connected land cells
3. **Boundary Marking**: Start BFS from all boundary land cells to mark reachable cells
4. **Count Enclaves**: Count all unmarked land cells as they represent enclaves

Implementation

```
def solution(grid):
    n, m = len(grid), len(grid[0])
    visited = [[False] * m for _ in range(n)]

    def bfs(row, col):
        """Perform BFS to mark all land cells connected to boundary"""
        visited[row][col] = True
        queue = [(row, col)]

        while queue:
            r, c = queue.pop(0)

            # Check 4 directions: right, left, up, down
            for i, j in zip([0, 0, -1, 1], [1, -1, 0, 0]):
                adj_row = r + i
                adj_col = c + j

                # Check bounds
                if (adj_row >= 0 and adj_col >= 0 and
                    adj_row < n and adj_col < m):

                    # If it's land and not visited
                    if (grid[adj_row][adj_col] == 1 and
                        not visited[adj_row][adj_col]):
                        visited[adj_row][adj_col] = True
                        queue.append((adj_row, adj_col))

    # Mark all boundary-connected land cells
    for i in range(n):
        for j in range(m):
            # Check if cell is on boundary
            if i == 0 or i == n-1 or j == 0 or j == m-1:
                if grid[i][j] == 1 and not visited[i][j]:
                    bfs(i, j)

    # Count unvisited land cells (enclaves)
    count = 0
    for i in range(n):
        for j in range(m):
            if grid[i][j] == 1 and not visited[i][j]:
                count += 1

    return count

# Test the solution
grid = [
    [0, 0, 0, 0],
    [1, 0, 1, 0],
    [0, 1, 1, 0],
    [0, 0, 0, 0]
]
print(solution(grid))  # Output: 3
```

Optimized Implementation (In-place Modification)

```python
def solution_optimized(grid):
    """
    Modify the grid in-place to avoid extra space for visited array
    """
    n, m = len(grid), len(grid[0])

    def dfs(row, col):
        if (row < 0 or col < 0 or row >= n or col >= m or
            grid[row][col] != 1):
            return

        # Mark as visited by changing to 2
        grid[row][col] = 2

        # Explore 4 directions
        directions = [(0, 1), (0, -1), (-1, 0), (1, 0)]
        for dr, dc in directions:
            dfs(row + dr, col + dc)

    # Mark all boundary-connected land cells
    for i in range(n):
        for j in range(m):
            if i == 0 or i == n-1 or j == 0 or j == m-1:
                if grid[i][j] == 1:
                    dfs(i, j)

    # Count remaining land cells (enclaves)
    count = 0
    for i in range(n):
        for j in range(m):
            if grid[i][j] == 1:
                count += 1

    # Optional: Restore the grid
    for i in range(n):
        for j in range(m):
            if grid[i][j] == 2:
                grid[i][j] = 1

    return count
```

Step-by-Step Visualization

For the first example:

```
Original Grid:          After Boundary BFS:      Enclaves Count:
[0, 0, 0, 0]           [0, 0, 0, 0]            [0, 0, 0, 0]
[1, 0, 1, 0]    →      [V, 0, 1, 0]     →      [-, 0, E, 0]
[0, 1, 1, 0]           [0, 1, 1, 0]            [0, E, E, 0]
[0, 0, 0, 0]           [0, 0, 0, 0]            [0, 0, 0, 0]

V = Visited (boundary-connected)
E = Enclave (3 total)
```

Algorithm Details

- **Time Complexity**: O(N × M) where N is rows and M is columns
- **Space Complexity**: O(N × M) for the visited array and queue
- **Method**: Breadth-First Search (BFS) from boundary cells

Surrounded Regions

Problem Statement

Given a matrix mat of size N x M where every element is either 'O' or 'X'. Replace all 'O' with 'X' that is surrounded by 'X'.

An 'O' (or a set of 'O') is considered to be surrounded by 'X' if there are 'X' at locations just below, above, left, and right of it.

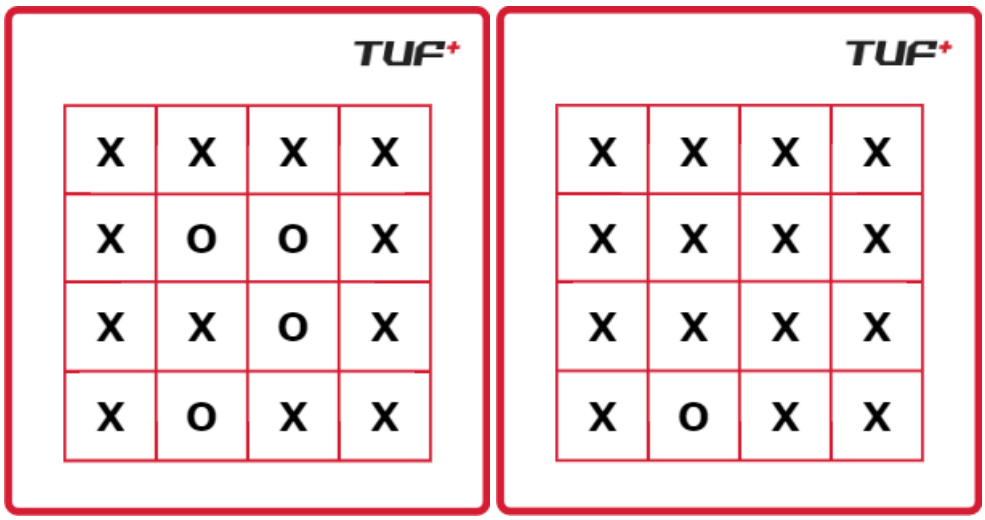
Examples

Example 1

```
Input: mat = [
    ["X", "X", "X", "X"],
    ["X", "O", "O", "X"],
    ["X", "X", "O", "X"],
    ["X", "O", "X", "X"]
]

Output: [
    ["X", "X", "X", "X"],
    ["X", "X", "X", "X"],
    ["X", "X", "X", "X"],
    ["X", "O", "X", "X"]
]
```

Explanation:



- The 'O' cells at positions (1,1), (1,2), and (2,2) are completely surrounded by 'X' cells, so they are replaced with 'X'
- The 'O' at position (3,1) is adjacent to the boundary, so it cannot be completely surrounded and remains unchanged

Example 2

```
Input: mat = [
    ["X", "X", "X"],
    ["X", "O", "X"],
    ["X", "X", "X"]
]

Output: [
    ["X", "X", "X"],
    ["X", "X", "X"],
    ["X", "X", "X"]
]
```

Explanation: The only 'O' cell at position (1,1) is completely surrounded by 'X' cells in all directions, so it is replaced with 'X'.

Intuition

Key Insight: Boundary elements cannot be replaced with 'X' as they are not surrounded by 'X' from all 4 directions. If an 'O' (or a set of 'O') is connected to a boundary 'O', then it cannot be replaced with 'X'.

Strategy: Instead of finding surrounded regions directly, we can:

1. Find all 'O' cells that are connected to the boundary
2. Mark these cells as "safe" (cannot be replaced)
3. Replace all remaining 'O' cells with 'X'

Approach

1. **Initialize**: Create a visited array and direction vectors for 4-directional movement
2. **Boundary Traversal**: Start BFS/DFS from all boundary 'O' cells
3. **Mark Connected**: Mark all 'O' cells connected to boundary as visited (safe)
4. **Replace Surrounded**: Convert all unvisited 'O' cells to 'X'

Implementation

```python
def solution(grid):
    n, m = len(grid), len(grid[0])
    visited = [[False] * m for _ in range(n)]

    def bfs(row, col):
        """BFS to mark all 'O' cells connected to boundary"""
        visited[row][col] = True
        queue = [(row, col)]

        while queue:
```

```
            r, c = queue.pop(0)

            # Check 4 directions: right, left, up, down
            for i, j in zip([0, 0, -1, 1], [1, -1, 0, 0]):
                adj_row = r + i
                adj_col = c + j

                # Check bounds
                if (adj_row >= 0 and adj_col >= 0 and
                    adj_row < n and adj_col < m):

                    # If it's 'O' and not visited
                    if (grid[adj_row][adj_col] == 'O' and
                        not visited[adj_row][adj_col]):
                        visited[adj_row][adj_col] = True
                        queue.append((adj_row, adj_col))

    # Mark all boundary-connected 'O' cells
    for i in range(n):
        for j in range(m):
            # Check if cell is on boundary
            if i == 0 or i == n-1 or j == 0 or j == m-1:
                if grid[i][j] == 'O' and not visited[i][j]:
                    bfs(i, j)

    # Replace all unvisited 'O' cells with 'X'
    for i in range(n):
        for j in range(m):
            if grid[i][j] == 'O' and not visited[i][j]:
                grid[i][j] = 'X'

    return grid

# Test the solution
mat = [
    ["X", "X", "X", "X"],
    ["X", "O", "O", "X"],
    ["X", "X", "O", "X"],
    ["X", "O", "X", "X"]
]
print(solution(mat))
```

Space-Optimized Implementation

```
def solution_optimized(grid):
    """
    Modify the grid in-place to avoid extra space for visited array
    """
    n, m = len(grid), len(grid[0])

    def dfs(row, col):
        if (row < 0 or col < 0 or row >= n or col >= m or
            grid[row][col] != 'O'):
            return

        # Mark as safe by changing to a temporary character
        grid[row][col] = '#'

        # Explore 4 directions
```

```
        directions = [(0, 1), (0, -1), (-1, 0), (1, 0)]
        for dr, dc in directions:
            dfs(row + dr, col + dc)

    # Mark all boundary-connected 'O' cells as safe
    for i in range(n):
        for j in range(m):
            if i == 0 or i == n-1 or j == 0 or j == m-1:
                if grid[i][j] == 'O':
                    dfs(i, j)

    # Process the grid: replace 'O' with 'X' and '#' back to 'O'
    for i in range(n):
        for j in range(m):
            if grid[i][j] == 'O':
                grid[i][j] = 'X'  # Surrounded region
            elif grid[i][j] == '#':
                grid[i][j] = 'O'  # Safe region

    return grid
```

Algorithm Details

- **Time Complexity**: O(N × M) where N is rows and M is columns
- **Space Complexity**: O(N × M) for the visited array and recursion stack

Flood Fill Algorithm - Complete Guide

Problem Statement

An image is represented by a 2-D array of integers, where each integer represents the pixel value of the image. Given a coordinate `(sr, sc)` representing the starting pixel (row and column) of the flood fill, and a pixel value `newColor`, perform a "flood fill" operation on the image.

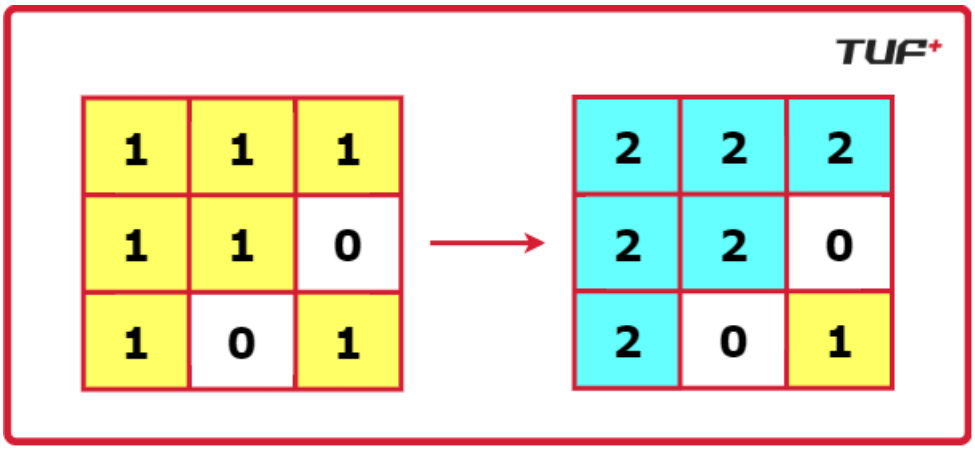
What is Flood Fill?

To perform a flood fill:

1. Consider the starting pixel
2. Find all pixels connected **4-directionally** to the starting pixel that have the **same color** as the starting pixel
3. Continue finding pixels connected 4-directionally to those pixels (also with the same color)
4. Replace the color of all these connected pixels with the `newColor`

Examples

Example 1



Input:

```
image = [[1, 1, 1],
         [1, 1, 0],
         [1, 0, 1]]
sr = 1, sc = 1, newColor = 2
```

Output:

```
[[2, 2, 2],
 [2, 2, 0],
 [2, 0, 1]]
```

Explanation: Starting from position (1, 1) with value 1, all connected pixels with value 1 are changed to 2. The bottom-right corner (2, 2) is not changed because it's not 4-directionally connected to the starting pixel.

Example 2

Input:

```
image = [[0, 1, 0],
         [1, 1, 0],
         [0, 0, 1]]
sr = 2, sc = 2, newColor = 3
```

Output:

```
[[0, 1, 0],
 [1, 1, 0],
 [0, 0, 3]]
```

Explanation: Starting from position (2, 2) with value 1, only this single pixel needs to be changed since no adjacent pixels have the same color.

Intuition

Graph Perspective

- Think of all pixels in the image as **nodes/vertices**
- Each pixel is connected to its 4 neighbors (up, right, down, left) via **edges**
- Use any traversal algorithm (DFS/BFS) to find all neighbors with the same pixel value
- During traversal, change each pixel's value to the new color

Efficient Neighbor Traversal

The 4 neighbors of any cell can be accessed using direction vectors:

```
delRow = [-1, 0, 1, 0]  # up, right, down, left
delCol = [0, 1, 0, -1]  # up, right, down, left
```

Approach

1. **Initialize:** Create a new image to store the flood-filled result (optional - can modify original)
2. **Direction Vectors:** Define vectors for moving up, right, down, and left
3. **Boundary Check:** Create helper function to ensure pixels are within image bounds
4. **Traversal:** Starting from the given pixel, perform DFS/BFS traversal
5. **Color Change:** During traversal, mark all pixels with the same initial color using the new color
6. **Return:** Once traversal terminates, return the flood-filled image

Implementation

BFS Approach

```
def flood_fill(grid, srcRow, srcCol, newColor):
    n, m = len(grid), len(grid[0])
    queue = [(srcRow, srcCol)]
    initColor = grid[srcRow][srcCol]

    # If new color is same as initial color, no change needed
    if newColor == initColor:
        return grid

    # Mark starting pixel with new color
    grid[srcRow][srcCol] = newColor

    # BFS traversal
    while queue:
        r, c = queue.pop(0)

        # Check all 4 directions
        for i, j in zip([0, 0, -1, 1], [1, -1, 0, 0]):
            adj_row = r + i
            adj_col = c + j

            # Check bounds and color match
            if (adj_row >= 0 and adj_col >= 0 and
                adj_row < n and adj_col < m):
                if grid[adj_row][adj_col] == initColor:
                    grid[adj_row][adj_col] = newColor
                    queue.append((adj_row, adj_col))
```

```
    return grid

# Test the algorithm
grid = [[1, 1, 1],
        [1, 1, 0],
        [1, 0, 1]]
sr, sc = 1, 1
newColor = 2
result = flood_fill(grid, sr, sc, newColor)
print(result)
```

DFS Approach

```
def flood_fill_dfs(grid, srcRow, srcCol, newColor):
    n, m = len(grid), len(grid[0])
    initColor = grid[srcRow][srcCol]

    if newColor == initColor:
        return grid

    def dfs(row, col):
        # Check bounds and color match
        if (row < 0 or row >= n or col < 0 or col >= m or
            grid[row][col] != initColor):
            return

        # Change color
        grid[row][col] = newColor

        # Recursively fill 4 directions
        dfs(row - 1, col)  # up
        dfs(row + 1, col)  # down
        dfs(row, col - 1)  # left
        dfs(row, col + 1)  # right

    dfs(srcRow, srcCol)
    return grid
```

Algorithm Details

- **Time Complexity:** O(n × m) where n and m are the dimensions of the image
- **Space Complexity:**
  - BFS: O(n × m) for the queue in worst case
  - DFS: O(n × m) for the recursion stack in worst case

Rotten Oranges

Problem Statement

Given an n x m grid, where each cell has the following values:

- **2** - represents a rotten orange
- **1** - represents a fresh orange
- **0** - represents an empty cell

Every minute, if a fresh orange is adjacent to a rotten orange in 4-direction (upward, downwards, right, and left) it becomes rotten.

Return the minimum number of minutes required such that none of the cells has a fresh orange. If it's not possible, return -1.

Examples

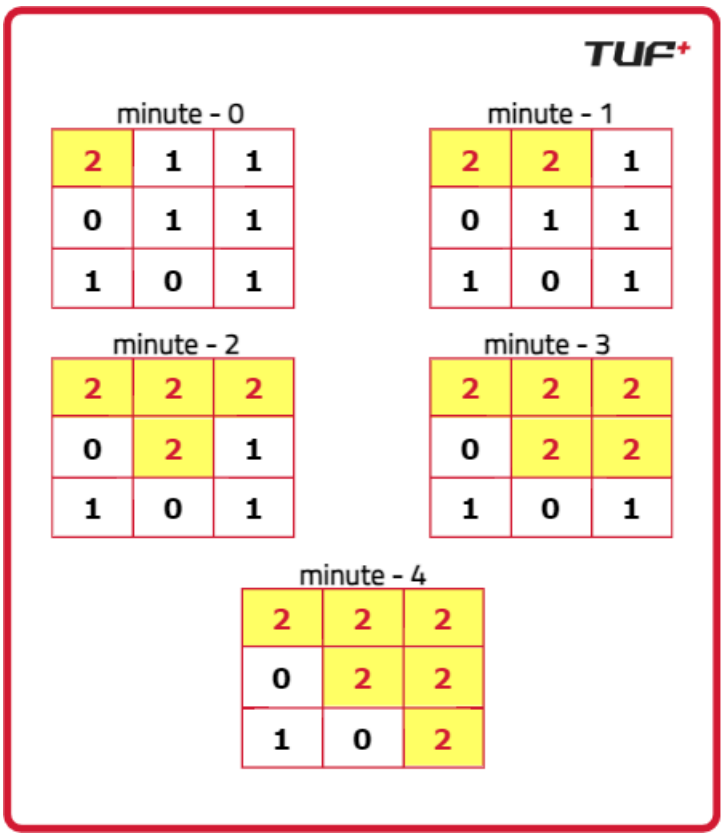
Example 1

Input:

```
grid = [[2, 1, 1],
        [0, 1, 1],
        [1, 0, 1]]
```

Output:

```
-1
```



Explanation: Orange at (2,0) cannot be rotten because it's not connected to any rotten orange through adjacent cells.

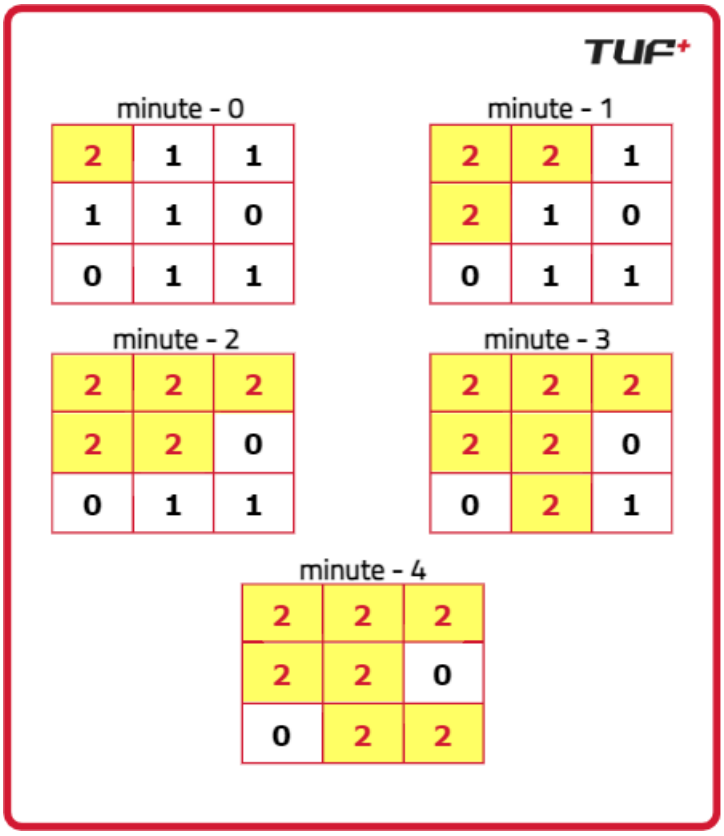
Example 2

Input:

```
grid = [[2, 1, 1],
        [1, 1, 0],
        [0, 1, 1]]
```

Output:

```
4
```



Explanation: - Minute 0: Initial state with one rotten orange at (0,0) - Minute 1: Orange at (0,1) and (1,0) become rotten - Minute 2: Orange at (0,2) and (1,1) become rotten - Minute 3: Orange at (2,1) becomes rotten - Minute 4: Orange at (2,2) becomes rotten

Intuition

The key insight is that for each rotten orange, fresh oranges in its 4 directions will become rotten in the next minute. This creates a **multi-source BFS** problem where:

- Each minute represents a **level** in BFS traversal
- All oranges at the same level rot simultaneously
- We need to track the maximum time taken for any orange to rot

Multi-Source BFS Strategy:

1. Start with all initially rotten oranges in the queue

2. Process level by level (minute by minute)
3. For each rotten orange, spread rot to adjacent fresh oranges
4. Continue until no more oranges can be rotten

Approach

1. **Initialize:** Create a queue for BFS and count total fresh oranges
2. **Find Initial Rotten:** Traverse grid to find all initially rotten oranges and add them to queue with time = 0
3. **BFS Traversal:** Process queue level by level:
   - For each rotten orange, check its 4 neighbors
   - If neighbor is fresh, make it rotten and add to queue with incremented time
   - Track maximum time encountered
4. **Validation:** Check if all fresh oranges became rotten
5. **Return:** Return maximum time if all rotten, else -1

Implementation

```
def solution(grid):
    n, m = len(grid), len(grid[0])
    queue = []
    fresh = 0
    ans = 0

    # Find all initially rotten oranges and count fresh ones
    for i in range(n):
        for j in range(m):
            if grid[i][j] == 2:
                queue.append((i, j, 0))  # (row, col, time)
            elif grid[i][j] == 1:
                fresh += 1

    # BFS to spread rot
    while queue:
        r, c, time = queue.pop(0)

        # Check all 4 directions
        for i, j in zip([0, 0, -1, 1], [1, -1, 0, 0]):
            adj_row = r + i
            adj_col = c + j

            # Check bounds
            if (adj_row >= 0 and adj_col >= 0 and
                adj_row < n and adj_col < m):

                # If adjacent cell has fresh orange
                if grid[adj_row][adj_col] == 1:
                    grid[adj_row][adj_col] = 2  # Make it rotten
                    fresh -= 1  # Decrease fresh count
                    queue.append((adj_row, adj_col, time + 1))
                    ans = max(ans, time + 1)  # Update max time

    # Check if all oranges are rotten
    if fresh > 0:
        return -1
    return ans
```

```
# Test the algorithm
grid = [[2, 1, 1],
        [1, 1, 0],
        [0, 1, 1]]
print(solution(grid))  # Output: 4
```

Algorithm Details

- **Time Complexity:** O(n × m) where n and m are the dimensions of the grid
- **Space Complexity:** O(n × m) for the queue in worst case (when all cells are rotten initially)

Distance of Nearest Cell Having One

Problem Statement

Given a binary grid of N x M. Find the distance of the nearest 1 in the grid for each cell.

The distance is calculated as |i1 - i2| + |j1 - j2|, where i1, j1 are the row number and column number of the current cell, and i2, j2 are the row number and column number of the nearest cell having value 1.

Examples

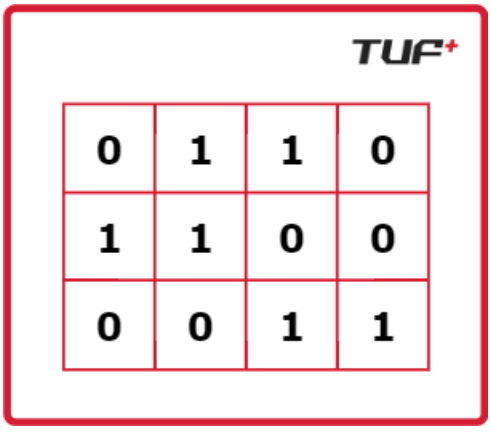
Example 1

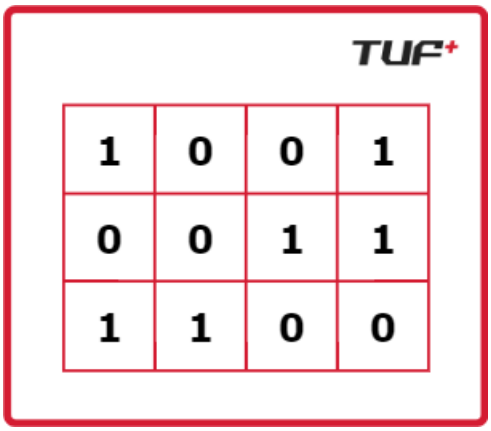
Input:

```
grid = [[0, 1, 1, 0],
        [1, 1, 0, 0],
        [0, 0, 1, 1]]
```

Output:

```
[[1, 0, 0, 1],
 [0, 0, 1, 1],
 [1, 1, 0, 0]]
```





Explanation: - 0's at (0,0), (0,3), (1,2), (1,3), (2,0) and (2,1) are at a distance of 1 from 1's at (0,1), (0,2), (0,2), (2,3), (1,0) and (1,1) respectively.

Example 2

Input:

```
grid = [[1, 0, 1],
        [1, 1, 0],
        [1, 0, 0]]
```

Output:

```
[[0, 1, 0],
 [0, 0, 1],
 [0, 1, 2]]
```

Explanation:

- 0's at (0,1), (1,2), (2,1) and (2,2) are at a distance of 1, 1, 1 and 2 from 1's at (0,0), (0,2), (2,0) and (1,1) respectively.

Intuition

To find the nearest 1 in the grid for each cell, the **multi-source BFS** algorithm is perfect. The key insight is:

- Start BFS from all cells containing '1' simultaneously
- BFS naturally explores cells in order of increasing distance
- The first time we reach any cell, it's guaranteed to be via the shortest path

Why BFS over DFS? BFS ensures that when we first visit a cell, we do so via the shortest possible path. This is because BFS explores all cells at distance 1 before moving to distance 2, and so on.

Multi-Source BFS Strategy:

1. Add all '1' cells to the queue with distance 0
2. Process level by level (distance by distance)
3. For each cell, explore its 4 neighbors
4. First time we reach a '0' cell determines its minimum distance

Approach

1. **Initialize:** Create a queue for BFS and mark all '1' cells as visited
2. **Multi-Source Start:** Add all '1' cells to queue with distance 0
3. **BFS Traversal:** Process queue level by level:
   - For each cell, check its 4 neighbors
   - If neighbor is unvisited '0', mark it with current distance + 1
   - Add the neighbor to queue for further exploration
4. **Result:** The grid now contains minimum distances for each cell

Implementation

```
def solution(grid):
    n, m = len(grid), len(grid[0])
    queue = []

    # Add all '1' cells to queue and mark them as visited
    for i in range(n):
        for j in range(m):
            if grid[i][j] == 1:
                queue.append((i, j, 0))  # (row, col, distance)
                grid[i][j] = -1  # Mark as visited (will be restored to 0 later)

    # BFS to find minimum distances
    while queue:
        r, c, time = queue.pop(0)

        # Check all 4 directions
        for i, j in zip([0, 0, -1, 1], [1, -1, 0, 0]):
            adj_row = r + i
            adj_col = c + j

            # Check bounds
            if (adj_row >= 0 and adj_col >= 0 and
                adj_row < n and adj_col < m):

                # If unvisited '0' cell
                if grid[adj_row][adj_col] == 0:
                    grid[adj_row][adj_col] = time + 1
                    queue.append((adj_row, adj_col, time + 1))

    # Restore '1' cells (marked as -1) back to distance 0
    for i in range(n):
        for j in range(m):
            if grid[i][j] == -1:
                grid[i][j] = 0

    return grid

# Test the algorithm
grid = [[0, 1, 1, 0],
        [1, 1, 0, 0],
        [0, 0, 1, 1]]
print(solution(grid))
```

Alternative Implementation (Using Separate Distance Matrix)

```python
def solution_separate_matrix(grid):
    n, m = len(grid), len(grid[0])
    distance = [[float('inf')] * m for _ in range(n)]
    queue = []

    # Initialize distance matrix and queue
    for i in range(n):
        for j in range(m):
            if grid[i][j] == 1:
                distance[i][j] = 0
                queue.append((i, j))

    directions = [(0, 1), (0, -1), (-1, 0), (1, 0)]

    # BFS traversal
    while queue:
        r, c = queue.pop(0)

        for dr, dc in directions:
            nr, nc = r + dr, c + dc

            if (0 <= nr < n and 0 <= nc < m and
                distance[nr][nc] > distance[r][c] + 1):
                distance[nr][nc] = distance[r][c] + 1
                queue.append((nr, nc))

    return distance
```

Algorithm Details

- **Time Complexity:** O(n × m) where n and m are the dimensions of the grid
- **Space Complexity:** O(n × m) for the queue in worst case

Cycle Detection in Undirected Graphs

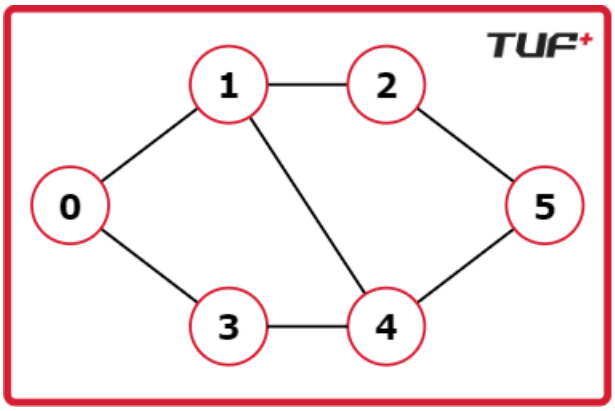
Problem Statement

Given an undirected graph with V vertices labeled from 0 to V-1, determine if the graph contains any cycles. The graph is represented using an adjacency list where `adj[i]` lists all nodes connected to node `i`.

Note: The graph does not contain any self-edges (edges where a vertex is connected to itself).

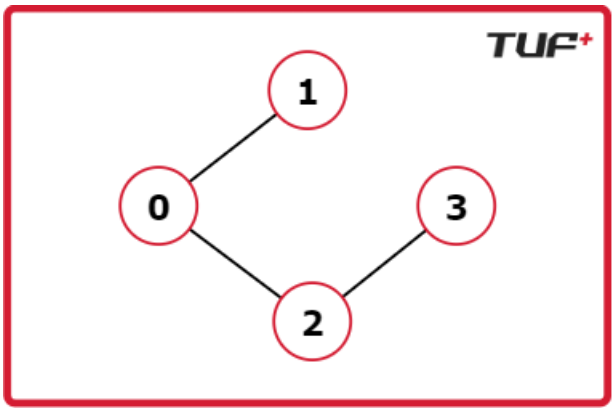
Examples

Example 1




```
Input: V = 6, adj = [[1, 3], [0, 2, 4], [1, 5], [0, 4], [1, 3, 5], [2, 4]]
Output: True
Explanation: The graph contains a cycle: 0 → 1 → 2 → 5 → 4 → 1
```

Example 2




```
Input: V = 4, adj = [[1, 2], [0], [0, 3], [2]]
Output: False
Explanation: The graph does not contain any cycles
```

Intuition

In an undirected graph, a cycle is formed when a path exists that returns to the starting vertex without reusing an edge. The key insight is:

During traversal, if we encounter a vertex that has already been visited and is not the immediate parent of the current vertex, a cycle exists.

This works because:

- If we visit a node that's already been visited
- AND it's not our immediate parent (which would just be backtracking)
- Then we've found an alternative path to reach that node, forming a cycle

Approach 1: Depth-First Search (DFS)

Algorithm

1. Create an adjacency list representation of the graph using sets
2. Initialize a visited dictionary to track visited nodes
3. For each unvisited node, perform DFS
4. During DFS, if we encounter a visited node that's not the parent, return True (cycle found)
5. If no cycles found after checking all components, return False

Implementation

```
#Using depth first search
def solution(n,edges):
    nodes=[i for i in range(n)]
    graph={node:set() for node in nodes}
    for i,j in edges:
        graph[i].add(j)
        graph[j].add(i)
    def dfs(node,parent):
        visited[node]=True
        for adj_node in graph[node]:
            if(not visited[adj_node]):
                check=dfs(adj_node,node)
                if(check==True):
                    return True
            else:
                if(adj_node!=parent):
                    return True
        return False
    visited={i:False for i in nodes}
    for node in nodes:
        if(not visited[node]):
            check=dfs(node,-1)
            if(check==True):
                return True
    return False
```

Time Complexity: O(V + E)

- V: Number of vertices
- E: Number of edges
- We visit each vertex once and each edge twice (once from each endpoint)

Space Complexity: O(V)

- Visited dictionary: O(V)
- Recursion stack: O(V) in worst case
- Adjacency list with sets: O(V + E)

Approach 2: Breadth-First Search (BFS)

Algorithm

1. Create an adjacency list representation of the graph using sets
2. Initialize a visited dictionary to track visited nodes
3. For each unvisited node, perform BFS
4. Use a deque to store (node, parent) pairs
5. During BFS, if we encounter a visited node that's not the parent, return True
6. If no cycles found after checking all components, return False

Implementation

```
#Using breadth first search

from collections import deque
def solution(n,edges):
    nodes=[i for i in range(n)]
    graph={node:set() for node in nodes}
    for i,j in edges:
        graph[i].add(j)
        graph[j].add(i)

    def bfs(node):
        visited[node]=True
        queue=deque()
        queue.append((node,-1))
        while queue:
            node,parent=queue.popleft()
            for adj_node in graph[node]:
                if(not visited[adj_node]):
                    queue.append((adj_node,node))
                    visited[adj_node]=True
                else:
                    if(adj_node!=parent):
                        return True
        return False

    visited={i:False for i in nodes}
    for node in nodes:
        if(not visited[node]):
            check=bfs(node)
            if(check==True):
                return True
    return False
```

Time Complexity: O(V + E)

- Same as DFS approach

Space Complexity: O(V)

- Visited dictionary: O(V)
- Queue: O(V) in worst case

- Adjacency list with sets: O(V + E)

Test Example

```
n=6
edges=[(0, 1), (0, 3), (1, 2), (1, 4), (2, 5), (3, 4), (4, 5)]
print(solution(n,edges))
```

Bipartite Graph Detection

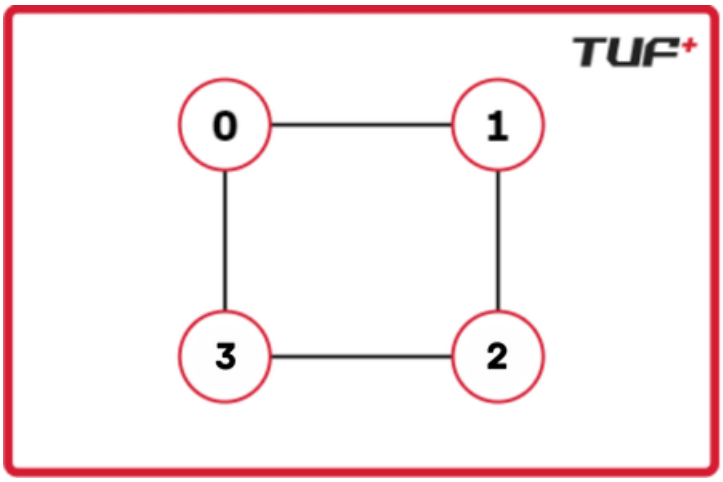
Problem Statement

Given an undirected graph with V vertices labeled from 0 to V-1, represented using an adjacency list where `adj[i]` lists all nodes connected to node `i`, determine if the graph is bipartite or not.

Definition: A graph is bipartite if the nodes can be partitioned into two independent sets A and B such that every edge in the graph connects a node in set A and a node in set B.

Examples

Example 1



- **Input**: V=4, adj = [[1,3],[0,2],[1,3],[0,2]] - **Output**: True - **Explanation**: The given graph is bipartite since we can partition the nodes into two sets: {0, 2} and {1, 3}.

Example 2

- **Input**: V=4, adj = [[1,2,3],[0,2],[0,1,3],[0,2]]
- **Output**: False
- **Explanation**: The graph is not bipartite. If we attempt to partition the nodes into two sets, we encounter an edge that connects two nodes within the same set, which violates the bipartite property.

Intuition

A bipartite graph is a graph that can be colored using 2 colors such that no adjacent nodes have the same color. Key insights:

- Any linear graph with no cycle is always a bipartite graph
- With a cycle, any graph with an even cycle length can also be a bipartite graph
- Any graph with an odd cycle length can never be a bipartite graph

Approach: To check if the given graph is bipartite, we can verify if the nodes can be colored alternatingly. If alternate coloring of nodes is possible, the graph is bipartite, otherwise not.

Solution 1: Using Depth First Search (DFS)

Approach Steps:

1. **Build Graph**: Create adjacency list representation from edge list
2. **Initialize Data Structures**: Create visited and color tracking dictionaries
3. **DFS Traversal**: For each unvisited node, start DFS with color 0
4. **Color Assignment**: Mark current node as visited and assign given color
5. **Recursive Exploration**: For each adjacent node:
    - If unvisited: recursively call DFS with opposite color
    - If visited: check for color conflict
6. **Conflict Detection**: Return False if adjacent nodes have same color
7. **Component Handling**: Process all disconnected components

Code Implementation:

```
def solution(n,edges):
    nodes=[i for i in range(n)]
    graph={node:set() for node in nodes}
    for i,j in edges:
        graph[i].add(j)
        graph[j].add(i)

    def dfs(node,current_color):
        visited[node]=True
        color[node]=current_color
        for adj_node in graph[node]:
            if(not visited[adj_node]):
                check=dfs(adj_node,int(not current_color))
                if(check==False):
                    return False
            else:
                if(color[adj_node]==current_color):
                    return False
        return True

    visited={i:False for i in nodes}
    color={i:-1 for i in nodes}
    for node in nodes:
        if(not visited[node]):
            check=dfs(node,0)
```

```
            if(check==False):
                return False
    return True
```

Complexity Analysis:

- **Time Complexity**: O(V + E)
  - Each vertex is visited exactly once: O(V)
  - Each edge is traversed exactly twice (once from each endpoint): O(E)
  - Total: O(V + E)
- **Space Complexity**: O(V)
  - Visited dictionary: O(V)
  - Color dictionary: O(V)
  - Recursion stack: O(V) in worst case (linear graph)
  - Graph storage: O(V + E) but not counted as it's input representation

Solution 2: Using Breadth First Search (BFS)

Approach Steps:

1. **Build Graph**: Create adjacency list representation from edge list
2. **Initialize Data Structures**: Create visited and color tracking dictionaries
3. **BFS Traversal**: For each unvisited node, start BFS with color 0
4. **Queue Processing**: Initialize queue with starting node and mark as visited
5. **Level-by-Level Exploration**: While queue is not empty:
   - Dequeue current node
   - For each adjacent node:
     - If unvisited: enqueue, mark visited, assign opposite color
     - If visited: check for color conflict
6. **Conflict Detection**: Return False if adjacent nodes have same color
7. **Component Handling**: Process all disconnected components

Code Implementation:

```
from collections import deque

def solution(n,edges):
    nodes=[i for i in range(n)]
    graph={node:set() for node in nodes}
    for i,j in edges:
        graph[i].add(j)
        graph[j].add(i)

    def bfs(node):
        visited[node]=True
        queue=deque()
        queue.append(node)
        color[node]=0
        while queue:
            node=queue.popleft()
            for adj_node in graph[node]:
```

```
                if(not visited[adj_node]):
                    queue.append(adj_node)
                    visited[adj_node]=True
                    color[adj_node]=int(not color[node])
                else:
                    if(color[node]==color[adj_node]):
                        return False
        return True

    visited={i:False for i in nodes}
    color={i:-1 for i in nodes}
    for node in nodes:
        if(not visited[node]):
            check=bfs(node)
            if(check==False):
                return False
    return True
```

Complexity Analysis:

- **Time Complexity**: O(V + E)
  - Each vertex is visited exactly once: O(V)
  - Each edge is processed exactly twice (once from each endpoint): O(E)
  - Queue operations (enqueue/dequeue): O(1) per operation
  - Total: O(V + E)
- **Space Complexity**: O(V)
  - Visited dictionary: O(V)
  - Color dictionary: O(V)
  - Queue storage: O(V) in worst case (when all nodes are at same level)
  - Graph storage: O(V + E) but not counted as it's input representation

Topological Sort or Kahn's Algorithm

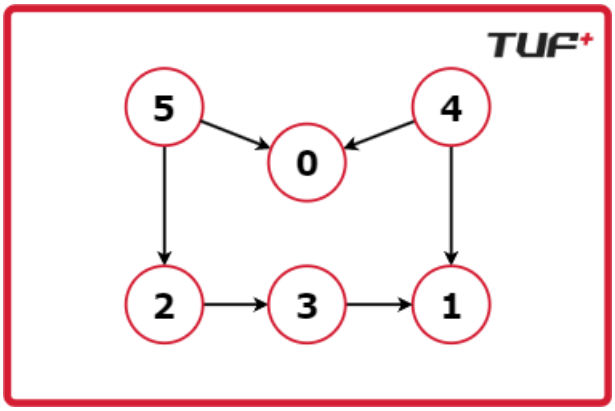
Problem Statement

Given a Directed Acyclic Graph (DAG) with V vertices labeled from 0 to V-1. The graph is represented using an adjacency list where adj[i] lists all nodes connected to node. Find any Topological Sorting of that Graph.

In topological sorting, node u will always appear before node v if there is a directed edge from node u towards node v (u -> v).

The Output will be your topological sort list.

Examples



Example 1:

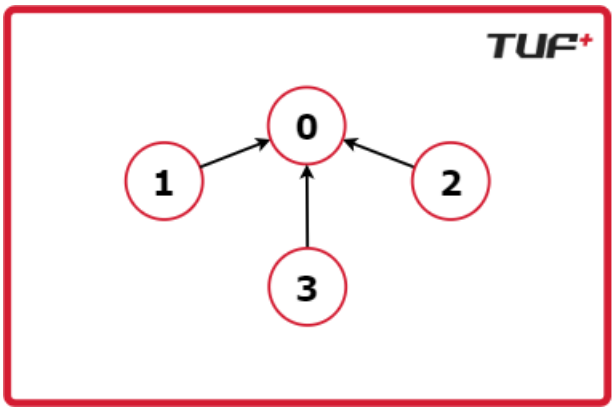
Input: V = 6, adj = [[], [], [3], [1], [0,1], [0,2]] **Output:** [5, 4, 2, 3, 1, 0]

Explanation: A graph may have multiple topological sortings. The result is one of them. The necessary conditions for the ordering are:

- According to edge 5 -> 0, node 5 must appear before node 0 in the ordering.
- According to edge 4 -> 0, node 4 must appear before node 0 in the ordering.
- According to edge 5 -> 2, node 5 must appear before node 2 in the ordering.
- According to edge 2 -> 3, node 2 must appear before node 3 in the ordering.
- According to edge 3 -> 1, node 3 must appear before node 1 in the ordering.
- According to edge 4 -> 1, node 4 must appear before node 1 in the ordering.

The above result satisfies all the necessary conditions. [4, 5, 2, 3, 1, 0] is also one such topological sorting that satisfies all the conditions.

Example 2:



Input: V = 4, adj = [[], [0], [0], [0]] **Output:** [3, 2, 1, 0]

Explanation: The necessary conditions for the ordering are:

- For edge 1 -> 0 node 1 must appear before node 0.
- For edge 2 -> 0 node 2 must appear before node 0.
- For edge 3 -> 0 node 3 must appear before node 0.

Example 3:

Input: V = 3, adj = [[1], [2], []] **Output:** [0, 1, 2]

Solution 1: Using Depth First Search (DFS)

Intuition

Think of it like getting dressed - you must put on your underwear before your pants, and socks before shoes.

In DFS approach: We go as deep as possible first (like checking what you need to wear under your shirt), and only when we've handled everything that depends on us, we add ourselves to the final order. It's like saying "I can only be placed in the final order after all my dependencies are done."

We use a stack because the last person to finish their dependencies should be first in our final answer - just like when you finish getting dressed, the last thing you put on is often the first thing people see.

Approach Steps

1. Initialize a visited dictionary to track processed nodes
2. Create an empty stack to store the topological order
3. For each unvisited node, perform DFS:
    - Mark the current node as visited
    - Recursively visit all adjacent nodes that haven't been visited
    - After processing all adjacent nodes, push the current node to the stack
4. Pop all elements from the stack to get the topological order

Code

```
# using depth first search
def solution(nodes,adj_list):
    graph=adj_list
```

```
    visited={i:False for i in nodes}  # Track visited nodes
    stack=[]  # Stack to store topological order

    def dfs(node):
        visited[node]=True  # Mark current node as visited
        for adj_node in graph[node]:  # Visit all adjacent nodes
            if(not visited[adj_node]):
                dfs(adj_node)  # Recursive DFS call
        stack.append(node)  # Add node to stack after processing all dependencies

    # Start DFS from all unvisited nodes
    for i in nodes:
        if not visited[i]:
            dfs(i)

    # Pop from stack to get topological order
    ans=[]
    while stack:
        ans.append(stack.pop())
    return ans
```

Time and Space Complexity

- **Time Complexity:** O(V + E) where V is the number of vertices and E is the number of edges
- **Space Complexity:** O(V) for the visited dictionary, stack, and recursion stack

Solution 2: Using Kahn's Algorithm (BFS approach)

Intuition

Think of it like taking courses in college - you can't take Advanced Math until you've completed Basic Math first.

In Kahn's algorithm: We start with courses that have no prerequisites (in-degree = 0), take them first, and then see what new courses become available. It's like saying "What can I do right now?" and then updating the list as we complete each task.

We use a queue because we want to process things in the order they become available - first come, first served for things that are ready to be done.

Approach Steps

1. Calculate the in-degree (number of incoming edges) for each node
2. Add all nodes with in-degree 0 to a queue
3. While the queue is not empty:
    - Remove a node from the queue and add it to the result
    - For each adjacent node, decrease its in-degree by 1
    - If the adjacent node's in-degree becomes 0, add it to the queue
4. Return the result list

Code

```
# using depth first search
from collections import deque
def solution(nodes,graph):
```

```
    # Calculate in-degree for each node
    inDegree={i:0 for i in nodes}
    for node,adj_nodes in graph.items():
        for i in adj_nodes:
            inDegree[i]+=1

    # Add all nodes with in-degree 0 to queue
    queue=deque()
    for node,degree in inDegree.items():
        if(degree==0):
            queue.append(node)

    ans=[]
    while queue:
        node=queue.popleft()  # Process node with no dependencies
        ans.append(node)

        # Reduce in-degree of adjacent nodes
        for adj_node in graph[node]:
            inDegree[adj_node]-=1
            if(inDegree[adj_node]==0):  # If no more dependencies, add to queue
                queue.append(adj_node)
    return ans
```

Time and Space Complexity

- **Time Complexity:** O(V + E) where V is the number of vertices and E is the number of edges
- **Space Complexity:** O(V) for the in-degree dictionary, queue, and result list

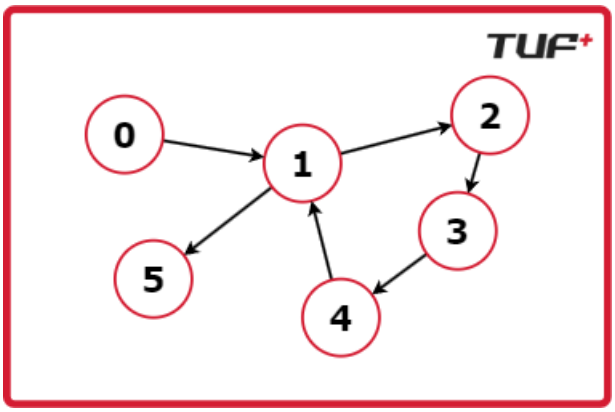
Detect a Cycle in a Directed Graph

Problem Statement

Given a directed graph with V vertices labeled from 0 to V-1. The graph is represented using an adjacency list where adj[i] lists all nodes connected to node. Determine if the graph contains any cycles.

Examples

Example 1:



Input: V = 6, adj = [[1], [2, 5], [3], [4], [1], []] **Output:** True

Explanation: The graph contains a cycle: 1 -> 2 -> 3 -> 4 -> 1.

Example 2:

Input: V = 4, adj = [[1,2], [2], [], [0,2]] **Output:** False

Explanation: The graph does not contain a cycle.

Solution 1: Using Depth First Search (DFS)

Intuition

Think of it like walking through a maze where some paths are one-way streets. A cycle exists if you can start walking from any point and eventually come back to a place you've already been to in your current journey.

The key insight: It's not enough to just remember places you've visited before - you need to remember which places are part of your current path. If you meet someone who's already on your current path, you've found a cycle! But if you meet someone who visited earlier via a different route, that's okay.

It's like leaving breadcrumbs on your current path - if you find your own breadcrumbs, you're going in circles!

Approach Steps

1. Keep track of two things for each node:
    - `visited` : Have we ever been to this node?
    - `pathVisited` : Is this node part of our current path?
2. For each unvisited node, start a DFS:
    - Mark the current node as visited and add it to current path
    - Visit all adjacent nodes
    - If we find an unvisited adjacent node, recursively check it
    - If we find a node that's already in our current path, we found a cycle!
    - Before returning, remove the current node from the path (backtrack)
3. If any DFS finds a cycle, return True; otherwise return False

Code

```
# using depth first search
def solution(nodes,adj_list):
    graph=adj_list
    visited={i:False for i in nodes}  # Track all visited nodes
    pathVisited={i:False for i in nodes}  # Track nodes in current path

    def dfs(node):
        visited[node]=True  # Mark as visited
        pathVisited[node]=True  # Add to current path
        for adj_node in graph[node]:  # Check all adjacent nodes
            if(not visited[adj_node]):  # If not visited, explore recursively
                check=dfs(adj_node)
                if(check==True):  # If cycle found in recursion, return True
                    return True
            else:  # If already visited
```

```
                if(pathVisited[adj_node]):  # If in current path, cycle found!
                    return True
            pathVisited[node]=False  # Remove from current path (backtrack)
            return False

        # Check each unvisited node
        for i in nodes:
            if not visited[i]:
                check=dfs(i)
                if(check==True):  # If cycle found, return True
                    return True
        return False  # No cycle found
```

Time and Space Complexity

- **Time Complexity:** O(V + E) where V is the number of vertices and E is the number of edges
- **Space Complexity:** O(V) for the visited arrays and recursion stack

Solution 2: Using Topological Sort (Kahn's Algorithm)

Intuition

Think of it like arranging tasks where some tasks must be done before others. If you can arrange ALL tasks in a proper order where no task depends on itself (directly or indirectly), then there's no cycle.

But if you can't arrange all tasks because some tasks are waiting for each other in a circle (like "Task A needs Task B, Task B needs Task C, Task C needs Task A"), then there's a cycle!

The clever trick: Use topological sort to try arranging all nodes. If we can arrange all V nodes, no cycle exists. If we can't arrange all nodes (some are stuck waiting), a cycle exists.

Approach Steps

1. Use Kahn's algorithm to perform topological sort:
    - Calculate in-degree for each node
    - Add all nodes with in-degree 0 to queue
    - Process nodes one by one, reducing in-degree of their neighbors
    - Keep track of how many nodes we successfully processed
2. If we processed all V nodes, no cycle exists (return False)
3. If we couldn't process all nodes, a cycle exists (return True)

Code

```
from collections import deque

def solution(nodes,graph):
    def topologicalSort(node,graph):
        # Calculate in-degree for each node
        inDegree={i:0 for i in nodes}
        for node,adj_nodes in graph.items():
            for i in adj_nodes:
                inDegree[i]+=1

        # Add all nodes with in-degree 0 to queue
```

```
        queue=deque()
        for node,degree in inDegree.items():
            if(degree==0):
                queue.append(node)

        ans=[]
        while queue:
            node=queue.popleft()  # Process node with no dependencies
            ans.append(node)

            # Reduce in-degree of adjacent nodes
            for adj_node in graph[node]:
                inDegree[adj_node]-=1
                if(inDegree[adj_node]==0):  # If no more dependencies, add to queue
                    queue.append(adj_node)
        return ans

    topoSort=topologicalSort(nodes,graph)
    if(len(topoSort)==len(nodes)):  # If all nodes processed, no cycle
        return False
    return True  # If some nodes couldn't be processed, cycle exists
```

Time and Space Complexity

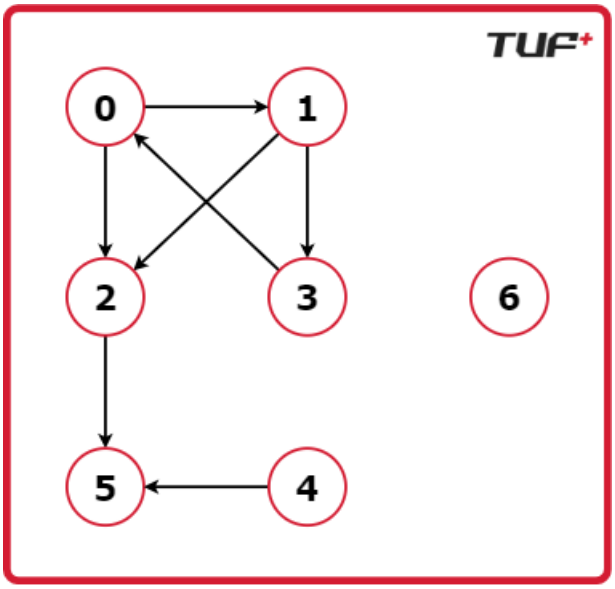
- **Time Complexity:** O(V + E) where V is the number of vertices and E is the number of edges
- **Space Complexity:** O(V) for the in-degree dictionary, queue, and result list

Find Eventual Safe States

Problem Statement

Given a directed graph with V vertices labeled from 0 to V-1. The graph is represented using an adjacency list where adj[i] lists all nodes adjacent to node i, meaning there is an edge from node i to each node in adj[i]. A node is a terminal node if there are no outgoing edges. A node is a safe node if every possible path starting from that node leads to a terminal node. Return an array containing all the safe nodes of the graph in ascending order.

Examples



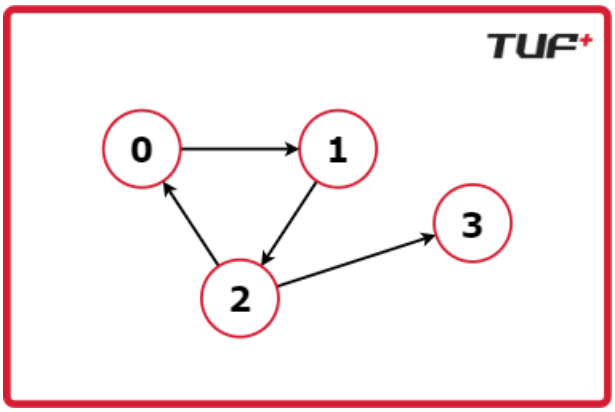
Example 1:

Input: V = 7, adj = [[1,2], [2,3], [5], [0], [5], [], []] **Output:** [2, 4, 5, 6]

Explanation:

- From node 0: two paths are there 0->2->5 and 0->1->3->0. The second path does not end at a terminal node. So it is not a safe node.
- From node 1: two paths exist: 1->3->0->1 and 1->2->5. But the first one does not end at a terminal node. So it is not a safe node.
- From node 2: only one path: 2->5 and 5 is a terminal node. So it is a safe node.
- From node 3: two paths: 3->0->1->3 and 3->0->2->5 but the first path does not end at a terminal node. So it is not a safe node.
- From node 4: Only one path: 4->5 and 5 is a terminal node. So it is also a safe node.
- From node 5: It is a terminal node. So it is a safe node as well.
- From node 6: It is a terminal node. So it is a safe node as well.

Example 2:



Input: V = 4, adj = [[1], [2], [0,3], []] **Output:** [3]

Explanation: Node 3 itself is a terminal node and it is a safe node as well. But all the paths from other nodes do not lead to a terminal node. So they are excluded from the answer.

Solution 1: Using Topological Sort (Kahn's Algorithm)

Intuition

Think of it like a river system flowing to the ocean. Safe nodes are like water sources that will eventually reach the ocean (terminal nodes) no matter which path they take. Unsafe nodes are like water sources that get stuck in whirlpools (cycles) and never reach the ocean.

The trick: Instead of following water downstream, we trace backwards from the ocean! We reverse all the rivers (edges) so that terminal nodes become sources, and then we see which original nodes we can reach by flowing backwards. If we can reach a node by flowing backwards from terminal nodes, it means all paths from that node lead to terminal nodes.

Approach Steps

1. **Reverse the graph**: Change all edges from A→B to B→A
2. **Apply topological sort on reversed graph**:
   - Terminal nodes (originally outdegree 0) become sources (indegree 0 in reversed graph)
   - Start from these terminal nodes and work backwards
   - Only nodes reachable from terminal nodes will be processed
3. **Sort the result**: Return all processed nodes in ascending order

Code

```
from collections import deque
def solution(nodes,edges):
    def topologicalSort(nodes,graph):
        # Calculate in-degree for each node in reversed graph
        inDegree={i:0 for i in nodes}
        for node,adj_nodes in graph.items():
            for i in adj_nodes:
                inDegree[i]+=1

        # Add all nodes with in-degree 0 to queue (terminal nodes in original graph)
        queue=deque()
        for node,degree in inDegree.items():
            if(degree==0):
                queue.append(node)

        ans=[]
        while queue:
            node=queue.popleft()  # Process safe node
            ans.append(node)

            # Reduce in-degree of adjacent nodes (nodes that lead to this safe node)
            for adj_node in graph[node]:
                inDegree[adj_node]-=1
                if(inDegree[adj_node]==0):  # If all paths lead to safe nodes
                    queue.append(adj_node)
        return ans

    # Create reversed graph
```

```
    graph={i:set() for i in nodes}
    for i,j in edges:
        graph[j].add(i)  # reverse edges: i->j becomes j->i

    topoSort=topologicalSort(nodes,graph)
    topoSort.sort()  # Return in ascending order
    return topoSort
```

Time and Space Complexity

- **Time Complexity:** O(V + E) for topological sort + O(V log V) for sorting = O(V + E + V log V)
- **Space Complexity:** O(V + E) for the reversed graph and auxiliary data structures

Solution 2: Using Depth First Search (DFS)

Intuition

Think of it like exploring a haunted house with multiple rooms. You want to find rooms that are "safe" - meaning no matter which door you take from that room, you'll eventually reach an exit (terminal node) and won't get trapped in a loop of scary rooms.

The key insight: A room is unsafe if you can get stuck in a loop of rooms (cycle) or if any path from that room leads to such a loop. We explore each room and mark it as safe only if ALL paths from it lead to exits, not to loops.

Approach Steps

1. **Track three states for each node**:
    - `visited` : Have we explored this node before?
    - `pathVisited` : Is this node part of our current exploration path?
    - `safe` : Is this node confirmed to be safe?
2. **For each unvisited node, perform DFS**:
    - Mark current node as visited and add to current path
    - Assume current node is unsafe initially
    - Explore all adjacent nodes
    - If any adjacent node leads to a cycle, current node is unsafe
    - If all paths are safe, mark current node as safe
    - Remove current node from path (backtrack)
3. **Return all safe nodes in sorted order**

Code

```
def solution(nodes, adj_list):
    graph = adj_list
    visited = {i: False for i in nodes}  # Track visited nodes
    pathVisited = {i: False for i in nodes}  # Track nodes in current path
    safe = {i: False for i in nodes}  # Track safe nodes

    def dfs(node):
        visited[node] = True  # Mark as visited
        pathVisited[node] = True  # Add to current path

        for adj_node in graph[node]:  # Explore all adjacent nodes
```

```
            if not visited[adj_node]:  # If not visited, explore recursively
                if not dfs(adj_node):  # If adjacent node is unsafe
                    pathVisited[node] = False  # Remove from path
                    return False  # Current node is also unsafe
            elif pathVisited[adj_node]:  # If in current path, cycle detected
                pathVisited[node] = False  # Remove from path
                return False  # Current node is unsafe
            elif not safe[adj_node]:  # If adjacent node is known to be unsafe
                pathVisited[node] = False  # Remove from path
                return False  # Current node is also unsafe

        pathVisited[node] = False  # Remove from current path (backtrack)
        safe[node] = True  # Mark as safe
        return True

    # Check each unvisited node
    for i in nodes:
        if not visited[i]:
            dfs(i)

    # Return all safe nodes in sorted order
    result = [node for node in nodes if safe[node]]
    return result
```

Time and Space Complexity

- **Time Complexity:** O(V + E) where V is the number of vertices and E is the number of edges
- **Space Complexity:** O(V) for the visited arrays and recursion stack

Course Schedule I

Problem Statement

There are a total of N tasks, labeled from 0 to N-1. Given an array arr where arr[i] = [a, b] indicates that you must take course b first if you want to take course a. Find if it is possible to finish all tasks.

Examples

Example 1:

Input: N = 4, arr = [[1,0],[2,1],[3,2]] **Output:** True

Explanation: It is possible to finish all the tasks in the order: 0 1 2 3. First, we will finish task 0. Then we will finish task 1, task 2, and task 3.

Example 2:

Input: N = 4, arr = [[0,1],[3,2],[1,3],[3,0]] **Output:** False

Explanation: It is impossible to finish all the tasks. Let's analyze the pairs:

- For pair {0, 1} -> we need to finish task 1 first and then task 0. (order: 1 0).

- For pair {3, 2} -> we need to finish task 2 first and then task 3. (order: 2 3).
- For pair {1, 3} -> we need to finish task 3 first and then task 1. (order: 3 1).
- But for pair {3, 0} -> we need to finish task 0 first and then task 3 but task 0 requires task 1 and task 1 requires task 3. So, it is not possible to finish all the tasks.

How this problem can be identified as a Graph problem?

The problem suggests that some courses must be completed before other courses. This is analogous to Topological Sort Algorithm in graph which helps to find a ordering where a node must come before other nodes in the ordering.

Hence, the courses can be represented as nodes of graphs and dependencies of courses can be shown as edges.

Now, For the graph formed, if the Topological sort can be found containing all the nodes (courses), all the courses can be completed in the order returned by topological sort. Else it is not possible to complete all the courses.

How to form the graph?

The pair [a,b] represents that the Course b must be completed before Course a.

Hence in the graph, two nodes representing Course a and b can be created with a directed edge from Node b to Node a. This way the topological sort will return Node b before Node a.

Solution: Using Topological Sort (Kahn's Algorithm)

Intuition

Think of it like planning your college courses where some courses have prerequisites. You can graduate (complete all courses) if and only if you can arrange all your courses in a valid order where you never take a course before completing its prerequisites.

The key insight: If there's a circular dependency (like "Course A needs Course B, Course B needs Course C, Course C needs Course A"), then it's impossible to complete all courses - you'll be stuck in an endless loop of prerequisites!

We use topological sort to try arranging all courses in a valid order. If we can arrange all N courses, then it's possible to complete them all. If some courses get stuck due to circular dependencies, it's impossible.

Approach Steps

1. **Build the graph**: Create a directed graph where each course is a node, and there's an edge from prerequisite to course (b → a for dependency [a,b])
2. **Apply topological sort using Kahn's algorithm**:
    - Calculate in-degree for each course (how many prerequisites it has)
    - Start with courses that have no prerequisites (in-degree = 0)
    - Process courses one by one, removing them and updating prerequisites count
    - Keep track of how many courses we successfully processed
3. **Check if all courses can be completed**:
    - If we processed all N courses, return True (possible to complete all)
    - If some courses remain unprocessed, return False (circular dependency exists)

Code

```
from collections import deque
def solution(nodes,edges):
    def topologicalSort(nodes,graph):
        # Calculate in-degree for each node (number of prerequisites)
        inDegree={i:0 for i in nodes}
        for node,adj_nodes in graph.items():
            for i in adj_nodes:
                inDegree[i]+=1

        # Add all nodes with in-degree 0 to queue (no prerequisites)
        queue=deque()
        for node,degree in inDegree.items():
            if(degree==0):
                queue.append(node)

        ans=[]
        while queue:
            node=queue.popleft()  # Take a course with no remaining prerequisites
            ans.append(node)

            # Remove this course as prerequisite for other courses
            for adj_node in graph[node]:
                inDegree[adj_node]-=1
                if(inDegree[adj_node]==0):  # If all prerequisites completed
                    queue.append(adj_node)
        return ans

    # Build the graph: for [a,b], add edge b -> a
    graph={i:set() for i in nodes}
    for i,j in edges:
        graph[j].add(i)  # j must come before i

    topoSort=topologicalSort(nodes,graph)
    if(len(topoSort)==len(nodes)):  # If all courses can be arranged
        return True
    return False  # If some courses stuck due to circular dependencies


n=4
edges= [[1,0],[2,1],[3,2]]
nodes=[i for i in range(n)]
print(solution(nodes,edges))
```

Time and Space Complexity

- **Time Complexity:** O(V + E) where V is the number of courses and E is the number of prerequisite relationships
- **Space Complexity:** O(V + E) for the graph representation and auxiliary data structures

Course Schedule II

Problem Statement

There are a total of N tasks, labeled from 0 to N-1. Given an array arr where arr[i] = [a, b] indicates that you must take course b first if you want to take course a. Find the order of tasks you should pick to finish all tasks. If no such ordering exists, return an empty array.

Examples

Example 1:

Input: N = 4, arr = [[1,0],[2,1],[3,2]] **Output:** [0, 1, 2, 3]

Explanation: First, finish task 0, as it has no prerequisites. Then, finish task 1, since it depends only on task 0. After that, finish task 2, since it depends only on task 1. Finally, finish task 3, since it depends only on task 2.

Example 2:

Input: N = 4, arr = [[0,1],[3,2],[1,3],[3,0]] **Output:** []

Explanation: It is impossible to finish all the tasks. Let's analyze the pairs:

- For pair {0, 1} → we need to finish task 1 first and then task 0 (order: 1 → 0).
- For pair {3, 2} → we need to finish task 2 first and then task 3 (order: 2 → 3).
- For pair {1, 3} → we need to finish task 3 first and then task 1 (order: 2 → 3 → 1 → 0).
- But for pair {3, 0} → we need to finish task 0 first and then task 3, which contradicts the previous order. So, it is not possible to finish all the tasks.

Solution: Using Topological Sort (Kahn's Algorithm)

Intuition

Think of it like creating a step-by-step study plan for your entire college curriculum. You need to figure out the exact order in which to take all your courses so that you never take a course before completing its prerequisites.

This is exactly like Course Schedule I, but now instead of just asking "Can I graduate?", we're asking "Give me the exact semester-by-semester plan to graduate!"

The approach is simple:

1. Start with courses that have no prerequisites (you can take them right away)
2. Take those courses and "unlock" the next set of courses
3. Keep going until you've planned all courses, or get stuck due to circular dependencies

If you can plan all N courses, return the plan. If you get stuck (due to circular dependencies), return an empty plan to indicate it's impossible.

Approach Steps

1. **Build the graph**: Create a directed graph where each course is a node, and there's an edge from prerequisite to course (b → a for dependency [a,b])
2. **Apply topological sort using Kahn's algorithm**:
    - Calculate in-degree for each course (how many prerequisites it has)

- Start with courses that have no prerequisites (in-degree = 0)
- Process courses one by one, adding them to our study plan
- For each course taken, reduce the prerequisite count of dependent courses
- Add newly available courses (those with no remaining prerequisites) to the queue

3. **Return the result**:
   - If we planned all N courses, return the complete study plan
   - If some courses couldn't be planned (due to circular dependencies), return empty array

Code

```
from collections import deque
def solution(nodes,edges):
    def topologicalSort(nodes,graph):
        # Calculate in-degree for each node (number of prerequisites)
        inDegree={i:0 for i in nodes}
        for node,adj_nodes in graph.items():
            for i in adj_nodes:
                inDegree[i]+=1

        # Add all nodes with in-degree 0 to queue (no prerequisites)
        queue=deque()
        for node,degree in inDegree.items():
            if(degree==0):
                queue.append(node)

        ans=[]
        while queue:
            node=queue.popleft()  # Take next course with no remaining prerequisites
            ans.append(node)  # Add to our study plan

            # Remove this course as prerequisite for other courses
            for adj_node in graph[node]:
                inDegree[adj_node]-=1
                if(inDegree[adj_node]==0):  # If all prerequisites completed
                    queue.append(adj_node)  # Course becomes available
        return ans

    # Build the graph: for [a,b], add edge b -> a
    graph={i:set() for i in nodes}
    for i,j in edges:
        graph[j].add(i)  # j must come before i

    topoSort=topologicalSort(nodes,graph)
    if(len(topoSort)==len(nodes)):  # If all courses can be planned
        return topoSort  # Return the complete study plan
    return []  # Return empty array if impossible due to circular dependencies
```

Test Case

```
n=4
edges= [[1,0],[2,1],[3,2]]
nodes=[i for i in range(n)]
print(solution(nodes,edges))
```

Time and Space Complexity

- **Time Complexity:** O(V + E) where V is the number of courses and E is the number of prerequisite relationships
- **Space Complexity:** O(V + E) for the graph representation and auxiliary data structures

Graph Algorithms - Complete Index

📚 Table of Contents

Core Shortest Path Algorithms

1. Shortest Path in DAG
   - Topological sorting approach
   - DFS-based topological sort
   - Kahn's algorithm implementation
   - Time: O(N + M), Space: O(N + M)
2. Shortest Path in Unweighted Graph
   - BFS approach for unit weights
   - Level-wise exploration
   - Time: O(N + M), Space: O(N + M)
3. Dijkstra's Algorithm
   - Single-source shortest path with non-negative weights
   - Priority queue implementation
   - Time: O((V + E) log V), Space: O(V + E)
4. Bellman-Ford Algorithm
   - Handles negative edge weights
   - Negative cycle detection
   - Time: O(V × E), Space: O(V)
5. Floyd-Warshall Algorithm
   - All-pairs shortest paths
   - Dynamic programming approach
   - Time: $O(V^3)$, Space: O(1)
6. Johnson's Algorithm
   - All-pairs with negative edges
   - Combines Bellman-Ford + Dijkstra
   - Time: $O(V^2 \log V + VE)$, Space: $O(V^2 + E)$

Advanced Shortest Path Problems

7. Print Shortest Path
   - Path reconstruction using parent tracking
   - Modified Dijkstra's with path storage
   - Time: O((V + E) log V), Space: O(V + E)

8. Cheapest Flights within K Stops
    - Constrained shortest path problem
    - Modified Dijkstra's with stop counting
    - Time: O(V + E × K), Space: O(V + E)
9. Number of Ways to Arrive at Destination
    - Shortest path counting problem
    - Modified Dijkstra's with path enumeration
    - Time: O((V + E) log V), Space: O(V + E)

Graph Traversal Applications

10. Word Ladder I
    - Shortest transformation sequence
    - BFS on word graph
    - Time: $O(M^2 \times N)$, Space: O(N × M)
11. Word Ladder II
    - All shortest transformation sequences
    - BFS + DFS with backtracking
    - Time: $O(N \times M^2 \times 26^L)$, Space: O(N × M + P × L)

Specialized Graph Problems

12. Minimum Multiplications to Reach End
    - Number transformation problem
    - BFS on state space (0-99999)
    - Time: O(100000 × M), Space: O(100000)
13. Shortest Distance in Binary Maze
    - Grid pathfinding problem
    - BFS on 2D grid
    - Time: O(n × m), Space: O(n × m)
14. Path with Minimum Effort
    - Minimize maximum edge weight
    - Modified Dijkstra's with max operation
    - Time: O(n × m × log(n × m)), Space: O(n × m)
15. Find City with Smallest Number of Neighbors
    - Distance threshold problem
    - Floyd-Warshall + counting
    - Time: $O(N^3)$, Space: $O(N^2)$

Advanced Search Algorithms

16. *A Search Algorithm**
    - Heuristic-guided pathfinding
    - Priority queue with f-cost = g-cost + h-cost
    - Time: O(b^d), Space: O(b^d)

🔍 Quick Reference Guide

Algorithm Selection Matrix

Problem Type	Recommended Algorithm	Key Characteristics
Single-source, non-negative weights	Dijkstra's	Fastest for this case
Single-source, with negative weights	Bellman-Ford	Detects negative cycles
Single-source, DAG	Topological Sort + DP	Optimal for DAGs
Single-source, unweighted	BFS	Simplest and fastest
All-pairs, small graphs	Floyd-Warshall	Easy to implement
All-pairs, large sparse graphs	Johnson's	More efficient than Floyd-Warshall
Path reconstruction needed	Modified Dijkstra's	Tracks parent pointers
Constrained shortest path	Modified BFS/Dijkstra's	Problem-specific modifications
Heuristic-guided search	A*	Best for pathfinding with heuristics

Complexity Quick Reference

Algorithm	Time Complexity	Space Complexity	Graph Type
BFS	O(V + E)	O(V)	Unweighted
Dijkstra's	O((V + E) log V)	O(V + E)	Non-negative weights
Bellman-Ford	O(V × E)	O(V)	Any weights
Floyd-Warshall	$O(V^3)$	$O(V^2)$	Any weights
Johnson's	$O(V^2 \log V + VE)$	$O(V^2 + E)$	Any weights
A*	O(b^d)	O(b^d)	Heuristic search

Common Problem Patterns

🎯 Pattern 1: Basic Shortest Path

- **Problems**: Dijkstra's, BFS on unweighted graphs
- **Key insight**: Direct application of standard algorithms
- **Implementation**: Priority queue or simple queue

🎯 Pattern 2: Path Reconstruction

- **Problems**: Print shortest path, Word Ladder II
- **Key insight**: Track parent pointers during search
- **Implementation**: Additional parent/predecessor tracking

🎯 Pattern 3: Constrained Shortest Path

- **Problems**: Cheapest flights with K stops, Path with minimum effort
- **Key insight**: Modify cost function or add constraints
- **Implementation**: Extended state space or modified relaxation

🎯 Pattern 4: State Space Search

- **Problems**: Word Ladder, Minimum multiplications
- **Key insight**: Transform problem into graph traversal
- **Implementation**: Generate neighbors dynamically

🎯 Pattern 5: Multiple Shortest Paths

- **Problems**: Number of ways to arrive, Word Ladder II
- **Key insight**: Count or enumerate instead of just finding one path
- **Implementation**: Modified algorithms with counting/path storage

📝 Implementation Tips

Common Data Structures

```
# Priority queue for Dijkstra's/A*
import heapq
pq = [(cost, node)]
heapq.heappush(pq, (new_cost, new_node))
cost, node = heapq.heappop(pq)

# Adjacency list representation
graph = defaultdict(list)
graph[u].append((v, weight))

# Distance tracking
dist = [float('inf')] * n
dist[source] = 0

# Parent tracking for path reconstruction
parent = [-1] * n
parent[v] = u  # u is parent of v
```

Edge Cases to Consider

- Source equals destination
- Unreachable destinations
- Negative cycles (for Bellman-Ford)
- Empty graphs
- Single node graphs
- Disconnected components

Optimization Techniques

- Early termination when destination reached
- Bidirectional search for long paths
- Visited set to avoid reprocessing
- Efficient heuristics for A*

🚀 Problem-Solving Strategy

Step 1: Problem Analysis

1. **Identify graph type**: Weighted/unweighted, directed/undirected
2. **Determine requirements**: Single-source vs all-pairs, path reconstruction needed?
3. **Check constraints**: Negative weights, cycles, special conditions

Step 2: Algorithm Selection

- Use the selection matrix above
- Consider time/space trade-offs
- Account for expected graph density

Step 3: Implementation Approach

- Start with standard algorithm template
- Add problem-specific modifications
- Handle edge cases and constraints

Step 4: Optimization

- Profile for bottlenecks
- Apply appropriate optimizations
- Validate correctness with test cases

This index provides a comprehensive overview of all graph algorithms covered in the document, with quick reference guides for algorithm selection, complexity analysis, and implementation strategies. Each algorithm includes links to detailed explanations, code implementations, and complexity analysis.

Shortest Path in DAG

Problem Description

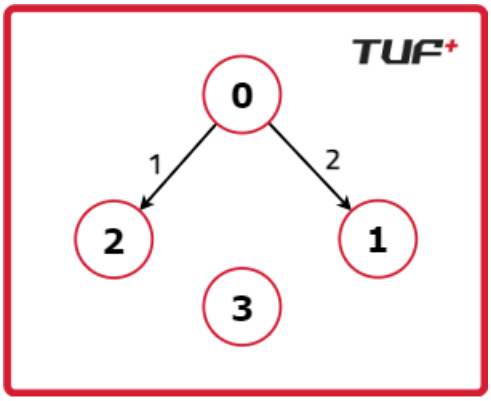
Given a Directed Acyclic Graph (DAG) with N vertices (numbered 0 to N-1) and M edges, find the shortest path from source vertex 0 to all other vertices. Each edge has a weight/distance. If any vertex cannot be reached from the source, return -1 for that vertex.

Think of it like finding the shortest route from your home (vertex 0) to all other places in a city where roads are one-way and there are no circular routes.

Examples

Input

```
N = 4, M = 2
edges = [[0,1,2], [0,2,1]]
```



Output

```
[0, 2, 1, -1]
```

Explanation

- Distance to vertex 0 (source): 0 (starting point)
- Distance to vertex 1: 2 (direct path 0→1 with weight 2)
- Distance to vertex 2: 1 (direct path 0→2 with weight 1)
- Distance to vertex 3: -1 (no path exists from 0 to 3)

Input

```
N = 6, M = 7
edges = [[0,1,2], [0,4,1], [4,5,4], [4,2,2], [1,2,3], [2,3,6], [5,3,1]]
```

Output

```
[0, 2, 3, 6, 1, 5]
```

Explanation

- Distance to vertex 0: 0 (source)
- Distance to vertex 1: 2 (path 0→1, weight 2)

- Distance to vertex 2: 3 (path 0→4→2, weight 1+2=3, shorter than 0→1→2 which is 2+3=5)
- Distance to vertex 3: 6 (path 0→4→5→3, weight 1+4+1=6)
- Distance to vertex 4: 1 (path 0→4, weight 1)
- Distance to vertex 5: 5 (path 0→4→5, weight 1+4=5)

Solution

Use topological sorting to process vertices in the correct order, then relax edges to find shortest distances. Since it's a DAG, we can process vertices in topological order to ensure we always have the shortest path to a vertex before processing its neighbors.

Intuition

The key insight is that in a DAG, if we process vertices in topological order, by the time we reach any vertex, we've already found the shortest paths to all vertices that can reach it. This is like solving a puzzle where you need to complete certain pieces before others - topological order tells us the correct sequence.

Since there are no cycles, once we find the shortest path to a vertex, it's guaranteed to be optimal. We don't need to worry about finding a better path later (unlike in graphs with cycles).

Approach Steps

1. **Build the graph**: Create an adjacency list representation from the edge list
2. **Topological sorting**: Use DFS to get the topological order of vertices
3. **Initialize distances**: Set distance to source as 0, all others as infinity
4. **Process vertices**: For each vertex in topological order:
    - If the vertex is reachable (distance < infinity)
    - Update distances to all its neighbors using edge relaxation
5. **Handle unreachable vertices**: Set distance of unreachable vertices to -1
6. **Return result**: Return the distance array

Code

Method 1: Topological Sort with DFS

```
from collections import defaultdict

def shortestPath(N, M, edges):
    # Build adjacency list
    graph = defaultdict(list)
    for u, v, weight in edges:
        graph[u].append((v, weight))

    # Topological sort using DFS
    def topological_sort():
        visited = [False] * N
        stack = []

        def dfs(node):
            visited[node] = True
```

```
            for neighbor, _ in graph[node]:
                if not visited[neighbor]:
                    dfs(neighbor)
            stack.append(node)

        for i in range(N):
            if not visited[i]:
                dfs(i)

        return stack[::-1]  # Reverse to get correct topological order

    # Get topological order
    topo_order = topological_sort()

    # Initialize distances
    dist = [float('inf')] * N
    dist[0] = 0  # Source vertex distance is 0

    # Process vertices in topological order
    for node in topo_order:
        if dist[node] != float('inf'):  # If node is reachable
            for neighbor, weight in graph[node]:
                # Relax the edge
                if dist[node] + weight < dist[neighbor]:
                    dist[neighbor] = dist[node] + weight

    # Convert unreachable vertices distance to -1
    for i in range(N):
        if dist[i] == float('inf'):
            dist[i] = -1

    return dist
```

Method 2: Kahn's Algorithm for Topological Sort

```
from collections import defaultdict, deque

def shortestPathKahn(N, M, edges):
    # Build adjacency list and calculate in-degrees
    graph = defaultdict(list)
    in_degree = [0] * N

    for u, v, weight in edges:
        graph[u].append((v, weight))
        in_degree[v] += 1

    # Topological sort using Kahn's algorithm
    queue = deque()
    for i in range(N):
        if in_degree[i] == 0:
            queue.append(i)

    topo_order = []
    while queue:
        node = queue.popleft()
        topo_order.append(node)

        for neighbor, _ in graph[node]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
```

```
                queue.append(neighbor)

    # Initialize distances and process
    dist = [float('inf')] * N
    dist[0] = 0

    for node in topo_order:
        if dist[node] != float('inf'):
            for neighbor, weight in graph[node]:
                if dist[node] + weight < dist[neighbor]:
                    dist[neighbor] = dist[node] + weight

    # Convert unreachable to -1
    for i in range(N):
        if dist[i] == float('inf'):
            dist[i] = -1

    return dist
```

Method 3: BFS-like Approach (Alternative Implementation)

```
from collections import deque

def solution(nodes, graph, source):
    # Initialize distance array
    distance = {i: float('inf') for i in nodes}
    distance[source] = 0

    # Use queue for BFS-like traversal
    queue = deque()
    queue.append(source)

    while queue:
        node = queue.popleft()
        # Process all adjacent nodes
        for adj_node in graph[node].keys():
            # Relax the edge if shorter path found
            if distance[node] + graph[node][adj_node] < distance[adj_node]:
                distance[adj_node] = distance[node] + graph[node][adj_node]
                queue.append(adj_node)

    # Convert unreachable nodes to -1
    for node, dist in distance.items():
        if dist == float('inf'):
            distance[node] = -1

    return list(distance.values())

class Solution:
    def shortestPath(self, N, M, edges):
        # Create nodes list and adjacency list representation
        nodes = [i for i in range(N)]
        graph = {i: {} for i in nodes}

        # Build the graph
        for i, j, v in edges:
            graph[i][j] = v

        return solution(nodes, graph, 0)
```

Note: Method 3 uses a BFS-like approach but may not work correctly for all DAG cases as it doesn't guarantee topological processing order. Methods 1 and 2 are recommended for reliable results.

Time and Space Complexity

Time Complexity: O(N + M)

- Topological sorting takes O(N + M) time where we visit each vertex once and traverse each edge once
- Processing vertices in topological order and relaxing edges takes O(M) time
- Overall: O(N + M)

Space Complexity: O(N + M)

- Adjacency list representation takes O(M) space to store all edges
- Topological sorting requires O(N) space for visited array and recursion stack (DFS) or queue (Kahn's)
- Distance array takes O(N) space
- Overall: O(N + M)

The algorithm is very efficient because we only visit each vertex and edge once, making it much faster than general shortest path algorithms like Dijkstra's for DAGs.

Shortest Path in Undirected Graph with Unit Weights

Problem Description

Given an undirected graph with N vertices (numbered 0 to N-1) and M edges where all edges have unit weight (weight = 1), find the shortest path from source vertex 0 to all other vertices. If any vertex is unreachable from the source, return -1 for that vertex.

Think of it like finding the minimum number of steps needed to reach every location from your starting point, where you can move in any direction along connected paths.

Examples

Input

```
n = 9, m = 10
edges = [[0,1],[0,3],[3,4],[4,5],[5,6],[1,2],[2,6],[6,7],[7,8],[6,8]]
```

Output

```
[0, 1, 2, 1, 2, 3, 3, 4, 4]
```

Explanation

Starting from vertex 0:

- Distance to vertex 0: 0 (source)
- Distance to vertex 1: 1 (direct connection: 0→1)
- Distance to vertex 2: 2 (path: 0→1→2)
- Distance to vertex 3: 1 (direct connection: 0→3)
- Distance to vertex 4: 2 (path: 0→3→4)
- Distance to vertex 5: 3 (path: 0→3→4→5)
- Distance to vertex 6: 3 (shortest path: 0→1→2→6 or 0→3→4→5→6)
- Distance to vertex 7: 4 (path: 0→1→2→6→7)
- Distance to vertex 8: 4 (path: 0→1→2→6→8)

Input

```
n = 8, m = 10
edges = [[1,0],[2,1],[0,3],[3,7],[3,4],[7,4],[7,6],[4,5],[4,6],[6,5]]
```

Output

```
[0, 1, 2, 1, 2, 3, 3, 2]
```

Explanation

- Distance to vertex 7: 2 (path: 0→3→7, shorter than going through other nodes)
- All distances represent the minimum number of edges needed to reach each vertex from source 0

Solution

Use BFS (Breadth-First Search) traversal starting from the source vertex. Since all edges have unit weight, BFS naturally finds the shortest path because it explores nodes level by level, ensuring the first time a node is reached, it's via the shortest path.

Intuition

The key insight is that BFS explores nodes level by level. In an unweighted graph (or unit weight graph), this means:

- Level 0: Source node (distance 0)
- Level 1: All nodes directly connected to source (distance 1)
- Level 2: All nodes reachable in 2 steps (distance 2)
- And so on...

Since BFS visits nodes in increasing order of their distance from source, the first time we reach any node is guaranteed to be via the shortest path. This is much simpler than using Dijkstra's algorithm for weighted graphs.

Approach Steps

1. **Build the graph**: Create an adjacency list for the undirected graph (add edges in both directions)
2. **Initialize data structures**:
   - Distance array with infinity for all nodes except source (set to 0)
   - Queue for BFS traversal, starting with source node
3. **BFS traversal**:
   - Process each node from queue
   - For each unvisited neighbor, update its distance and add to queue
4. **Handle unreachable nodes**: Convert any remaining infinity distances to -1
5. **Return result**: Return the distance array

Code

Method 1: Standard BFS with Visited Array

```
from collections import deque

def solution(nodes, graph, source):
    # Initialize visited and distance arrays
    visited = {i: False for i in nodes}
    distance = {i: float('inf') for i in nodes}

    # Set source distance and mark as visited
    distance[source] = 0
    visited[source] = True

    # BFS using queue
    queue = deque()
    queue.append(source)

    while queue:
        node = queue.popleft()

        # Process all adjacent nodes
        for adj_node in graph[node]:
            if not visited[adj_node]:
                # Update distance (always current distance + 1 for unit weights)
                distance[adj_node] = distance[node] + 1
                visited[adj_node] = True
                queue.append(adj_node)

    # Convert unreachable nodes to -1
    for node, dist in distance.items():
        if dist == float('inf'):
            distance[node] = -1

    return list(distance.values())

class Solution:
    def shortestPath(self, edges, N, M):
        # Create nodes and adjacency list
        nodes = [i for i in range(N)]
        graph = {i: set() for i in nodes}
```

```
        # Build undirected graph (add edge in both directions)
        for i, j in edges:
            graph[i].add(j)
            graph[j].add(i)

        return solution(nodes, graph, 0)
```

Method 2: Optimized BFS (Distance Array as Visited Array)

```
from collections import deque

def shortestPathOptimized(N, M, edges):
    # Build adjacency list
    graph = [[] for _ in range(N)]
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    # Initialize distance array
    distance = [float('inf')] * N
    distance[0] = 0  # Source distance is 0

    # BFS traversal
    queue = deque([0])

    while queue:
        node = queue.popleft()

        for neighbor in graph[node]:
            # If not visited (distance is still infinity)
            if distance[neighbor] == float('inf'):
                distance[neighbor] = distance[node] + 1
                queue.append(neighbor)

    # Convert unreachable nodes to -1
    for i in range(N):
        if distance[i] == float('inf'):
            distance[i] = -1

    return distance
```

Method 3: Clean Implementation

```
from collections import deque

def shortestPath(N, M, edges):
    # Create adjacency list
    adj = [[] for _ in range(N)]
    for u, v in edges:
        adj[u].append(v)
        adj[v].append(v)

    # BFS from source (node 0)
    dist = [-1] * N  # Initialize with -1 (unreachable)
    dist[0] = 0      # Source distance is 0
    queue = deque([0])

    while queue:
```

```
        current = queue.popleft()

        for neighbor in adj[current]:
            if dist[neighbor] == -1:  # Not visited
                dist[neighbor] = dist[current] + 1
                queue.append(neighbor)

    return dist
```

Time and Space Complexity

Time Complexity: O(N + M)

- We visit each vertex exactly once: O(N)
- We examine each edge exactly twice (once from each endpoint): O(M)
- Overall: O(N + M) where N is vertices and M is edges

Space Complexity: O(N + M)

- Adjacency list representation takes O(M) space to store all edges
- Distance array takes O(N) space
- BFS queue can contain at most O(N) vertices in worst case
- Overall: O(N + M)

Key Advantages:

- Much simpler than Dijkstra's algorithm for unweighted graphs
- Optimal time complexity for this problem
- BFS guarantees shortest path for unit weight graphs
- No need for priority queue unlike Dijkstra's algorithm

Dijkstra's Algorithm

Problem Description

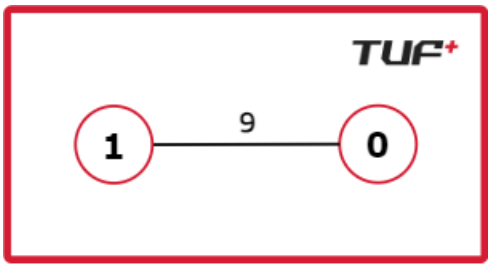
Given a weighted, undirected graph with V vertices (numbered 0 to V-1) and a source node S, find the shortest distance from the source vertex to all other vertices. If a vertex is not reachable from the source, return 10^9 for that vertex.

Think of it like finding the shortest route (by total distance/cost) from your home to all other locations in a city, where roads have different lengths/costs.

Examples

Input

```
V = 2, adj = [[[1, 9]], [[0, 9]]], S = 0
```



Output

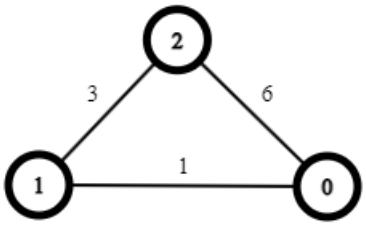
```
[0, 9]
```

Explanation

- Distance from source 0 to itself: 0
- Distance from source 0 to vertex 1: 9 (direct path with weight 9)

Input

```
V = 3, adj = [[[1, 1], [2, 6]], [[2, 3], [0, 1]], [[1, 3], [0, 6]]], S = 2
```



Output

```
[4, 3, 0]
```

Explanation

Starting from source vertex 2:

- Distance to vertex 0: 4 (path: 2→1→0 with weights 3+1=4, shorter than direct path 2→0 with weight 6)
- Distance to vertex 1: 3 (direct path: 2→1 with weight 3)
- Distance to vertex 2: 0 (source itself)

Solution

Use Dijkstra's algorithm with a priority queue (min-heap) to always process the node with the smallest known distance first. This greedy approach ensures we find the optimal shortest path to all vertices.

Intuition

Dijkstra's algorithm works on the principle that once we've found the shortest path to a vertex, we can use that vertex to potentially find shorter paths to other vertices. It's like solving a puzzle piece by piece - once we're certain about the shortest distance to a vertex, we can use it as a stepping stone to reach other vertices.

The key insight is to always process the vertex with the smallest tentative distance first. This ensures that when we process a vertex, we've already found its shortest path from the source.

Why does this work?

- When we extract a vertex with minimum distance from the priority queue, that distance is guaranteed to be the shortest possible
- Any alternative path to this vertex would have to go through other vertices with larger distances, making the total path longer

Approach Steps

1. **Initialize data structures**:
   - Distance array with infinity for all nodes except source (set to 0)
   - Priority queue (min-heap) to store {distance, node} pairs
2. **Add source to queue**: Push {0, source} to priority queue
3. **Main algorithm loop**:
   - Extract node with minimum distance from queue
   - For each neighbor, perform edge relaxation:
     - If current_distance + edge_weight < neighbor_distance, update it
     - Push the updated {new_distance, neighbor} to queue
4. **Continue until queue is empty**: All reachable nodes will have optimal distances
5. **Return result**: Distance array contains shortest paths from source

Code

Method 1: Using Min-Heap (Provided Implementation)

```
import heapq

def solution(nodes, graph, source):
    # Initialize priority queue and distance array
    queue = []
    heapq.heappush(queue, (0, source))
    distance = {i: 10**9 for i in nodes}
    distance[source] = 0

    while queue:
        dist, node = heapq.heappop(queue)

        # Skip if we've already found a better path
        if dist > distance[node]:
            continue

        # Process all neighbors
        for adj_node, adj_weight in graph[node].items():
```

```
            if distance[node] + adj_weight < distance[adj_node]:
                distance[adj_node] = distance[node] + adj_weight
                heapq.heappush(queue, (distance[adj_node], adj_node))

    return list(distance.values())

class Solution:
    def dijkstra(self, V, adj, S):
        # Convert adjacency list to dictionary format
        graph = {}
        for i, v in enumerate(adj):
            graph[i] = dict(v)

        nodes = [i for i in range(V)]
        ans = solution(nodes, graph, S)
        return ans
```

Method 2: Clean Implementation with Standard Format

```
import heapq

def dijkstra(V, adj, source):
    # Initialize distance array
    dist = [10**9] * V
    dist[source] = 0

    # Priority queue: (distance, node)
    pq = [(0, source)]

    while pq:
        current_dist, u = heapq.heappop(pq)

        # Skip if we've already processed this node with better distance
        if current_dist > dist[u]:
            continue

        # Process all neighbors
        for neighbor, weight in adj[u]:
            new_dist = dist[u] + weight

            # If we found a shorter path
            if new_dist < dist[neighbor]:
                dist[neighbor] = new_dist
                heapq.heappush(pq, (new_dist, neighbor))

    return dist

class Solution:
    def dijkstra(self, V, adj, S):
        return dijkstra(V, adj, S)
```

Method 3: With Visited Set Optimization

```
import heapq

def dijkstraOptimized(V, adj, source):
    # Initialize distance and visited arrays
    dist = [10**9] * V
```

```
    dist[source] = 0
    visited = [False] * V

    # Priority queue
    pq = [(0, source)]

    while pq:
        current_dist, u = heapq.heappop(pq)

        # Skip if already processed
        if visited[u]:
            continue

        visited[u] = True

        # Process neighbors
        for neighbor, weight in adj[u]:
            if not visited[neighbor]:
                new_dist = dist[u] + weight
                if new_dist < dist[neighbor]:
                    dist[neighbor] = new_dist
                    heapq.heappush(pq, (new_dist, neighbor))

    return dist
```

Time and Space Complexity

Time Complexity: O((V + E) log V)

- Each vertex is extracted from the priority queue at most once: O(V log V)
- Each edge is relaxed at most once, and each relaxation involves a heap operation: O(E log V)
- Overall: O((V + E) log V) where V is vertices and E is edges

Space Complexity: O(V + E)

- Adjacency list representation takes O(E) space
- Distance array takes O(V) space
- Priority queue can contain at most O(E) elements in worst case
- Overall: O(V + E)

Important Notes:

- **Works only with non-negative edge weights** - negative weights can break the greedy property
- **Handles both directed and undirected graphs** - just ensure adjacency list is built correctly
- **More efficient than Bellman-Ford** for graphs without negative edges
- **Priority queue ensures optimal substructure** - always processes minimum distance node first

When to use Dijkstra's:

- Single-source shortest path with non-negative weights
- Need shortest path to all vertices from one source
- Graph is dense (many edges) - better than multiple BFS calls
- Real-world applications: GPS navigation, network routing, social networks

Print Shortest Path

Problem Description

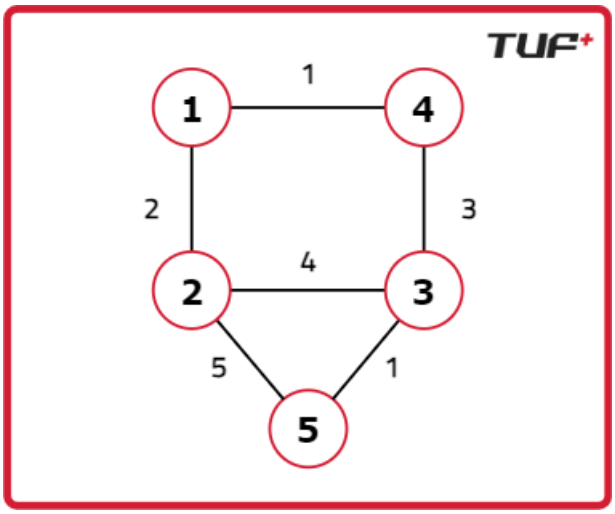
Given a weighted undirected graph with n vertices (numbered 1 to n) and m edges, find the shortest path between vertex 1 and vertex n. If no path exists, return [-1]. If a path exists, return a list where the first element is the total weight of the shortest path, followed by the vertices in the shortest path from 1 to n.

Think of it like finding the cheapest route from city 1 to city n, and not just knowing the cost, but also remembering which cities you passed through.

Examples

Input

```
n = 5, m = 6
edges = [[1,2,2], [2,5,5], [2,3,4], [1,4,1], [4,3,3], [3,5,1]]
```



Output

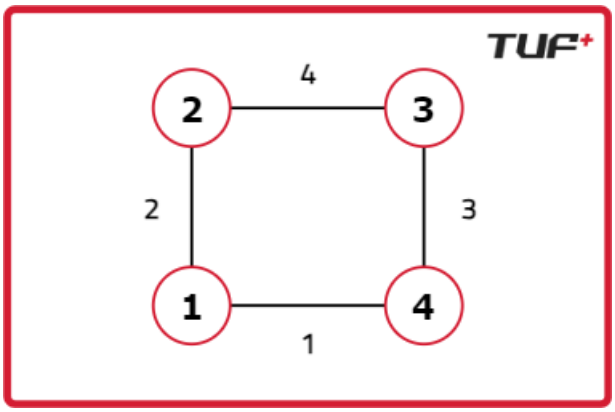
```
[5, 1, 4, 3, 5]
```

Explanation

- Shortest path from 1 to 5 has total weight 5
- Path: 1→4 (weight 1) → 3 (weight 3) → 5 (weight 1)
- Total: 1 + 3 + 1 = 5
- Alternative path 1→2→5 would cost 2 + 5 = 7 (more expensive)

Input

```
n = 4, m = 4
edges = [[1,2,2], [2,3,4], [1,4,1], [4,3,3]]
```



Output ``` [1, 1, 4] ```

Explanation

- Shortest path from 1 to 4 has total weight 1
- Path: 1→4 (direct connection with weight 1)
- Alternative path 1→2→3→4 would cost 2 + 4 + 3 = 9 (much more expensive)

Solution

Use a modified Dijkstra's algorithm that not only finds shortest distances but also tracks the parent of each vertex to reconstruct the shortest path. After finding shortest distances, trace back from destination to source using parent information.

Intuition

Regular Dijkstra's algorithm finds the shortest distance to all vertices, but doesn't remember the actual path. To print the shortest path, we need to track how we reached each vertex optimally.

The key modification is maintaining a parent array alongside the distance array. When we relax an edge and find a shorter path to a vertex, we not only update its distance but also record which vertex we came from (its parent in the shortest path tree).

Path Reconstruction Logic:

- Start from the destination vertex
- Keep following parent pointers until we reach the source
- Reverse this path to get source→destination order

Approach Steps

1. **Initialize data structures**:
    - Distance array with infinity for all nodes except source (set to 0)
    - Parent array to track the shortest path tree
    - Priority queue for Dijkstra's algorithm

2. **Run modified Dijkstra's**:
   - Process vertices in order of shortest distance
   - When relaxing edges, update both distance and parent information
3. **Check if destination is reachable**: If distance to destination is still infinity, return [-1]
4. **Reconstruct path**: Trace back from destination to source using parent array
5. **Format result**: Return [total_weight, vertex1, vertex2, ..., destination]

Code

Method 1: Modified Implementation (Based on Provided Code)

```
import heapq

def solution(nodes, graph, source, destination):
    # Initialize data structures
    queue = []
    distance = {i: float('inf') for i in nodes}
    parent = {i: None for i in nodes}

    # Set source distance and add to queue
    distance[source] = 0
    heapq.heappush(queue, (0, source))

    while queue:
        dist, node = heapq.heappop(queue)

        # Skip if we've found better path already
        if dist > distance[node]:
            continue

        # Process all neighbors
        for adj_node, adj_weight in graph[node].items():
            new_dist = dist + adj_weight
            if new_dist < distance[adj_node]:
                distance[adj_node] = new_dist
                parent[adj_node] = node
                heapq.heappush(queue, (new_dist, adj_node))

    # Check if destination is reachable
    if distance[destination] == float('inf'):
        return [-1]

    # Reconstruct path from destination to source
    path = []
    current = destination
    while current is not None:
        path.append(current)
        current = parent[current]

    # Reverse to get source to destination path
    path.reverse()

    # Return [total_weight, path...]
    return [distance[destination]] + path

class Solution:
    def shortestPath(self, n, m, edges):
```

```
        # Create nodes and graph
        nodes = [i for i in range(1, n + 1)]
        graph = {i: {} for i in nodes}

        # Build undirected graph
        for u, v, weight in edges:
            graph[u][v] = weight
            graph[v][u] = weight

        return solution(nodes, graph, 1, n)
```

Method 2: Clean Standard Implementation

```
import heapq

def printShortestPath(n, m, edges):
    # Build adjacency list
    graph = [[] for _ in range(n + 1)]
    for u, v, weight in edges:
        graph[u].append((v, weight))
        graph[v].append((u, weight))

    # Dijkstra's with path tracking
    dist = [float('inf')] * (n + 1)
    parent = [-1] * (n + 1)

    dist[1] = 0
    pq = [(0, 1)]  # (distance, node)

    while pq:
        d, u = heapq.heappop(pq)

        if d > dist[u]:
            continue

        for v, weight in graph[u]:
            new_dist = dist[u] + weight
            if new_dist < dist[v]:
                dist[v] = new_dist
                parent[v] = u
                heapq.heappush(pq, (new_dist, v))

    # Check if destination is reachable
    if dist[n] == float('inf'):
        return [-1]

    # Reconstruct path
    path = []
    current = n
    while current != -1:
        path.append(current)
        current = parent[current]

    path.reverse()
    return [dist[n]] + path

class Solution:
    def shortestPath(self, n, m, edges):
        return printShortestPath(n, m, edges)
```

Method 3: With Early Termination Optimization

```
import heapq

def shortestPathOptimized(n, m, edges):
    # Build graph
    graph = [[] for _ in range(n + 1)]
    for u, v, weight in edges:
        graph[u].append((v, weight))
        graph[v].append((u, weight))

    # Dijkstra's algorithm
    dist = [float('inf')] * (n + 1)
    parent = [-1] * (n + 1)
    visited = [False] * (n + 1)

    dist[1] = 0
    pq = [(0, 1)]

    while pq:
        d, u = heapq.heappop(pq)

        if visited[u]:
            continue

        visited[u] = True

        # Early termination if we reached destination
        if u == n:
            break

        for v, weight in graph[u]:
            if not visited[v]:
                new_dist = dist[u] + weight
                if new_dist < dist[v]:
                    dist[v] = new_dist
                    parent[v] = u
                    heapq.heappush(pq, (new_dist, v))

    # Check reachability and reconstruct path
    if dist[n] == float('inf'):
        return [-1]

    path = []
    current = n
    while current != -1:
        path.append(current)
        current = parent[current]

    path.reverse()
    return [dist[n]] + path
```

Time and Space Complexity

Time Complexity: O((V + E) log V)

- Same as standard Dijkstra's algorithm
- Path reconstruction takes O(V) additional time

- Overall: O((V + E) log V) where V is vertices and E is edges

Space Complexity: O(V + E)

- Adjacency list: O(E) space
- Distance and parent arrays: O(V) space each
- Priority queue: O(E) space in worst case
- Path reconstruction: O(V) space for storing path
- Overall: O(V + E)

Key Differences from Standard Dijkstra's:

- **Parent tracking**: Additional parent array to remember shortest path tree
- **Path reconstruction**: Trace back from destination to source using parent pointers
- **Output format**: Return both distance and actual path vertices
- **Early termination**: Can stop once destination is reached (optimization)

Important Notes:

- **Works only with non-negative weights** like standard Dijkstra's
- **Path uniqueness**: Multiple shortest paths may exist, algorithm returns one of them
- **1-indexed vertices**: Problem uses 1-based indexing instead of 0-based
- **Undirected graph**: Each edge is added in both directions

Cheapest Flights within K Stops

Problem Description

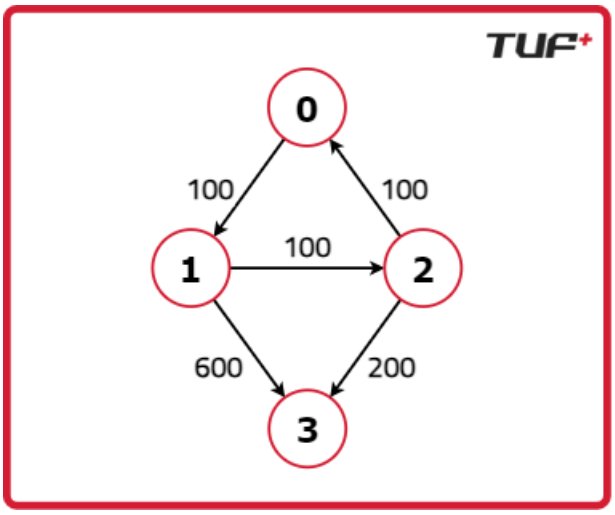
Given n cities connected by flights, find the cheapest price to travel from source city to destination city with at most k stops. Each flight has a cost, and you need to minimize the total cost while respecting the maximum number of stops allowed. If no valid route exists, return -1.

Think of it like booking the cheapest flight ticket with a layover limit - you want the lowest price but can't have too many connecting flights.

Examples

Input

```
n = 4, flights = [[0,1,100],[1,2,100],[2,0,100],[1,3,600],[2,3,200]], src = 0, dst = 3, k = 1
```



Output

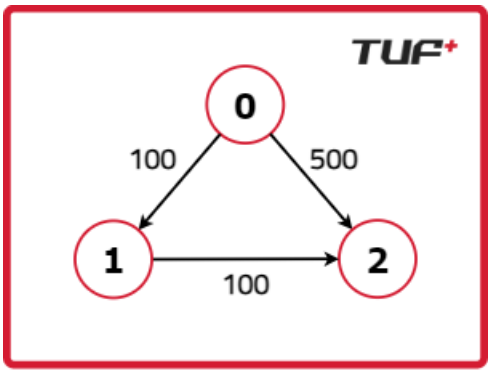
```
700
```

Explanation

- Path 0→1→3: Cost = 100 + 600 = 700 (1 stop, valid)
- Path 0→1→2→3: Cost = 100 + 100 + 200 = 400 (2 stops, invalid as k=1)
- The cheaper path is invalid due to stop limit, so answer is 700

Input

```
n = 3, flights = [[0,1,100],[1,2,100],[0,2,500]], src = 0, dst = 2, k = 1
```



Output

```
200
```

Explanation

- Direct path 0→2: Cost = 500 (0 stops)
- Path with stop 0→1→2: Cost = 100 + 100 = 200 (1 stop)

- The path with 1 stop is cheaper: 200

Solution

Use a modified Dijkstra's algorithm that prioritizes by number of stops instead of distance. Since stops increase monotonically, we can use a simple queue instead of a priority queue for better time complexity.

Intuition

This problem combines two constraints: minimizing cost and limiting stops. Regular Dijkstra's won't work directly because:

1. **Standard Dijkstra's prioritizes minimum distance** - but a longer distance path might have fewer stops
2. **We need to track both cost and stops** - not just the minimum cost

Key insight: Since stops increase by exactly 1 at each level, we can use BFS-like traversal where we process paths level by level based on number of stops. This ensures we don't exceed the stop limit.

Why prioritize stops over cost?

- If we prioritize cost first, we might get stuck with expensive paths that have fewer stops
- By processing paths with fewer stops first, we ensure we don't miss valid solutions due to exceeding the stop limit

Approach Steps

1. **Build the graph**: Create adjacency list representation from flights
2. **Initialize data structures**:
   - Distance array to track minimum cost to reach each city
   - Queue to store (stops, city, cost) tuples
3. **BFS-like traversal**:
   - Start from source with 0 stops and 0 cost
   - For each city, explore all neighboring cities
   - Only proceed if stops ≤ k and we found a cheaper path
4. **Update distances**: Keep track of minimum cost to reach each city
5. **Return result**: Return minimum cost to destination, or -1 if unreachable

Code

Method 1: Modified Dijkstra's (Based on Provided Code)

```python
import heapq

def solution(nodes, graph, source, destination, k):
    # Use priority queue: (stops, node, cost)
    queue = []
    distance = {i: float('inf') for i in nodes}

    # Start from source with 0 stops and 0 cost
    heapq.heappush(queue, (0, source, 0))
```

```
    distance[source] = 0

    while queue:
        steps, node, cost = heapq.heappop(queue)

        # Skip if exceeded stop limit
        if steps > k:
            continue

        # Process all neighbors
        for adj_node, flight_cost in graph[node].items():
            new_cost = cost + flight_cost

            # If we found a cheaper path and within stop limit
            if new_cost < distance[adj_node]:
                distance[adj_node] = new_cost
                # Add to queue with incremented stops
                heapq.heappush(queue, (steps + 1, adj_node, new_cost))

    return distance[destination] if distance[destination] != float('inf') else -1

class Solution:
    def findCheapestPrice(self, n: int, flights: List[List[int]], src: int, dst: int, k: int) -> int:
        nodes = [i for i in range(n)]
        graph = {i: {} for i in nodes}

        # Build directed graph
        for u, v, cost in flights:
            graph[u][v] = cost

        return solution(nodes, graph, src, dst, k)
```

Method 2: BFS with Queue (More Efficient)

```
from collections import deque

def findCheapestPriceBFS(n, flights, src, dst, k):
    # Build adjacency list
    graph = [[] for _ in range(n)]
    for u, v, cost in flights:
        graph[u].append((v, cost))

    # Use simple queue since stops increase monotonically
    queue = deque([(src, 0, 0)])  # (city, cost, stops)
    min_cost = [float('inf')] * n
    min_cost[src] = 0

    while queue:
        city, cost, stops = queue.popleft()

        # Skip if exceeded stop limit
        if stops > k:
            continue

        # Process all neighbors
        for next_city, flight_cost in graph[city]:
            new_cost = cost + flight_cost

            # Only proceed if we found a cheaper path
            if new_cost < min_cost[next_city]:
```

```
                min_cost[next_city] = new_cost
                queue.append((next_city, new_cost, stops + 1))

    return min_cost[dst] if min_cost[dst] != float('inf') else -1
```

Method 3: Bellman-Ford Variation (Alternative Approach)

```
def findCheapestPriceBellmanFord(n, flights, src, dst, k):
    # Initialize distance array
    dist = [float('inf')] * n
    dist[src] = 0

    # Relax edges at most k+1 times (k stops = k+1 edges)
    for i in range(k + 1):
        temp = dist.copy()  # Use previous iteration's distances

        for u, v, cost in flights:
            if dist[u] != float('inf'):
                temp[v] = min(temp[v], dist[u] + cost)

        dist = temp

    return dist[dst] if dist[dst] != float('inf') else -1
```

Method 4: DFS with Memoization (For Learning)

```
def findCheapestPriceDFS(n, flights, src, dst, k):
    # Build adjacency list
    graph = [[] for _ in range(n)]
    for u, v, cost in flights:
        graph[u].append((v, cost))

    # Memoization: (city, stops_left) -> min_cost
    memo = {}

    def dfs(city, stops_left):
        if city == dst:
            return 0
        if stops_left < 0:
            return float('inf')
        if (city, stops_left) in memo:
            return memo[(city, stops_left)]

        min_cost = float('inf')
        for next_city, flight_cost in graph[city]:
            cost = flight_cost + dfs(next_city, stops_left - 1)
            min_cost = min(min_cost, cost)

        memo[(city, stops_left)] = min_cost
        return min_cost

    result = dfs(src, k + 1)  # k stops = k+1 edges
    return result if result != float('inf') else -1
```

Time and Space Complexity

Method 1 & 2 (BFS/Modified Dijkstra's):

Time Complexity: O(V + E × K)

- Each city can be visited multiple times (up to k+1 times)
- Each edge is processed at most k+1 times
- V is number of cities, E is number of flights

Space Complexity: O(V + E)

- Adjacency list: O(E) space
- Queue and distance array: O(V) space

Method 3 (Bellman-Ford):

Time Complexity: O(K × E)

- K+1 iterations of edge relaxation
- Each iteration processes all E flights

Space Complexity: O(V)

- Only distance arrays needed

Key Points to Remember

Why not standard Dijkstra's?

- Standard Dijkstra's prioritizes minimum distance
- Here we need to balance cost vs. number of stops
- A cheaper path might exceed the stop limit

Why BFS works better here?

- Stops increase monotonically (always +1)
- No need for priority queue's logarithmic operations
- Simpler and more efficient for this specific problem

Edge Cases:

- Direct flight exists (0 stops)
- No path exists within k stops
- k = 0 (only direct flights allowed)
- Source equals destination

Real-world Applications:

- Flight booking systems with layover limits
- Network routing with hop count restrictions
- Supply chain optimization with intermediate stops

Number of Ways to Arrive at Destination

Problem Description

Given a city with n intersections (numbered 0 to n-1) connected by bi-directional roads, find the number of ways to travel from intersection 0 to intersection n-1 in the shortest possible time. Each road has a travel time, and you need to count all different shortest paths. Return the answer modulo 10^9 + 7.

Think of it like finding all the fastest routes on GPS - there might be multiple different paths that all take the same minimum time.

Examples

Input

```
n = 7, m = 10
roads = [[0,6,7],[0,1,2],[1,2,3],[1,3,3],[6,3,3],[3,5,1],[6,5,1],[2,5,1],[0,4,5],[4,6,2]]
```

Output

```
4
```

Explanation

Shortest time from 0 to 6 is 7 minutes. The four ways to achieve this are:

- Path 1: 0→6 (time: 7)
- Path 2: 0→4→6 (time: 5+2 = 7)
- Path 3: 0→1→2→5→6 (time: 2+3+1+1 = 7)
- Path 4: 0→1→3→5→6 (time: 2+3+1+1 = 7)

Input

```
n = 6, m = 8
roads = [[0,5,8],[0,2,2],[0,1,1],[1,3,3],[1,2,3],[2,5,6],[3,4,2],[4,5,2]]
```

Output

```
3
```

Explanation

Shortest time from 0 to 5 is 8 minutes. The three ways are:

- Path 1: 0→5 (time: 8)
- Path 2: 0→2→5 (time: 2+6 = 8)
- Path 3: 0→1→3→4→5 (time: 1+3+2+2 = 8)

Solution

Use a modified Dijkstra's algorithm that tracks both the shortest distance and the number of ways to achieve that shortest distance to each node. When we find a path of equal length, we add the ways; when we find a shorter path, we replace the ways.

Intuition

This problem combines shortest path finding with path counting. The key insight is:

Number of ways to reach a node in shortest time = Sum of ways to reach all parent nodes that can lead to this node in shortest time

For example, if we can reach node C in shortest time from both node A and node B:

- ways[C] = ways[A] + ways[B]

Modified Dijkstra's Logic:

1. **Shorter path found**: Update distance and reset ways count
2. **Equal path found**: Keep same distance but add to ways count
3. **Longer path found**: Ignore (standard Dijkstra's behavior)

This works because when we process nodes in order of shortest distance (thanks to priority queue), we guarantee that we've found the optimal way to reach each node.

Approach Steps

1. **Initialize data structures**:
    - Distance array with infinity for all nodes except source (set to 0)
    - Ways array with 0 for all nodes except source (set to 1)
    - Priority queue for Dijkstra's algorithm
2. **Run modified Dijkstra's**:
    - Process nodes in order of shortest distance
    - For each neighbor, check if we found shorter, equal, or longer path
3. **Handle three cases**:
    - **Shorter path**: Update distance and reset ways count
    - **Equal path**: Keep distance, add to ways count
    - **Longer path**: Ignore
4. **Apply modulo**: Use modulo 10^9+7 to prevent integer overflow
5. **Return result**: Return ways[destination]

Code

Method 1: Modified Implementation (Based on Provided Code)

```python
import heapq

MOD = 10**9 + 7

def solution(nodes, graph, src, dest):
    # Initialize distance and ways arrays
    distance = {i: float('inf') for i in nodes}
    ways = {i: 0 for i in nodes}

    # Priority queue: (distance, node)
    queue = [(0, src)]
    distance[src] = 0
    ways[src] = 1

    while queue:
        dist, node = heapq.heappop(queue)

        # Skip if we've already processed this node with better distance
        if dist > distance[node]:
            continue

        # Process all neighbors
        for adj_node, adj_weight in graph[node].items():
            new_dist = distance[node] + adj_weight

            if new_dist < distance[adj_node]:
                # Found shorter path: update distance and reset ways
                distance[adj_node] = new_dist
                ways[adj_node] = ways[node]
                heapq.heappush(queue, (new_dist, adj_node))

            elif new_dist == distance[adj_node]:
                # Found equal path: add to ways count
                ways[adj_node] = (ways[node] + ways[adj_node]) % MOD

    return ways[dest]

class Solution:
    def countPaths(self, n, roads):
        nodes = [i for i in range(n)]
        graph = {i: {} for i in nodes}

        # Build undirected graph
        for u, v, weight in roads:
            graph[u][v] = weight
            graph[v][u] = weight

        return solution(nodes, graph, 0, n - 1)
```

Method 2: Clean Standard Implementation

```python
import heapq

def countPaths(n, roads):
```

```
    MOD = 10**9 + 7

    # Build adjacency list
    graph = [[] for _ in range(n)]
    for u, v, time in roads:
        graph[u].append((v, time))
        graph[v].append((u, time))

    # Dijkstra's with path counting
    dist = [float('inf')] * n
    ways = [0] * n

    dist[0] = 0
    ways[0] = 1

    pq = [(0, 0)]  # (distance, node)

    while pq:
        d, u = heapq.heappop(pq)

        # Skip outdated entries
        if d > dist[u]:
            continue

        for v, time in graph[u]:
            new_dist = dist[u] + time

            if new_dist < dist[v]:
                # Shorter path found
                dist[v] = new_dist
                ways[v] = ways[u]
                heapq.heappush(pq, (new_dist, v))

            elif new_dist == dist[v]:
                # Equal path found
                ways[v] = (ways[v] + ways[u]) % MOD

    return ways[n - 1]

class Solution:
    def countPaths(self, n, roads):
        return countPaths(n, roads)
```

Method 3: With Visited Set (Alternative)

```
import heapq

def countPathsWithVisited(n, roads):
    MOD = 10**9 + 7

    # Build graph
    graph = [[] for _ in range(n)]
    for u, v, time in roads:
        graph[u].append((v, time))
        graph[v].append((u, time))

    # Initialize arrays
    dist = [float('inf')] * n
    ways = [0] * n
    processed = [False] * n
```

```
    dist[0] = 0
    ways[0] = 1

    pq = [(0, 0)]

    while pq:
        d, u = heapq.heappop(pq)

        if processed[u]:
            continue

        processed[u] = True

        for v, time in graph[u]:
            new_dist = dist[u] + time

            if new_dist < dist[v]:
                dist[v] = new_dist
                ways[v] = ways[u]
                if not processed[v]:
                    heapq.heappush(pq, (new_dist, v))

            elif new_dist == dist[v] and not processed[v]:
                ways[v] = (ways[v] + ways[u]) % MOD

    return ways[n - 1]
```

Method 4: Debug Version (For Understanding)

```
import heapq

def countPathsDebug(n, roads):
    MOD = 10**9 + 7

    graph = [[] for _ in range(n)]
    for u, v, time in roads:
        graph[u].append((v, time))
        graph[v].append((u, time))

    dist = [float('inf')] * n
    ways = [0] * n

    dist[0] = 0
    ways[0] = 1

    pq = [(0, 0)]

    while pq:
        d, u = heapq.heappop(pq)

        if d > dist[u]:
            continue

        print(f"Processing node {u} with distance {d}")

        for v, time in graph[u]:
            new_dist = dist[u] + time

            if new_dist < dist[v]:
```

```
                print(f"  Shorter path to {v}: {new_dist} (was {dist[v]})")
                dist[v] = new_dist
                ways[v] = ways[u]
                heapq.heappush(pq, (new_dist, v))

            elif new_dist == dist[v]:
                print(f"  Equal path to {v}: {new_dist}, adding {ways[u]} ways")
                ways[v] = (ways[v] + ways[u]) % MOD

    print(f"Final distances: {dist}")
    print(f"Final ways: {ways}")

    return ways[n - 1]
```

Time and Space Complexity

Time Complexity: O((V + E) log V)

- Same as standard Dijkstra's algorithm
- Each vertex is processed at most once when extracted from priority queue
- Each edge relaxation involves heap operations
- V is number of intersections, E is number of roads

Space Complexity: O(V + E)

- Adjacency list representation: O(E) space
- Distance and ways arrays: O(V) space each
- Priority queue: O(V) space in worst case
- Overall: O(V + E)

Key Points to Remember

Why modulo is needed:

- Number of paths can grow exponentially
- Without modulo, integer overflow would occur
- Apply modulo only when adding ways, not when comparing distances

Critical insight:

- We only add ways when distances are equal
- This ensures we're only counting shortest paths
- The ways array represents number of shortest paths to each node

Edge cases:

- Direct path exists (might not be shortest)
- Multiple equal shortest paths
- No path exists (shouldn't happen given problem constraints)
- Single node graph (n=1)

Common mistakes:

- Forgetting to apply modulo
- Adding ways for longer paths (should only add for equal distances)
- Not handling duplicate entries in priority queue properly

Real-world applications:

- GPS routing systems (finding all fastest routes)
- Network routing (load balancing across equal-cost paths)
- Transportation optimization
- Supply chain route planning

Word Ladder I

Problem Description

Given a startWord and a targetWord, along with a list of unique words (wordList), find the length of the shortest transformation sequence from startWord to targetWord. In each transformation, only one letter can be changed, and each intermediate word must exist in the wordList. If no transformation sequence exists, return 0.

Think of it like a word puzzle where you change one letter at a time to transform one word into another, but every intermediate step must be a valid word from your dictionary.

Examples

Input

```
wordList = ["des","der","dfr","dgt","dfs"], startWord = "der", targetWord = "dfs"
```

Output

```
3
```

Explanation

Transformation sequence: "der" → "dfr" → "dfs"

- Step 1: "der" (starting word)
- Step 2: "dfr" (replace 'e' with 'f')
- Step 3: "dfs" (replace 'r' with 's') Total length: 3

Input

```
wordList = ["geek", "gefk"], startWord = "gedk", targetWord = "geek"
```

Output

```
2
```

Explanation

Transformation sequence: "gedk" → "geek"

- Step 1: "gedk" (starting word)
- Step 2: "geek" (replace 'd' with 'e') Total length: 2

Input

```
wordList = ["hot", "dot", "dog", "lot", "log"], startWord = "hit", targetWord = "cog"
```

Output

```
0
```

Explanation

No transformation sequence possible since "cog" is not in the wordList.

Solution

Use BFS (Breadth-First Search) to explore all possible word transformations level by level. BFS guarantees finding the shortest transformation sequence because it explores all words reachable in k steps before exploring words reachable in k+1 steps.

Intuition

This is a shortest path problem in an unweighted graph where:

- **Nodes**: Words (startWord + all words in wordList)
- **Edges**: Direct transformations (words differing by exactly one character)
- **Goal**: Find shortest path from startWord to targetWord

Why BFS works perfectly:

1. **Unweighted graph**: Each transformation has equal "cost" (1 step)
2. **Level-wise exploration**: BFS explores all words reachable in 1 step, then 2 steps, then 3 steps, etc.
3. **First occurrence = shortest path**: When we first reach the targetWord, we've found the shortest sequence

Key optimization: Remove words from wordList once visited to avoid cycles and redundant processing.

Approach Steps

1. **Setup data structures**:
    - Convert wordList to set for O(1) lookup and removal
    - Initialize BFS queue with (startWord, steps=1)
    - Remove startWord from set if present (avoid revisiting)
2. **BFS traversal**:
    - Process words level by level using queue
    - For each word, try changing each character position
3. **Generate transformations**:
    - For each position i, try all 26 letters ('a' to 'z')
    - Create new word by replacing character at position i
4. **Valid transformation check**:
    - If new word exists in wordList set, it's a valid transformation
    - Add to queue with incremented step count
    - Remove from set to mark as visited
5. **Target check**: If current word equals targetWord, return step count

Code

Method 1: Standard BFS Implementation (Based on Provided Code)

```
from collections import deque

def solution(wordList, startWord, targetWord):
    # Initialize BFS queue and convert wordList to set
    queue = deque()
    queue.append((startWord, 1))
    wordSet = set(wordList)

    # Remove startWord from set to avoid revisiting
    wordSet.discard(startWord)

    while queue:
        word, steps = queue.popleft()

        # Check if we reached target
        if word == targetWord:
            return steps

        # Try all possible single character transformations
        for i in range(len(word)):
            for char in 'abcdefghijklmnopqrstuvwxyz':
                newWord = word[:i] + char + word[i+1:]

                # If transformation is valid and exists in wordList
                if newWord in wordSet:
                    queue.append((newWord, steps + 1))
```

```
                    wordSet.discard(newWord)  # Mark as visited

    return 0  # No transformation sequence found

class Solution:
    def wordLadderLength(self, startWord, targetWord, wordList):
        return solution(wordList, startWord, targetWord)
```

Method 2: Early Termination Check

```
from collections import deque

def wordLadderLength(startWord, targetWord, wordList):
    # Early check: if targetWord not in wordList, return 0
    if targetWord not in wordList:
        return 0

    # Convert to set for O(1) operations
    wordSet = set(wordList)
    queue = deque([(startWord, 1)])
    wordSet.discard(startWord)

    while queue:
        word, steps = queue.popleft()

        if word == targetWord:
            return steps

        # Generate all possible transformations
        for i in range(len(word)):
            for c in 'abcdefghijklmnopqrstuvwxyz':
                newWord = word[:i] + c + word[i+1:]

                if newWord in wordSet:
                    if newWord == targetWord:
                        return steps + 1
                    queue.append((newWord, steps + 1))
                    wordSet.remove(newWord)

    return 0
```

Method 3: Bidirectional BFS (Advanced Optimization)

```
from collections import deque

def wordLadderBidirectional(startWord, targetWord, wordList):
    if targetWord not in wordList:
        return 0

    wordSet = set(wordList)

    # Two queues for bidirectional search
    beginQueue = deque([(startWord, 1)])
    endQueue = deque([(targetWord, 1)])

    beginVisited = {startWord: 1}
    endVisited = {targetWord: 1}
```

```
    def getNeighbors(word):
        neighbors = []
        for i in range(len(word)):
            for c in 'abcdefghijklmnopqrstuvwxyz':
                newWord = word[:i] + c + word[i+1:]
                if newWord in wordSet:
                    neighbors.append(newWord)
        return neighbors

    def bfsLevel(queue, visited, otherVisited):
        for _ in range(len(queue)):
            word, steps = queue.popleft()

            for neighbor in getNeighbors(word):
                if neighbor in otherVisited:
                    return steps + otherVisited[neighbor]

                if neighbor not in visited:
                    visited[neighbor] = steps + 1
                    queue.append((neighbor, steps + 1))
        return 0

    while beginQueue and endQueue:
        # Always expand the smaller queue
        if len(beginQueue) <= len(endQueue):
            result = bfsLevel(beginQueue, beginVisited, endVisited)
        else:
            result = bfsLevel(endQueue, endVisited, beginVisited)

        if result:
            return result

    return 0
```

Method 4: With Path Tracking (For Learning)

```
from collections import deque

def wordLadderWithPath(startWord, targetWord, wordList):
    if targetWord not in wordList:
        return 0, []

    wordSet = set(wordList)
    queue = deque([(startWord, 1, [startWord])])
    wordSet.discard(startWord)

    while queue:
        word, steps, path = queue.popleft()

        if word == targetWord:
            return steps, path

        for i in range(len(word)):
            for c in 'abcdefghijklmnopqrstuvwxyz':
                newWord = word[:i] + c + word[i+1:]

                if newWord in wordSet:
                    newPath = path + [newWord]
                    queue.append((newWord, steps + 1, newPath))
                    wordSet.remove(newWord)
```

```
        return 0, []
```

Time and Space Complexity

Time Complexity: O($M^2 \times N$)

- M = length of each word
- N = number of words in wordList
- For each word, we try M positions × 26 characters = 26M transformations
- Each transformation takes O(M) time to create new string
- We might visit all N words: O(26M × M × N) = O($M^2 \times N$)

Space Complexity: O(N × M)

- WordSet storage: O(N × M) for storing all words
- BFS queue: O(N) in worst case
- Each word takes O(M) space
- Overall: O(N × M)

Key Points to Remember

Why BFS over DFS?

- BFS explores level by level, guaranteeing shortest path
- DFS might find a valid path but not necessarily the shortest one
- In unweighted graphs, BFS always finds optimal solution first

Critical optimization - removing visited words:

- Prevents infinite loops and revisiting
- Safe because: if we reach a word in k steps, any future path to it will be ≥ k steps
- Uses O(1) set operations for efficiency

Edge cases:

- targetWord not in wordList → return 0
- startWord equals targetWord → return 1
- Empty wordList → return 0
- No valid transformation sequence exists → return 0

Common mistakes:

- Forgetting to remove visited words (leads to infinite loops)
- Using list instead of set for wordList (TLE due to O(n) lookups)
- Not handling case where startWord is not in wordList
- Counting steps incorrectly (should count words in sequence, not transformations)

Real-world applications:

- Spell checkers (finding closest valid words)

- DNA sequence analysis (mutations with single base changes)
- Text processing and natural language processing
- Game development (word puzzles and transformations)

Word Ladder II

Problem Description

Given a startWord and a targetWord, along with a list of unique words (wordList), find all shortest transformation sequences from startWord to targetWord. In each transformation, only one letter can be changed, and each intermediate word must exist in the wordList. Return all possible shortest paths, or an empty list if no transformation sequence exists.

Think of it like finding all the shortest routes in a word puzzle - not just one optimal path, but every possible way to get from start to finish in the minimum number of steps.

Examples

Input

```
startWord = "der", targetWord = "dfs", wordList = ["des", "der", "dfr", "dgt", "dfs"]
```

Output

```
[["der", "dfr", "dfs"], ["der", "des", "dfs"]]
```

Explanation

Two shortest transformation sequences of length 3:

1. "der" → "dfr" → "dfs" (change 'e'→'f', then 'r'→'s')
2. "der" → "des" → "dfs" (change 'r'→'s', then 'e'→'f')

Input

```
startWord = "gedk", targetWord = "geek", wordList = ["geek", "gefk"]
```

Output

```
[["gedk", "geek"]]
```

Explanation

Only one shortest transformation sequence of length 2:

- "gedk" → "geek" (change 'd'→'e')

Solution

Use a two-phase approach: First, build a level-wise graph using BFS to find the shortest distance to all words. Then, use DFS with backtracking to reconstruct all shortest paths from target back to source.

Intuition

Word Ladder II is more complex than Word Ladder I because we need ALL shortest paths, not just the length. The key challenges are:

1. **Multiple paths**: We can't stop at first occurrence of target word
2. **Path reconstruction**: Need to remember how to build the actual sequences
3. **Efficiency**: Avoid exponential path enumeration during BFS

Two-phase approach:

- **Phase 1 (BFS)**: Build a level-wise map showing shortest distance to reach each word
- **Phase 2 (DFS)**: Reconstruct all paths by traversing from target back to source

Why this works:

- BFS ensures we find the shortest distance to each word
- Level information helps us only follow edges that are part of shortest paths
- DFS with backtracking efficiently explores all valid paths

Approach Steps

1. **Phase 1 - BFS to build level map**:
    - Use BFS to explore all reachable words level by level
    - Record the shortest distance (level) to reach each word
    - Continue until all reachable words are processed
2. **Phase 2 - DFS path reconstruction**:
    - Start DFS from target word
    - Only move to words that are one level closer to start
    - Use backtracking to explore all valid paths
3. **Level constraint**: Only follow edges where `level[current] = level[next] + 1`
4. **Path building**: Collect complete paths and reverse them (since we go target→start)
5. **Return result**: All collected shortest transformation sequences

Code

Method 1: BFS + DFS Approach (Based on Provided Code)

```
from collections import deque, defaultdict

def solution(wordList, startWord, endWord):
    def dfs(word, path):
        if word == startWord:
            ans.append(path[::-1])  # Reverse path since we go target->start
            return

        # Try all single character transformations
        for i in range(len(word)):
            for char in "abcdefghijklmnopqrstuvwxyz":
                newWord = word[:i] + char + word[i+1:]

                # Only follow edges that are part of shortest paths
                if newWord in levels and levels[word] == (levels[newWord] + 1):
                    dfs(newWord, path + [newWord])

    # Phase 1: BFS to build level map
    queue = deque([(startWord, 1)])
    wordSet = set(wordList)
    wordSet.discard(startWord)

    levels = defaultdict(int)
    levels[startWord] = 1

    while queue:
        word, steps = queue.popleft()

        for i in range(len(word)):
            for char in "abcdefghijklmnopqrstuvwxyz":
                newWord = word[:i] + char + word[i+1:]

                if newWord in wordSet:
                    queue.append((newWord, steps + 1))
                    levels[newWord] = steps + 1
                    wordSet.discard(newWord)

    # Phase 2: DFS to reconstruct all paths
    ans = []
    if endWord in levels:
        dfs(endWord, [endWord])

    return ans

class Solution:
    def findSequences(self, beginWord, endWord, wordList):
        return solution(wordList, beginWord, endWord)
```

Method 2: Clean BFS + Backtracking Implementation

```
from collections import deque, defaultdict

def findLadders(beginWord, endWord, wordList):
    if endWord not in wordList:
        return []

    wordSet = set(wordList)

    # BFS to build parent-child relationships
```

```
    queue = deque([beginWord])
    visited = {beginWord}
    found = False
    parent = defaultdict(list)

    while queue and not found:
        level_visited = set()

        for _ in range(len(queue)):
            word = queue.popleft()

            for i in range(len(word)):
                for c in 'abcdefghijklmnopqrstuvwxyz':
                    newWord = word[:i] + c + word[i+1:]

                    if newWord in wordSet and newWord not in visited:
                        if newWord == endWord:
                            found = True

                        if newWord not in level_visited:
                            level_visited.add(newWord)
                            queue.append(newWord)

                        parent[newWord].append(word)

        visited.update(level_visited)

    # DFS to construct all paths
    def dfs(word, path, result):
        if word == beginWord:
            result.append([beginWord] + path[::-1])
            return

        for p in parent[word]:
            dfs(p, path + [word], result)

    result = []
    if found:
        dfs(endWord, [], result)

    return result
```

Method 3: Level-by-Level BFS with Path Tracking

```
from collections import deque

def findLaddersLevelwise(beginWord, endWord, wordList):
    if endWord not in wordList:
        return []

    wordSet = set(wordList)

    # Start with initial word and its path
    queue = deque([[beginWord]])
    visited = set([beginWord])
    result = []

    while queue:
        level_visited = set()
```

```
        # Process all paths at current level
        for _ in range(len(queue)):
            path = queue.popleft()
            word = path[-1]

            # If we reached target, this level has shortest paths
            if word == endWord:
                result.append(path)
                continue

            # Generate transformations for current word
            for i in range(len(word)):
                for c in 'abcdefghijklmnopqrstuvwxyz':
                    newWord = word[:i] + c + word[i+1:]

                    if newWord in wordSet and newWord not in visited:
                        level_visited.add(newWord)
                        queue.append(path + [newWord])

        # If we found target at this level, stop (all shortest found)
        if result:
            break

        # Mark level words as visited
        visited.update(level_visited)

    return result
```

Method 4: Optimized Bidirectional BFS + DFS

```
from collections import defaultdict, deque

def findLaddersBidirectional(beginWord, endWord, wordList):
    if endWord not in wordList:
        return []

    wordSet = set(wordList)

    # Build adjacency graph first
    neighbors = defaultdict(list)
    for word in [beginWord] + wordList:
        for i in range(len(word)):
            pattern = word[:i] + '*' + word[i+1:]
            neighbors[pattern].append(word)

    def bfs():
        queue = deque([beginWord])
        visited = {beginWord: [[beginWord]]}

        while queue:
            word = queue.popleft()

            if word == endWord:
                return visited[endWord]

            for i in range(len(word)):
                pattern = word[:i] + '*' + word[i+1:]

                for nextWord in neighbors[pattern]:
                    if nextWord == word:
```

```
                    continue

                if nextWord not in visited:
                    visited[nextWord] = []
                    queue.append(nextWord)

                # Add all paths that lead to current word
                for path in visited[word]:
                    if nextWord not in path:  # Avoid cycles
                        visited[nextWord].append(path + [nextWord])

        return []

    return bfs()
```

Time and Space Complexity

Time Complexity: O(N × M² × 26^L)

- N = number of words in wordList
- M = length of each word
- L = length of shortest path
- BFS phase: O(N × M² × 26) to explore all transformations
- DFS phase: O(26^L) in worst case for path reconstruction
- Overall dominated by path enumeration

Space Complexity: O(N × M + P × L)

- N × M for storing wordList and level information
- P × L where P is number of shortest paths and L is path length
- In worst case, could be exponential in number of paths

Key Points to Remember

Differences from Word Ladder I:

- **Multiple paths**: Need all shortest sequences, not just length
- **No early termination**: Can't stop at first occurrence of target
- **Path reconstruction**: Must remember actual transformation sequences
- **Delayed removal**: Can't remove words immediately to allow multiple paths

Critical optimizations:

- **Two-phase approach**: BFS for distances, DFS for paths
- **Level constraint**: Only follow edges that are part of shortest paths
- **Backtracking**: Efficiently explore all valid path combinations

Common pitfalls:

- **Exponential explosion**: Naive path tracking during BFS leads to TLE
- **Incorrect path reconstruction**: Must reverse paths when going target→start
- **Missing shortest paths**: Removing words too early can miss valid sequences

Edge cases:

- No transformation sequence exists → return []
- Multiple shortest paths of different structures
- Start word equals target word → return [[startWord]]
- Target word not in wordList → return []

Interview tips:

- Explain the two-phase approach clearly
- Mention why naive BFS with path tracking fails
- Discuss time complexity trade-offs
- Consider bidirectional BFS for very long paths

Real-world applications:

- Finding all optimal solutions in puzzle games
- Route planning with multiple equally good options
- Genetic algorithm pathways in bioinformatics
- Natural language processing for word similarity

Minimum multiplications to reach end

Given a starting number, an ending number, and an array of numbers, find the minimum number of steps to transform the start number into the end number. At each step, you can multiply the current number by any number from the array, then take modulo 100000 to get the new number. If it's impossible to reach the end number, return -1.

Examples

Input

```
arr = [2, 5, 7], start = 3, end = 30
```

Output

```
2
```

Explanation

Think of this like a game where you're trying to reach a target number by multiplying your current number with choices from a given array:

- **Step 1:** Start with 3, multiply by 2 → 3 × 2 = 6 % 100000 = 6
- **Step 2:** Now with 6, multiply by 5 → 6 × 5 = 30 % 100000 = 30 ✓

We reached our target (30) in just 2 steps! It's like finding the shortest route in a maze where each room represents a number, and each door represents a multiplication operation.

Input

```
arr = [3, 4, 65], start = 7, end = 66175
```

Output

```
4
```

Explanation

- **Step 1:** 7 × 3 = 21 % 100000 = 21
- **Step 2:** 21 × 3 = 63 % 100000 = 63
- **Step 3:** 63 × 65 = 4095 % 100000 = 4095
- **Step 4:** 4095 × 65 = 266175 % 100000 = 66175 ✓

Solution

This is a **shortest path problem in disguise**! We use BFS (Breadth-First Search) to find the minimum number of steps. Each number we can reach is like a "node" in a graph, and multiplying by array elements creates "edges" to new nodes.

Intuition

Why think of this as a graph problem?

- **Nodes:** Each possible number (0 to 99999) is a node
- **Edges:** Multiplying by any array element and taking mod 100000 creates an edge to a new node
- **Goal:** Find shortest path from start node to end node

Why BFS works perfectly:

- BFS explores all possibilities level by level
- The first time we reach the target, we've found the minimum steps
- BFS guarantees shortest path in unweighted graphs (each step has cost 1)

Approach Steps

1. **Handle base case:** If start equals end, return 0 steps
2. **Initialize BFS setup:**
    - Create a queue starting with (start_number, 0_steps)
    - Create an array to track minimum steps to reach each number (0-99999)
3. **BFS traversal:**

    - Pop current number and step count from queue
    - Try multiplying with each element in the array
    - For each result after mod 100000:
        - If it equals target → return steps + 1
        - If we found a shorter path → update and add to queue
4. **Return -1 if target unreachable**

Code

```
def solution(arr, start, end):
    # Base case: already at target
    if start == end:
        return 0

    # BFS setup
    queue = [(start, 0)]  # (current_number, steps_taken)
    min_steps = [float('inf')] * 100000  # Track minimum steps to reach each number
    min_steps[start] = 0

    while queue:
        current_num, steps = queue.pop(0)

        # Try multiplying with each number in array
        for multiplier in arr:
            new_num = (current_num * multiplier) % 100000

            # Found target!
            if new_num == end:
                return steps + 1

            # Found shorter path to this number
            if steps + 1 < min_steps[new_num]:
                min_steps[new_num] = steps + 1
                queue.append((new_num, steps + 1))

    # Target unreachable
    return -1

class Solution:
    def minimumMultiplications(self, arr, start, end):
        return solution(arr, start, end)
```

Time and Space Complexity

Time Complexity: O(100000 × M)

- We might visit each of the 100000 possible numbers (0-99999) at most once
- For each number, we try M multipliers from the array
- In simple terms: We explore at most 100,000 numbers, and for each number we try all array elements

Space Complexity: O(100000)

- We use an array of size 100000 to track minimum steps
- The queue can hold at most 100000 elements in worst case

- In simple terms: We need memory proportional to the range of possible numbers (0-99999)

Shortest Distance in a Binary Maze

You are given an `n x m` grid (matrix) with cells containing either `0` or `1`. You can move from a cell to its **up, down, left, or right** neighbor **only if** that neighbor has value `1`.Given a **source** cell and a **destination** cell, find the **shortest number of steps** to reach the destination from the source. If the destination is not reachable, return `-1`.

Examples

Example 1

Input

```
grid = [
  [1, 1, 1, 1],
  [1, 1, 0, 1],
  [1, 1, 1, 1],
  [1, 1, 0, 0],
  [1, 0, 0, 1]
]
source = [0, 1]
destination = [2, 2]
```

Output

```
3
```

Explanation

A shortest path from `(0, 1)` to `(2, 2)` is:

- Down to `(1, 1)`
- Down to `(2, 1)`
- Right to `(2, 2)`

This takes **3** moves.

Analogy: Imagine walking on tiles where only tiles marked `1` are solid. You move one tile at a time (up/down/left/right). The shortest path is just the minimum number of steps on solid tiles to reach the target.

Example 2

Input

```
grid = [
  [1, 1, 1, 1, 1],
  [1, 1, 1, 1, 1],
  [1, 1, 1, 1, 0],
  [1, 0, 1, 0, 1]
]
source = [0, 0]
destination = [3, 4]
```

Output

```
-1
```

Explanation

There is no valid path of `1` s from the source to the destination. Hence the answer is `-1` .

Solution

Use **Breadth-First Search (BFS)** starting from the source:

- BFS explores the grid **level by level** (i.e., all cells at distance 1, then distance 2, and so on).
- Because every move costs the same (1 step), the **first time** we reach the destination, it is **guaranteed** to be via the shortest path.
- Maintain a **distance matrix** to store the shortest steps taken to reach each cell.

Intuition

- Dijkstra's algorithm is a general tool for shortest paths with varying edge weights.
- Here, **every move has equal cost (1)**, so **BFS** is enough and **faster/simpler**:
  - Use a **queue** (not a min-heap).
  - Once you pop a cell from the queue, all neighbors reachable in one more step get updated.

Neighbor traversal trick: Define direction arrays to iterate neighbors efficiently:

- `delRow = [-1, 0, 1, 0]`
- `delCol = [0, 1, 0, -1]` Each pair `(delRow[k], delCol[k])` gives one of the 4 directions.

Approach Steps

1. **Validate**:
   - If grid is empty → return `-1` .
   - If source or destination is outside the grid or lies on a `0` → return `-1` .

- If source == destination → return `0`.

2. **Initialize**:
   - Let `n, m` be grid dimensions.
   - `dist[n][m]` initialized to `∞` (or a large number).
   - Set `dist[source] = 0`.
   - Push `source` into a queue.
3. **BFS Loop**:
   - While the queue is not empty:
     - Pop `(r, c)`.
     - For each of the 4 neighbors `(nr, nc)`:
       - Check bounds and that `grid[nr][nc] == 1`.
       - If `dist[r][c] + 1 < dist[nr][nc]`:
         - Update `dist[nr][nc]`.
         - If `(nr, nc)` is the destination → **return** `dist[nr][nc]` (early exit).
         - Push `(nr, nc)` into the queue.
4. **End**:
   - If BFS finishes without reaching destination → return `-1`.

Edge Cases:

- Source equals destination → `0`.
- Source/destination on a `0` → `-1`.
- Grid with single cell → depends on that cell and equality of source/destination.

Code

```python
from collections import deque
from typing import List, Tuple

def shortest_distance_binary_maze(
    grid: List[List[int]],
    source: Tuple[int, int],
    destination: Tuple[int, int]
) -> int:
    """
    Returns the shortest number of steps from source to destination in a binary grid,
    moving only in 4 directions through cells with value 1. If unreachable, returns -1.
    """
    # Basic validation
    if not grid or not grid[0]:
        return -1

    n, m = len(grid), len(grid[0])
    sr, sc = source
```

```
    dr, dc = destination

    # Check bounds
    if not (0 <= sr < n and 0 <= sc < m and 0 <= dr < n and 0 <= dc < m):
        return -1

    # Must start/end on 1-cells
    if grid[sr][sc] == 0 or grid[dr][dc] == 0:
        return -1

    # Trivial case
    if (sr, sc) == (dr, dc):
        return 0

    # Distance matrix
    INF = 10**9
    dist = [[INF] * m for _ in range(n)]
    dist[sr][sc] = 0

    # Direction vectors: up, right, down, left
    directions = [(-1, 0), (0, 1), (1, 0), (0, -1)]

    # BFS queue
    q = deque([(sr, sc)])

    while q:
        r, c = q.popleft()
        current = dist[r][c]

        for drc, dcc in directions:
            nr, nc = r + drc, c + dcc
            # Check valid neighbor and passable cell
            if 0 <= nr < n and 0 <= nc < m and grid[nr][nc] == 1:
                # Relax edge if shorter
                if current + 1 < dist[nr][nc]:
                    dist[nr][nc] = current + 1
                    # Early exit if destination reached
                    if (nr, nc) == (dr, dc):
                        return dist[nr][nc]
                    q.append((nr, nc))

    return -1


# Optional class wrapper (common on coding platforms)
class Solution:
    def shortestPath(self, grid: List[List[int]], source: List[int], destination: List[int]) -> int:
        return shortest_distance_binary_maze(
            grid,
            (source[0], source[1]),
            (destination[0], destination[1])
        )
```

Time and Space Complexity

- **Time Complexity:** Each cell is processed at most once and we check up to 4 neighbors → **O(n × m)**.
- **Space Complexity:** Distance matrix `dist` uses `O(n × m)` space, and the queue holds at most `O(n × m)` cells → **O(n × m)**.

In simple words: We visit each cell at most one time and keep distances for each cell, so both time and memory grow with the number of cells in the grid.

Path with minimum effort

We are given a 2D grid `heights` where each cell's value represents the height of that point. The hiker starts at the top-left cell `(0,0)` and wants to reach the bottom-right cell `(rows-1, columns-1)`. He can move up, down, left, or right.

The **effort** of a route is defined as the maximum absolute difference in heights between any two consecutive cells in the route. Our goal is to find a route that minimizes this maximum effort.

Examples

Input

```
heights = [[1,2,2],[3,8,2],[5,3,5]]
```

Output

```
2
```

Explanation

One optimal path is `[1 → 3 → 5 → 3 → 5]`, where the largest step difference is `2`. This is better than `[1 → 2 → 2 → 2 → 5]`, which has a maximum step difference of `3`.

Input

```
heights = [[1,2,3],[3,8,4],[5,3,5]]
```

Output

```
1
```

Explanation

One optimal path is `[1 → 2 → 3 → 4 → 5]`, where the largest step difference is `1`.

Solution

We can adapt **Dijkstra's algorithm** to solve this:

- Instead of summing distances, we track the **maximum difference** seen so far.
- We use a **min-heap (priority queue)** to always expand the path with the smallest current effort.
- For each move, the new effort is `max(current_effort, height_diff)`.
- The first time we reach the destination, the effort is minimal.

Intuition

This is like normal Dijkstra, but with a twist: instead of adding edge weights, we take the **maximum** between the path's current effort and the next step's height difference. BFS would not work directly here because edge weights (differences) are not uniform.

Approach Steps

1. Create a `difference` matrix initialized to infinity, storing the minimal effort needed to reach each cell.
2. Use a min-heap and push `(effort=0, row=0, col=0)`.
3. While heap is not empty:
    - Pop the cell with smallest effort.
    - If it's the destination, return effort.
    - For each valid neighbor, calculate `new_effort = max(current_effort, abs(height_diff))`.
    - If `new_effort` is smaller than the stored value, update and push to heap.
4. Return the result when destination is reached.

Code

```
import heapq

def minimum_effort_path(heights):
    n, m = len(heights), len(heights[0])
    difference = [[float('inf')] * m for _ in range(n)]
    difference[0][0] = 0
    pq = [(0, 0, 0)]  # (effort, row, col)

    directions = [(1,0), (-1,0), (0,-1), (0,1)]

    while pq:
        effort, r, c = heapq.heappop(pq)
        if (r, c) == (n-1, m-1):
            return effort
        for dr, dc in directions:
            nr, nc = r + dr, c + dc
            if 0 <= nr < n and 0 <= nc < m:
```

```
                curr_diff = abs(heights[nr][nc] - heights[r][c])
                new_effort = max(effort, curr_diff)
                if new_effort < difference[nr][nc]:
                    difference[nr][nc] = new_effort
                    heapq.heappush(pq, (new_effort, nr, nc))
    return -1

# Example usage
print(minimum_effort_path([[1,2,2],[3,8,2],[5,3,5]]))  # Output: 2
print(minimum_effort_path([[1,2,3],[3,8,4],[5,3,5]]))  # Output: 1
```

Time and Space Complexity

- **Time Complexity:** `O(n*m*log(n*m))` because each cell is pushed/popped from the priority queue at most once.
- **Space Complexity:** `O(n*m)` for the `difference` matrix and priority queue.

Bellman-Ford Algorithm

Problem Description

Given a weighted and directed graph with V vertices and E edges, find the shortest distance from a source vertex S to all other vertices. Each edge is represented as [source, destination, weight]. If a vertex cannot be reached from the source, mark its distance as 10^9. If the graph contains a negative cycle (a cycle where the sum of all weights is negative), return [-1].

Examples

Example 1

Input

```
V = 6
Edges = [[3, 2, 6], [5, 3, 1], [0, 1, 5], [1, 5, -3], [1, 2, -2], [3, 4, -2], [2, 4, 3]]
S = 0
```

Output

```
[0, 5, 3, 3, 1, 2]
```

Explanation

- **Node 0 (source)**: Distance = 0
- **Node 1**: Shortest path is 0→1, distance = 5
- **Node 2**: Shortest path is 0→1→2, distance = 5 + (-2) = 3

- **Node 3**: Shortest path is 0→1→5→3, distance = 5 + (-3) + 1 = 3
- **Node 4**: Shortest path is 0→1→5→3→4, distance = 5 + (-3) + 1 + (-2) = 1
- **Node 5**: Shortest path is 0→1→5, distance = 5 + (-3) = 2

Example 2

Input

```
V = 2
Edges = [[0, 1, 9]]
S = 0
```

Output

```
[0, 9]
```

Explanation

- **Node 0 (source)**: Distance = 0
- **Node 1**: Shortest path is 0→1, distance = 9

Solution

The Bellman-Ford algorithm finds shortest paths from a source vertex to all other vertices in a weighted graph, even when the graph contains negative weight edges. It can also detect negative cycles in the graph.

Intuition

Think of the Bellman-Ford algorithm like this:

- Imagine you're trying to find the cheapest way to travel from one city to all other cities
- Some routes might have "negative costs" (like cashback offers)
- You need to check all possible routes multiple times to ensure you found the absolute cheapest path
- The algorithm is like a patient traveler who checks every possible route exactly (V-1) times
- If after (V-1) checks you can still find a cheaper route, it means there's an infinite money-making loop (negative cycle)

Why this approach works:

1. **Edge Relaxation**: If going through city A to reach city B is cheaper than the current known way to reach B, update the cost to reach B
2. **V-1 iterations**: In a graph with V vertices, the longest simple path can have at most (V-1) edges
3. **Negative cycle detection**: If we can still improve distances after V-1 iterations, there must be a negative cycle

Approach Steps

1. **Initialize distances**: Set source distance to 0, all others to infinity

2. **Relax edges V-1 times**: For each iteration, go through all edges and update distances if a shorter path is found
3. **Check for negative cycles**: Run one more iteration - if any distance can still be improved, a negative cycle exists
4. **Handle unreachable vertices**: Convert remaining infinity values to 10^9
5. **Return result**: Return the distance array or [-1] if negative cycle found

Code

```python
class Solution:
    def bellman_ford(self, V, edges, S):
        # Step 1: Initialize distances
        distance = [float('inf')] * V
        distance[S] = 0

        # Step 2: Relax edges V-1 times
        for _ in range(V - 1):
            for source, dest, weight in edges:
                # Edge relaxation: if we can reach dest through source with less cost
                if distance[source] != float('inf') and distance[source] + weight < distance[dest]:
                    distance[dest] = distance[source] + weight

        # Step 3: Check for negative cycles
        for source, dest, weight in edges:
            if distance[source] != float('inf') and distance[source] + weight < distance[dest]:
                return [-1]  # Negative cycle detected

        # Step 4: Handle unreachable vertices
        for i in range(V):
            if distance[i] == float('inf'):
                distance[i] = 10**9

        return distance
```

Time and Space Complexity

Time Complexity: O(V × E)

- **Why**: We perform V-1 iterations, and in each iteration, we check all E edges
- **In simple terms**: If you have V cities and E roads, you need to check all roads V times
- **Worst case**: For a complete graph, E can be V^2, making it $O(V^3)$

Space Complexity: O(V)

- **Why**: We only use extra space for the distance array of size V
- **In simple terms**: We need to store one distance value for each vertex
- **Additional space**: No extra data structures needed beyond the input

Comparison with Dijkstra's Algorithm

- **Dijkstra**: Faster O((V + E) log V) but cannot handle negative edges
- **Bellman-Ford**: Slower O(V × E) but handles negative edges and detects negative cycles
- **Use Bellman-Ford when**: Graph has negative edges or you need to detect negative cycles

Floyd-Warshall Algorithm

Problem Description

Given a weighted directed graph represented as an adjacency matrix, find the shortest distances between every pair of vertices. The matrix[i][j] represents the weight of the edge from vertex i to vertex j. If matrix[i][j] = -1, it means there is no direct edge from i to j. The goal is to find the shortest path between all pairs of vertices.

Examples

Example 1

Input

```
matrix = [[0, 2, -1, -1],
          [1, 0, 3, -1],
          [-1, -1, 0, 1],
          [3, 5, 4, 0]]
```

Output

```
[[0, 2, 5, 6],
 [1, 0, 3, 4],
 [4, 6, 0, 1],
 [3, 5, 4, 0]]
```

Explanation

Let's trace a few paths:

- **From 0 to 2**: Direct path doesn't exist (-1), but 0→1→2 = 2+3 = 5
- **From 0 to 3**: No direct path, but 0→1→2→3 = 2+3+1 = 6
- **From 2 to 0**: No direct path, but 2→3→0 = 1+3 = 4
- **From 1 to 3**: No direct path, but 1→2→3 = 3+1 = 4

Example 2

Input

```
matrix = [[0, 25],
          [-1, 0]]
```

Output

```
[[0, 25],
 [-1, 0]]
```

Explanation

- **From 0 to 1**: Direct path exists with distance 25
- **From 1 to 0**: No path exists, so it remains -1

Solution

The Floyd-Warshall algorithm finds the shortest paths between all pairs of vertices by systematically checking if going through an intermediate vertex k provides a shorter path from vertex i to vertex j than the current known shortest path.

Intuition

Think of this algorithm like finding the best flight connections:

The Problem: You want to know the cheapest way to fly between every pair of cities in the world.

The Approach: For each possible "layover city" k, check if:

- Flying from city i → layover city k → city j is cheaper than
- Flying directly from city i → city j (if direct flight exists)

Why it works:

- We try every possible intermediate city as a "stepping stone"
- We do this systematically for all pairs of cities
- By the end, we've considered all possible routes and found the cheapest ones

Real-world analogy: It's like a travel agent who checks every possible connection through every airport to find you the cheapest route between any two cities.

Approach Steps

1. **Initialize the matrix**: Replace all -1 values (no edge) with infinity to handle calculations properly
2. **Triple nested loop structure**:
    - **Outer loop (k)**: Try each vertex as an intermediate/via point
    - **Middle loop (i)**: Consider each vertex as source
    - **Inner loop (j)**: Consider each vertex as destination
3. **Path comparison**: For each pair (i,j), compare:
    - Current shortest known distance from i to j
    - Distance via intermediate vertex k: distance[i][k] + distance[k][j]
    - Take the minimum of these two
4. **Restore format**: Convert infinity values back to -1 for unreachable vertices

5. **Return result**: The matrix now contains shortest distances between all pairs

Code

```
class Solution:
    def shortest_distance(self, matrix):
        n = len(matrix)

        # Step 1: Replace -1 with infinity for easier calculations
        for i in range(n):
            for j in range(n):
                if matrix[i][j] == -1:
                    matrix[i][j] = float('inf')

        # Step 2: Floyd-Warshall algorithm
        # Try each vertex k as intermediate vertex
        for k in range(n):
            # For each source vertex i
            for i in range(n):
                # For each destination vertex j
                for j in range(n):
                    # Check if path via k is shorter than direct path
                    # matrix[i][j] = min(direct_path, via_k_path)
                    matrix[i][j] = min(matrix[i][j], matrix[i][k] + matrix[k][j])

        # Step 3: Convert infinity back to -1 for unreachable vertices
        for i in range(n):
            for j in range(n):
                if matrix[i][j] == float('inf'):
                    matrix[i][j] = -1

        return matrix
```

Alternative Implementation with Negative Cycle Detection

```
class Solution:
    def shortest_distance_with_cycle_detection(self, matrix):
        n = len(matrix)

        # Initialize
        for i in range(n):
            for j in range(n):
                if matrix[i][j] == -1:
                    matrix[i][j] = float('inf')

        # Floyd-Warshall
        for k in range(n):
            for i in range(n):
                for j in range(n):
                    matrix[i][j] = min(matrix[i][j], matrix[i][k] + matrix[k][j])

        # Check for negative cycles
        for i in range(n):
            if matrix[i][i] < 0:
                return "Negative cycle detected"

        # Restore format
        for i in range(n):
```

```
            for j in range(n):
                if matrix[i][j] == float('inf'):
                    matrix[i][j] = -1

        return matrix
```

Time and Space Complexity

Time Complexity: $O(V^3)$

- **Why**: Three nested loops, each running V times where V is the number of vertices
- **In simple terms**: For each vertex as intermediate point, we check all pairs of vertices
- **Breakdown**:
  - Outer loop (k): V iterations
  - Middle loop (i): V iterations
  - Inner loop (j): V iterations
  - Total: $V \times V \times V = V^3$

Space Complexity: $O(1)$

- **Why**: We modify the input matrix in-place, no additional space needed
- **In simple terms**: We only use the given matrix and a few variables
- **Note**: If we create a separate result matrix, space complexity becomes $O(V^2)$

Find City with Smallest Number of Neighbors

Problem Description

Given n cities numbered from 0 to n-1 and bidirectional weighted edges between them, find the city with the smallest number of cities that are reachable within a given distance threshold. If multiple cities have the same minimum count of reachable neighbors, return the city with the largest index number.

Examples

Example 1

Input

```
N = 4, M = 4
edges = [[0,1,3], [1,2,1], [1,3,4], [2,3,1]]
distanceThreshold = 4
```

Output

```
3
```

Explanation

Let's find reachable cities for each city within distance threshold 4:

City 0: Can reach cities 1 (distance=3), 2 (distance=4) → **2 cities City 1**: Can reach cities 0 (distance=3), 2 (distance=1), 3 (distance=2) → **3 cities**
City 2: Can reach cities 0 (distance=4), 1 (distance=1), 3 (distance=1) → **3 cities City 3**: Can reach cities 1 (distance=2), 2 (distance=1) → **2 cities**

Cities 0 and 3 both have minimum count (2 cities). Since we need the city with the largest index, answer is **City 3**.

Example 2

Input

```
N = 3, M = 2
edges = [[0,1,1], [0,2,3]]
distanceThreshold = 2
```

Output

```
2
```

Explanation

City 0: Can reach city 1 (distance=1) → **1 city City 1**: Can reach city 0 (distance=1) → **1 city City 2**: Cannot reach any city within threshold 2 → **0 cities**

City 2 has the minimum count (0 cities), so answer is **City 2**.

Solution

This problem requires finding shortest distances between all pairs of cities, then counting reachable neighbors for each city. We use Floyd-Warshall algorithm to compute all-pairs shortest paths, then count neighbors within the distance threshold for each city.

Intuition

Think of this problem like finding the most isolated city:

Real-world analogy: Imagine you're a delivery company trying to choose a city for your headquarters. You want to choose a city that has the fewest other cities within your delivery range (distance threshold), because:

- Less competition in nearby areas
- More focused service area
- If there's a tie, choose the city with the higher number (maybe better infrastructure)

Why this approach works:

1. **Step 1**: Calculate shortest distances between ALL city pairs (like having a complete road distance map)

2. **Step 2**: For each city, count how many other cities are within delivery range
3. **Step 3**: Pick the city with minimum neighbors (most isolated)
4. **Step 4**: If tie, pick the city with largest index

Why Floyd-Warshall: We need distances between ALL pairs of cities, making Floyd-Warshall perfect for this multi-source shortest path problem.

Approach Steps

1. **Create distance matrix**: Initialize a 2D matrix where matrix[i][j] represents shortest distance from city i to city j
    - Set direct edge weights from input
    - Set distance from city to itself as 0
    - Set all other distances to infinity
2. **Apply Floyd-Warshall algorithm**: Find shortest paths between all pairs of cities using intermediate cities
3. **Count reachable neighbors**: For each city, count how many other cities are reachable within the distance threshold
4. **Find optimal city**:
    - Find the city with minimum neighbor count
    - If there's a tie, choose the city with the largest index
    - Return that city

Code

```
class Solution:
    def findCity(self, n, m, edges, distanceThreshold):
        # Step 1: Initialize distance matrix
        matrix = [[float('inf')] * n for _ in range(n)]

        # Set direct edges (bidirectional)
        for source, dest, weight in edges:
            matrix[source][dest] = weight
            matrix[dest][source] = weight  # bidirectional

        # Distance from city to itself is 0
        for i in range(n):
            matrix[i][i] = 0

        # Step 2: Apply Floyd-Warshall algorithm
        # Try each city k as intermediate city
        for k in range(n):
            for i in range(n):
                for j in range(n):
                    # Check if path via city k is shorter
                    matrix[i][j] = min(matrix[i][j], matrix[i][k] + matrix[k][j])

        # Step 3: Count reachable neighbors for each city
        result_city = 0
        min_neighbors = float('inf')

        for i in range(n):
```

```
            neighbor_count = 0

            # Count cities reachable within threshold
            for j in range(n):
                if i != j and matrix[i][j] <= distanceThreshold:
                    neighbor_count += 1

            # Step 4: Update result (choose city with min neighbors, largest index in tie)
            if neighbor_count <= min_neighbors:
                min_neighbors = neighbor_count
                result_city = i  # This automatically handles "largest index" due to iteration order

        return result_city
```

Step-by-Step Trace for Example 1

```
# Initial matrix after setting edges:
matrix = [[0, 3, inf, inf],
          [3, 0, 1, 4],
          [inf, 1, 0, 1],
          [inf, 4, 1, 0]]

# After Floyd-Warshall:
matrix = [[0, 3, 4, 4],    # City 0 can reach: 1(3), 2(4), 3(4)
          [3, 0, 1, 2],    # City 1 can reach: 0(3), 2(1), 3(2)
          [4, 1, 0, 1],    # City 2 can reach: 0(4), 1(1), 3(1)
          [4, 2, 1, 0]]    # City 3 can reach: 0(4), 1(2), 2(1)

# Neighbor counts (within threshold 4):
# City 0: neighbors [1,2] = 2 cities
# City 1: neighbors [0,2,3] = 3 cities
# City 2: neighbors [0,1,3] = 3 cities
# City 3: neighbors [1,2] = 2 cities

# Result: Cities 0 and 3 tie with 2 neighbors, choose larger index = 3
```

Time and Space Complexity

Time Complexity: $O(N^3)$

- **Floyd-Warshall algorithm**: $O(N^3)$ with three nested loops
- **Counting neighbors**: $O(N^2)$ to check all pairs
- **Overall**: $O(N^3 + N^2) = O(N^3)$
- **In simple terms**: For each city as intermediate point, we update distances between all city pairs

Space Complexity: $O(N^2)$

- **Distance matrix**: $O(N^2)$ space to store shortest distances between all pairs
- **Additional variables**: $O(1)$ space
- **Overall**: $O(N^2)$
- **In simple terms**: We need a 2D array to store distances between every pair of cities

Why This Approach is Optimal

1. **All-pairs shortest path needed**: We must know distances between ALL city pairs, making Floyd-Warshall the right choice
2. **Handles complex graphs**: Works with any graph structure, finding optimal paths through multiple intermediate cities
3. **Single pass counting**: After computing distances, we can count neighbors in $O(N^2)$ time

Alternative Approaches (Less Efficient)

1. **Dijkstra from each city**: $O(N \times (N + M) \log N)$ - slower for dense graphs
2. **Bellman-Ford from each city**: $O(N^2 \times M)$ - much slower
3. **DFS/BFS with distance limit**: $O(N \times (N + M))$ - but may not find optimal paths

Key Insights

- **Bidirectional edges**: Remember to set both matrix[i][j] and matrix[j][i]
- **Tie-breaking rule**: When counts are equal, choose larger index (handled naturally by iteration order)
- **Self-distance**: Always set matrix[i][i] = 0
- **Threshold comparison**: Use ≤ (less than or equal) when counting neighbors

A* Search Algorithm

Problem Description

Given a grid with obstacles and a start and end position, find the shortest path from start to end using the A* search algorithm. The grid contains 0s (free cells) and 1s (obstacles). You can move in 4 directions (up, down, left, right) and each move has a cost of 1. The A* algorithm uses both the actual distance traveled (g-cost) and a heuristic estimate to the goal (h-cost) to efficiently find the optimal path.

Examples

Example 1

Input

```
grid = [[0, 0, 0, 0, 0],
        [0, 1, 1, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0],
        [0, 0, 0, 0, 0]]
start = (0, 0)
end = (4, 4)
```

Output

```
Path: [(0,0), (0,1), (0,2), (0,3), (0,4), (1,4), (2,4), (3,4), (4,4)]
Total Cost: 8
```

Explanation

The algorithm finds the shortest path by avoiding obstacles (1s) and using the Manhattan distance heuristic to guide the search efficiently. The path goes around the obstacles in the optimal way.

Example 2

Input

```
grid = [[0, 0, 0],
        [0, 1, 0],
        [0, 0, 0]]
start = (0, 0)
end = (2, 2)
```

Output

```
Path: [(0,0), (0,1), (0,2), (1,2), (2,2)]
Total Cost: 4
```

Explanation

The algorithm navigates around the single obstacle at (1,1) to reach the destination with minimum cost.

Solution

A* is an informed search algorithm that combines the benefits of Dijkstra's algorithm (guaranteed shortest path) with the efficiency of greedy best-first search (using heuristics). It maintains a priority queue of nodes to explore, ordered by f-cost = g-cost + h-cost, where g-cost is the actual distance from start and h-cost is the heuristic estimate to the goal.

Intuition

Think of A* like a smart GPS navigation system:

Real-world analogy: Imagine you're driving to a destination and your GPS has to choose between multiple routes:

- **G-cost (actual distance)**: How far you've already driven from your starting point
- **H-cost (heuristic)**: Straight-line distance to your destination (like "as the crow flies")
- **F-cost (total estimate)**: G-cost + H-cost = total estimated trip distance

Why this works:

1. **GPS logic**: The GPS always picks the route with the lowest total estimated distance (F-cost)
2. **Smart exploration**: Instead of exploring all possible routes equally, it prioritizes routes that seem most promising
3. **Guaranteed optimal**: If the heuristic never overestimates (admissible), you'll always find the shortest route

Key insight: A* is like having perfect intuition about which direction to explore first, making it much faster than blind search algorithms while still guaranteeing the optimal solution.

Approach Steps

1. **Initialize data structures**:
    - Priority queue (min-heap) for nodes to explore, ordered by f-cost
    - Sets to track visited nodes and nodes in queue
    - Dictionary to store g-costs and parent pointers for path reconstruction
2. **Set up starting conditions**:
    - Add start node to queue with g-cost = 0, h-cost = heuristic(start, end)
    - Mark start as discovered
3. **Main search loop**:
    - Extract node with lowest f-cost from queue
    - If it's the goal, reconstruct and return path
    - Mark current node as visited
    - Explore all valid neighbors
4. **Neighbor processing**:
    - Skip obstacles and already visited nodes
    - Calculate tentative g-cost for neighbor
    - If this path to neighbor is better than any previous path:
        - Update g-cost and parent
        - Add to queue if not already there
5. **Path reconstruction**:
    - Backtrack from goal to start using parent pointers
    - Return the path and total cost

Code

```
import heapq
from typing import List, Tuple, Optional

class AStar:
    def __init__(self, grid: List[List[int]]):
        self.grid = grid
        self.rows = len(grid)
        self.cols = len(grid[0])
        # 4-directional movement: up, down, left, right
        self.directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    def heuristic(self, pos1: Tuple[int, int], pos2: Tuple[int, int]) -> int:
        """Manhattan distance heuristic (admissible for 4-directional movement)"""
        return abs(pos1[0] - pos2[0]) + abs(pos1[1] - pos2[1])

    def is_valid(self, row: int, col: int) -> bool:
        """Check if position is within grid bounds and not an obstacle"""
        return (0 <= row < self.rows and
                0 <= col < self.cols and
```

```
                self.grid[row][col] == 0)

    def reconstruct_path(self, parent: dict, start: Tuple[int, int],
                         end: Tuple[int, int]) -> List[Tuple[int, int]]:
        """Backtrack from goal to start to build the path"""
        path = []
        current = end

        while current is not None:
            path.append(current)
            current = parent.get(current)

        path.reverse()
        return path

    def search(self, start: Tuple[int, int], end: Tuple[int, int]) -> Optional[Tuple[List[Tuple[int,
int]], int]]:
        """
        A* search algorithm implementation
        Returns: (path, cost) if path found, None otherwise
        """
        # Priority queue: (f_cost, g_cost, position)
        queue = [(0, 0, start)]

        # Track nodes we've seen and visited
        in_queue = {start}
        visited = set()

        # Cost from start to each node
        g_cost = {start: 0}

        # Parent pointers for path reconstruction
        parent = {start: None}

        while queue:
            # Get node with lowest f-cost
            current_f, current_g, current_pos = heapq.heappop(queue)

            # Skip if we've already processed this node
            if current_pos in visited:
                continue

            # Mark as visited
            visited.add(current_pos)
            in_queue.discard(current_pos)

            # Check if we reached the goal
            if current_pos == end:
                path = self.reconstruct_path(parent, start, end)
                return path, current_g

            # Explore neighbors
            for dr, dc in self.directions:
                neighbor_row = current_pos[0] + dr
                neighbor_col = current_pos[1] + dc
                neighbor_pos = (neighbor_row, neighbor_col)

                # Skip invalid positions or already visited nodes
                if not self.is_valid(neighbor_row, neighbor_col) or neighbor_pos in visited:
                    continue

                # Calculate tentative g-cost
```

```
                tentative_g = current_g + 1  # Each step costs 1

                # If we found a better path to this neighbor
                if neighbor_pos not in g_cost or tentative_g < g_cost[neighbor_pos]:
                    # Update costs and parent
                    g_cost[neighbor_pos] = tentative_g
                    parent[neighbor_pos] = current_pos

                    # Calculate f-cost and add to queue if not already there
                    if neighbor_pos not in in_queue:
                        h_cost = self.heuristic(neighbor_pos, end)
                        f_cost = tentative_g + h_cost
                        heapq.heappush(queue, (f_cost, tentative_g, neighbor_pos))
                        in_queue.add(neighbor_pos)

        # No path found
        return None

# Usage example
def solve_pathfinding(grid: List[List[int]], start: Tuple[int, int],
                      end: Tuple[int, int]) -> Optional[Tuple[List[Tuple[int, int]], int]]:
    """
    Solve pathfinding using A* algorithm

    Args:
        grid: 2D list where 0 = free space, 1 = obstacle
        start: Starting position (row, col)
        end: Goal position (row, col)

    Returns:
        Tuple of (path, cost) if path exists, None otherwise
    """
    astar = AStar(grid)
    return astar.search(start, end)

# Example usage
if __name__ == "__main__":
    grid = [[0, 0, 0, 0, 0],
            [0, 1, 1, 0, 0],
            [0, 0, 0, 0, 0],
            [0, 1, 1, 1, 0],
            [0, 0, 0, 0, 0]]

    start = (0, 0)
    end = (4, 4)

    result = solve_pathfinding(grid, start, end)
    if result:
        path, cost = result
        print(f"Path found: {path}")
        print(f"Total cost: {cost}")
    else:
        print("No path found")
```

Time and Space Complexity

Time Complexity: O(b^d)

- **Where**: b = branching factor (average neighbors per node), d = depth of optimal solution

- **In practice**: Much better than O(b^d) due to heuristic guidance
- **Grid case**: O(V log V) where V is number of free cells, similar to Dijkstra but with better practical performance
- **Why**: Each node is processed at most once, and heap operations take O(log V) time

Space Complexity: O(b^d)

- **Priority queue**: Can store up to O(b^d) nodes in worst case
- **Data structures**: O(V) for visited set, g_cost dictionary, and parent dictionary
- **Overall**: O(V) in practice for grid problems
- **Why**: We need to store information for all nodes we've discovered

Johnson's Algorithm

Problem Description

Given a weighted directed graph that may contain negative edge weights, find the shortest paths between all pairs of vertices. Johnson's Algorithm is designed to handle graphs with negative edges efficiently by combining the Bellman-Ford and Dijkstra algorithms. It first uses Bellman-Ford to detect negative cycles and reweight the graph, then applies Dijkstra from each vertex on the reweighted graph to find all-pairs shortest paths.

Examples

Example 1

Input

```
V = 4
edges = [[0, 1, -5], [0, 2, 2], [0, 3, 3], [1, 2, 4], [2, 3, 1]]
```

Output

```
Shortest distances between all pairs:
From 0: [0, -5, -1, 0]
From 1: [inf, 0, 4, 5]
From 2: [inf, inf, 0, 1]
From 3: [inf, inf, inf, 0]
```

Explanation

- **Step 1**: Add auxiliary vertex connected to all vertices with weight 0
- **Step 2**: Run Bellman-Ford from auxiliary vertex to get reweighting values
- **Step 3**: Reweight all edges using: new_weight = original_weight + h[u] - h[v]
- **Step 4**: Run Dijkstra from each vertex on reweighted graph
- **Step 5**: Convert back to original weights

Example 2 (Negative Cycle)

Input

```
V = 3
edges = [[0, 1, 1], [1, 2, -3], [2, 0, 1]]
```

Output

```
"Negative cycle detected"
```

Explanation

The cycle 0→1→2→0 has total weight: 1 + (-3) + 1 = -1, which is negative. Johnson's algorithm detects this during the Bellman-Ford phase and reports the negative cycle.

Solution

Johnson's Algorithm cleverly transforms a graph with negative edges into one with non-negative edges while preserving shortest path relationships. It then uses the faster Dijkstra algorithm multiple times instead of the slower Floyd-Warshall algorithm, making it more efficient for sparse graphs.

Intuition

Think of Johnson's Algorithm like adjusting prices in different cities to make them all positive:

Real-world analogy: Imagine you're a logistics company with shipping costs between cities, some of which give you money back (negative costs):

The Problem: You want to use fast route-finding algorithms, but they don't work with "cashback routes"

Johnson's Solution:

1. **Add a "virtual depot"**: Create an imaginary city connected to all real cities with 0 cost
2. **Calculate adjustment values**: Find the cheapest way to reach each city from this virtual depot
3. **Adjust all shipping costs**: Add/subtract adjustment values so all costs become positive
4. **Use fast algorithms**: Now you can use efficient algorithms on the adjusted costs
5. **Convert back**: Transform the results back to original cost scale

Why it works:

- The adjustment preserves the relative ordering of paths
- If path A was cheaper than path B originally, it stays cheaper after adjustment
- All edges become non-negative, enabling Dijkstra's algorithm

Key insight: It's like converting different currencies to a common positive currency, doing calculations, then converting back!

Approach Steps

1. **Add auxiliary vertex**: Create a new vertex and connect it to all original vertices with weight 0
2. **Run Bellman-Ford**:
    - Execute Bellman-Ford from the auxiliary vertex
    - If negative cycle detected, return "Negative cycle exists"
    - Otherwise, get shortest distances h[v] to all vertices
3. **Reweight all edges**: Transform each edge (u,v,w) to new weight: w' = w + h[u] - h[v]
4. **Run Dijkstra from each vertex**:
    - Apply Dijkstra algorithm from each vertex on the reweighted graph
    - This gives shortest distances in the transformed graph
5. **Convert back to original weights**: Transform distances back using: original_dist[u][v] = reweighted_dist[u][v] - h[u] + h[v]
6. **Return result**: Return the all-pairs shortest distance matrix

Code

```
import heapq
from typing import List, Optional, Dict, Tuple
from collections import defaultdict

class JohnsonsAlgorithm:
    def __init__(self):
        self.graph = defaultdict(list)
        self.V = 0

    def add_edge(self, u: int, v: int, weight: int):
        """Add a directed edge to the graph"""
        self.graph[u].append((v, weight))

    def bellman_ford(self, src: int, V: int) -> Optional[List[float]]:
        """
        Bellman-Ford algorithm to detect negative cycles and find shortest distances
        Returns None if negative cycle exists, otherwise returns distance array
        """
        # Initialize distances
        dist = [float('inf')] * V
        dist[src] = 0

        # Relax edges V-1 times
        for _ in range(V - 1):
            for u in self.graph:
                if dist[u] != float('inf'):
                    for v, weight in self.graph[u]:
                        if dist[u] + weight < dist[v]:
                            dist[v] = dist[u] + weight

        # Check for negative cycles
        for u in self.graph:
            if dist[u] != float('inf'):
```

```
            for v, weight in self.graph[u]:
                if dist[u] + weight < dist[v]:
                    return None  # Negative cycle detected

    return dist

def dijkstra(self, src: int, V: int) -> List[float]:
    """
    Dijkstra's algorithm for non-negative weighted graph
    """
    dist = [float('inf')] * V
    dist[src] = 0
    pq = [(0, src)]  # (distance, vertex)
    visited = set()

    while pq:
        d, u = heapq.heappop(pq)

        if u in visited:
            continue

        visited.add(u)

        for v, weight in self.graph[u]:
            if dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight
                heapq.heappush(pq, (dist[v], v))

    return dist

def johnsons_algorithm(self, edges: List[List[int]], V: int) -> Optional[List[List[float]]]:
    """
    Johnson's Algorithm for All-Pairs Shortest Path

    Args:
        edges: List of [source, destination, weight]
        V: Number of vertices (0 to V-1)

    Returns:
        2D list of shortest distances, or None if negative cycle exists
    """
    # Clear previous graph
    self.graph.clear()

    # Step 1: Add all original edges to graph
    for u, v, w in edges:
        self.add_edge(u, v, w)

    # Step 2: Add auxiliary vertex (vertex V) connected to all vertices with weight 0
    auxiliary_vertex = V
    for i in range(V):
        self.add_edge(auxiliary_vertex, i, 0)

    # Step 3: Run Bellman-Ford from auxiliary vertex
    h = self.bellman_ford(auxiliary_vertex, V + 1)

    if h is None:
        return None  # Negative cycle detected

    # Step 4: Remove auxiliary vertex and reweight edges
    self.graph.clear()
```

```
        # Add reweighted edges: new_weight = original_weight + h[u] - h[v]
        for u, v, w in edges:
            new_weight = w + h[u] - h[v]
            self.add_edge(u, v, new_weight)

        # Step 5: Run Dijkstra from each vertex
        result = []

        for src in range(V):
            # Get shortest distances from src in reweighted graph
            reweighted_dist = self.dijkstra(src, V)

            # Convert back to original weights: original = reweighted - h[src] + h[dest]
            original_dist = []
            for dest in range(V):
                if reweighted_dist[dest] == float('inf'):
                    original_dist.append(float('inf'))
                else:
                    # Convert back: dist[src][dest] = reweighted_dist[dest] - h[src] + h[dest]
                    original_dist.append(reweighted_dist[dest] - h[src] + h[dest])

            result.append(original_dist)

        return result

# Usage function
def solve_all_pairs_shortest_path(edges: List[List[int]], V: int) -> Optional[List[List[float]]]:
    """
    Solve All-Pairs Shortest Path using Johnson's Algorithm

    Args:
        edges: List of [source, destination, weight] representing directed edges
        V: Number of vertices (numbered 0 to V-1)

    Returns:
        2D matrix where result[i][j] is shortest distance from vertex i to vertex j,
        or None if negative cycle exists
    """
    johnson = JohnsonsAlgorithm()
    return johnson.johnsons_algorithm(edges, V)

# Example usage and testing
if __name__ == "__main__":
    # Example 1: Graph with negative edges but no negative cycle
    print("Example 1:")
    edges1 = [[0, 1, -5], [0, 2, 2], [0, 3, 3], [1, 2, 4], [2, 3, 1]]
    V1 = 4

    result1 = solve_all_pairs_shortest_path(edges1, V1)

    if result1:
        print("Shortest distances between all pairs:")
        for i in range(V1):
            print(f"From {i}: {[float('inf') if x == float('inf') else int(x) for x in result1[i]]}")
    else:
        print("Negative cycle detected")

    print("\nExample 2:")
    # Example 2: Graph with negative cycle
    edges2 = [[0, 1, 1], [1, 2, -3], [2, 0, 1]]
    V2 = 3
```

```
    result2 = solve_all_pairs_shortest_path(edges2, V2)

    if result2:
        print("Shortest distances between all pairs:")
        for i in range(V2):
            print(f"From {i}: {result2[i]}")
    else:
        print("Negative cycle detected")

# Utility function to print results nicely
def print_distance_matrix(matrix: List[List[float]], V: int):
    """Pretty print the distance matrix"""
    print("\nDistance Matrix:")
    print("    ", end="")
    for j in range(V):
        print(f"{j:6}", end="")
    print()

    for i in range(V):
        print(f"{i}: ", end="")
        for j in range(V):
            if matrix[i][j] == float('inf'):
                print("  INF", end="")
            else:
                print(f"{matrix[i][j]:6.0f}", end="")
        print()
```

Time and Space Complexity

Time Complexity: $O(V^2 \log V + VE)$

- **Bellman-Ford phase**: O(VE) for detecting negative cycles
- **Dijkstra phase**: $O(V \times (V \log V + E)) = O(V^2 \log V + VE)$ for V runs of Dijkstra
- **Overall**: $O(VE + V^2 \log V + VE) = O(V^2 \log V + VE)$
- **For dense graphs**: $O(V^3 \log V)$ when $E \approx V^2$
- **For sparse graphs**: $O(V^2 \log V)$ when $E \approx V$

Space Complexity: $O(V^2 + E)$

- **Distance matrices**: $O(V^2)$ for storing all-pairs shortest distances
- **Graph representation**: O(V + E) for adjacency list
- **Dijkstra's heap**: O(V) space per run
- **Reweighting arrays**: O(V) for h values
- **Overall**: $O(V^2 + E)$

Spanning Tree and Minimum Spanning Tree Explained

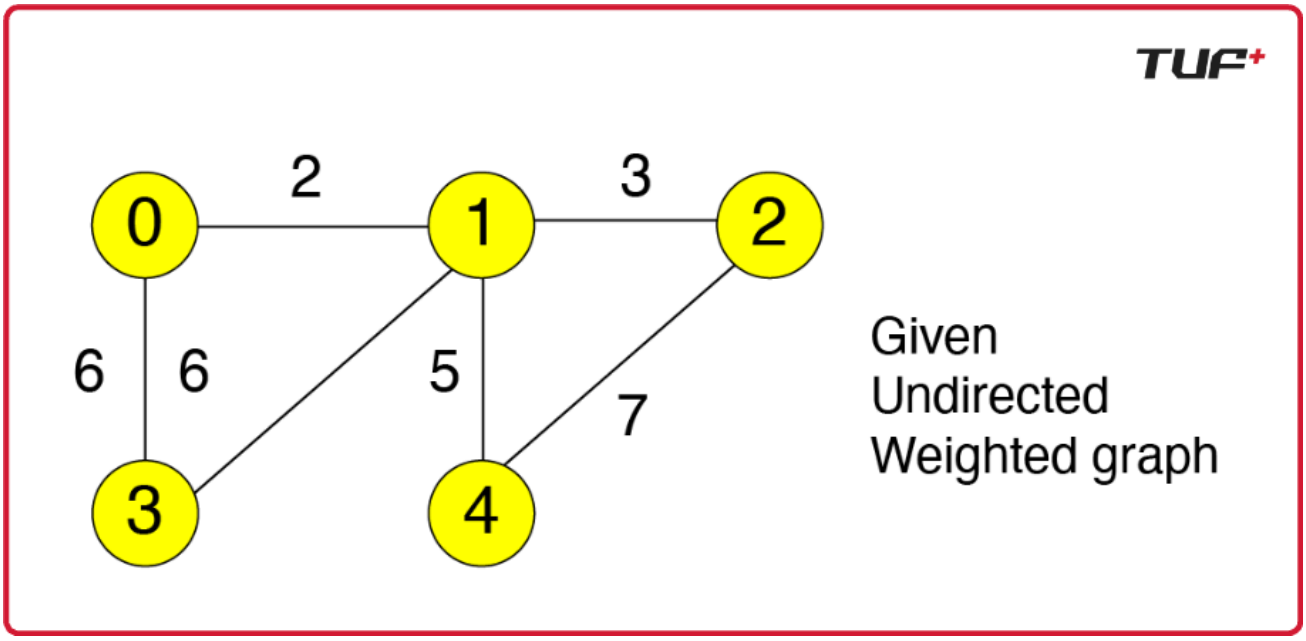
1. Spanning Tree

A **spanning tree** is a special type of subgraph of a weighted or unweighted graph. It has the following characteristics:

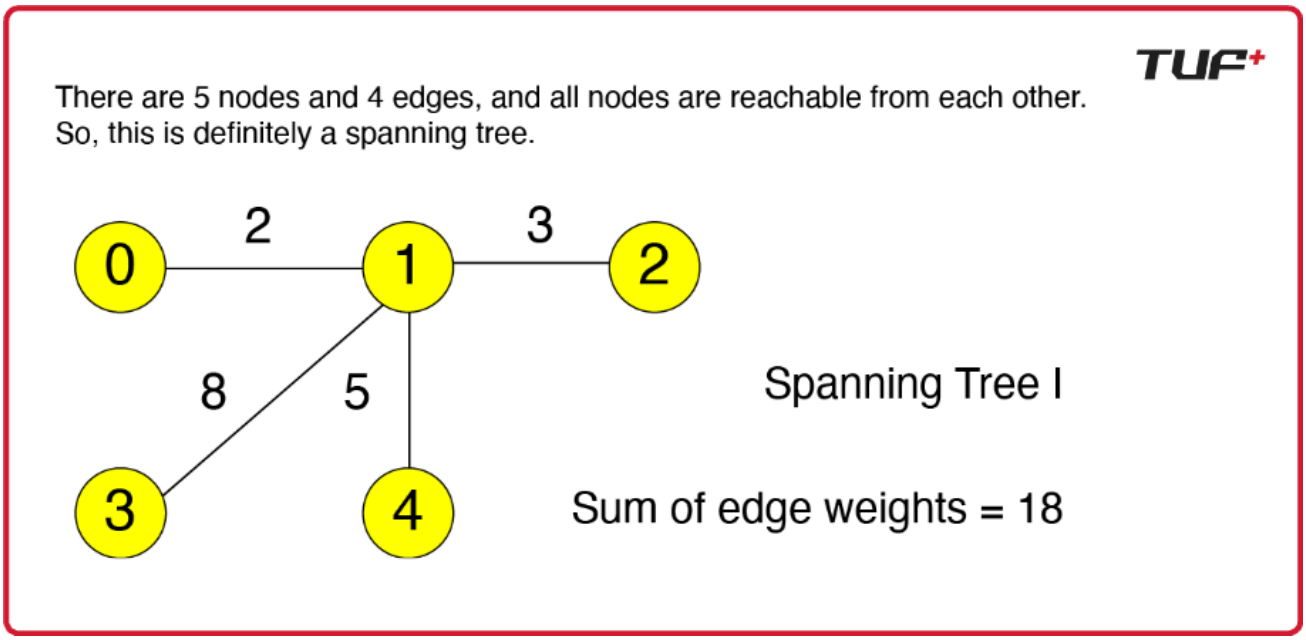
- **Contains all N nodes** from the original graph.
- Has exactly **N-1 edges**.
- All nodes are **reachable** from each other (graph is connected).
- It is a **tree** (meaning it contains no cycles).

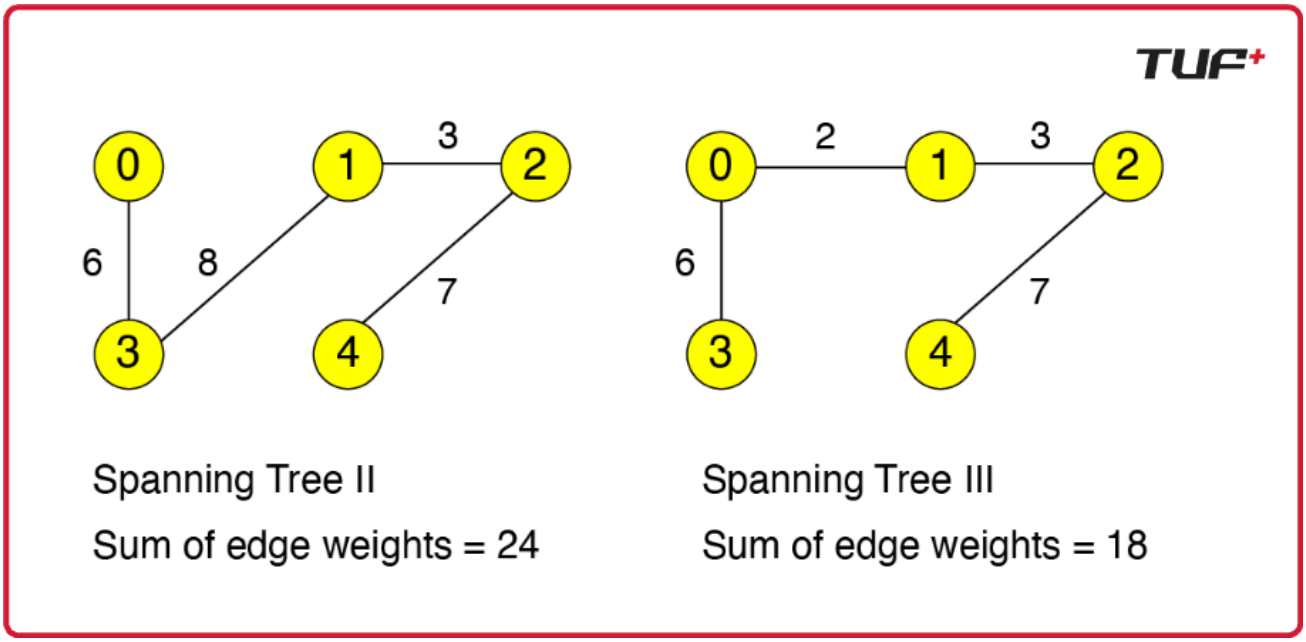
Example

- Suppose we have an **undirected weighted graph** with:
    - `N = 6` (nodes)
    - `M = 7` (edges)



* A spanning tree will be a subset of this graph with **6 nodes** and **5 edges** connecting all nodes without any cycles.





For example:

- **Spanning Tree 1**: Weight = 18
- **Spanning Tree 2**: Weight = 24
- **Spanning Tree 3**: Weight = 18

> **Note:** A graph can have **multiple spanning trees**.

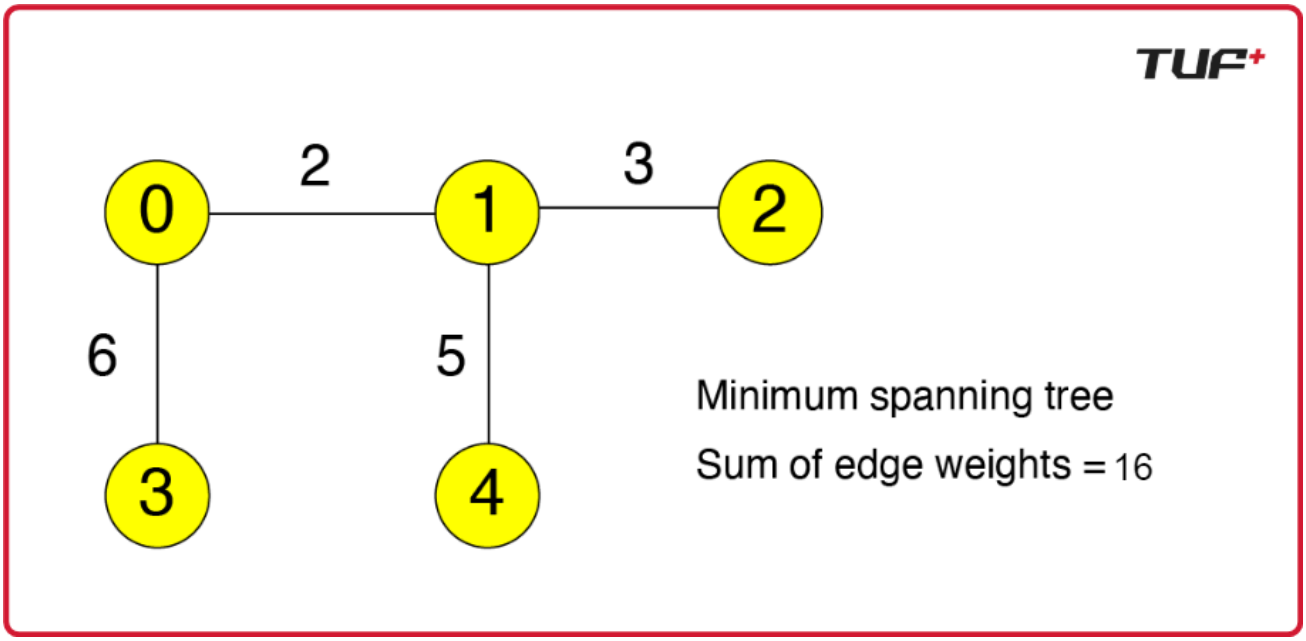
Weight of a Spanning Tree

The **weight** of a spanning tree is the **sum of the weights of all its edges**.

2. Minimum Spanning Tree (MST)

A **minimum spanning tree** is the spanning tree with the **smallest possible weight** among all spanning trees of the graph.

- If we list all spanning trees of a graph and calculate their weights, the MST will be the one with the **minimum total edge weight**.
- In the above example:
  - The MST has a total weight of **16** (smaller than 18 or 24).
  - A graph can have **more than one MST** if there are ties in weight.



3. Algorithms to Find MST

Two main algorithms are used to find the MST of a graph:

1. **Prim's Algorithm**
   - Starts with a single node and keeps adding the smallest possible edge that connects a new node.
2. **Kruskal's Algorithm**
   - Sorts all edges by weight and adds them one by one, ensuring no cycles are formed.

4. Real-Life Use Cases of MST

The MST concept is widely used in real-world problems where we need to **connect multiple points with the minimum cost**.

Examples:

1. **Telecommunications Networks**

    - Laying out cables to connect all phone exchanges with **minimum cable length**.
2. **Computer Networks**
    - Designing Local Area Networks (LAN) or Wide Area Networks (WAN) using the **least number of cables and networking hardware**.
3. **Transportation Networks**
    - Building road or railway systems connecting all cities without **redundant routes**.
4. **Water Supply Networks**
    - Creating the most cost-efficient pipeline network that connects all regions.

In short:

- **Spanning Tree:** All nodes connected, N-1 edges, no cycles.
- **Minimum Spanning Tree:** The spanning tree with the smallest total weight.
- **Algorithms:** Prim's, Kruskal's.
- **Applications:** Networks, transportation, utilities.

Prim's Algorithm Explained

1. Introduction

Prim's Algorithm is a greedy algorithm used to find the **Minimum Spanning Tree (MST)** of a **connected, weighted, undirected graph**.

The idea is to **start from a single node** and keep adding the **smallest edge** that connects a new node to the growing tree until all nodes are included.

2. Key Concepts

- **MST Goal:** Connect all nodes with minimum total edge weight.
- **Greedy Choice:** Always choose the smallest weight edge that adds a new vertex to the tree.
- **No Cycles:** The selected edges must not form cycles.

3. Steps of Prim's Algorithm

1. **Start** with any node (let's call it the starting vertex).
2. **Mark** this node as part of the MST.
3. From all edges that connect the MST to nodes not yet in the MST, **choose the edge with the smallest weight**.
4. **Add** the selected edge and the new node to the MST.
5. **Repeat** steps 3 and 4 until all nodes are included.

4. Example

Let's say we have 5 nodes (A, B, C, D, E) with weighted edges:

1. Start from A.
2. Find the smallest edge from A to another node → (A, B) with weight 2.
3. Now MST has A and B. Look for smallest edge connecting MST to new node → (B, C) with weight 3.
4. Repeat until all nodes are connected.

5. Python Implementation

```
import heapq

def prim_mst(nodes,graph):
    MST=[]
    MSTWeight=0
    visited={node:False for node in nodes}
    queue=[]
    src=nodes[0]
    visited[src]=True
    edges=[(adj_weight,src,adj_node) for adj_node,adj_weight in graph[src]]
    heapq.heapify(edges)
    while edges:
        weight,from_node,to_node=heapq.heappop(edges)
        if(not visited[to_node]):
            visited[to_node]=True
            MST.append((from_node,to_node,weight))
            MSTWeight+=weight
            for adj_node,adj_weight in graph[to_node]:
                if(not visited[adj_node]):
                    heapq.heappush(edges,(adj_weight,to_node,adj_node))
    return MST,MSTWeight

nodes=['A','B','C','D','E']
graph = {
    'A': [('B', 2), ('C', 3)],
    'B': [('A', 2), ('C', 1), ('D', 4)],
    'C': [('A', 3), ('B', 1), ('D', 5), ('E', 6)],
    'D': [('B', 4), ('C', 5), ('E', 7)],
    'E': [('C', 6), ('D', 7)]
}

mst, cost = prim_mst(nodes,graph)
print("MST Edges:", mst)
print("Total Cost:", cost)
```

Output:

```
MST Edges: [('A', 'B', 2), ('B', 'C', 1), ('B', 'D', 4), ('C', 'E', 6)]
Total Cost: 13
```

6. Time Complexity

- Using a **min-heap/priority queue**: **O(E log V)**
- Without priority queue (simple array): **$O(V^2)$**

Where:

- **V** = number of vertices
- **E** = number of edges

Disjoint Sets (Union-Find) - Complete Guide

📋 Table of Contents

What are Disjoint Sets?

Disjoint sets are a data structure that keeps track of a set of elements partitioned into disjoint (non-overlapping) subsets. They are different from mathematical sets as they are optimized for algorithmic purposes.

Key Properties

- **Disjoint**: No element is common between any two sets
- **Partition**: Every element belongs to exactly one set
- **Dynamic**: Sets can be merged using Union operation

Example Visualization

```
Graph with two components:
Component 1: {1, 2, 3, 4}
Component 2: {5, 6, 7, 8}

S1 ∩ S2 = ∅ (empty set - they are disjoint)
```

Universal Set Representation:

```
Initial State - Each vertex is its own disjoint set:

Universal Set = {1, 2, 3, 4, 5, 6, 7, 8}

Individual Sets:
{1}  {2}  {3}  {4}  {5}  {6}  {7}  {8}

Graph visualization:
1     2     3     4     5     6     7     8
o     o     o     o     o     o     o     o

(All vertices are disconnected initially)
```

Basic Operations

1. Find Operation

Purpose: Determine which set an element belongs to

```
find(5) → Set 2
find(3) → Set 1
find(7) → Set 2
```

2. Union Operation

Purpose: Merge two sets containing different elements

Trigger: When adding an edge (u,v) in a graph:

1. Find set of u
2. Find set of v
3. If different sets → Union them
4. If same set → Cycle detected!

```
Before: S1 = {1,2,3,4}, S2 = {5,6,7,8}
Add edge 4-8
After: S3 = {1,2,3,4,5,6,7,8}
```

Cycle Detection in Graphs

Disjoint sets are particularly useful for detecting cycles in undirected graphs (used in Kruskal's algorithm).

Algorithm Steps

1. Initialize each vertex as its own set
2. For each edge (u,v):
    - Find set of u
    - Find set of v
    - If same set → **Cycle detected!**
    - If different sets → Union them

Problem Statement

Objective: Given an undirected graph, use Disjoint Sets (Union-Find) to detect cycles and find the Minimum Spanning Tree (MST).

Problem: Process edges in order of their weights and determine:

1. Which edges can be safely added without creating cycles
2. Which edges would create cycles and should be rejected

3. Build the MST using Kruskal's algorithm approach

Complete Graph Specification

Graph Details:

- **Vertices**: 8 vertices {1, 2, 3, 4, 5, 6, 7, 8}
- **Total Edges**: 9 edges (more than minimum needed for spanning tree)
- **Expected MST Edges**: 7 edges (for 8 vertices, MST needs exactly 7 edges)

Complete Graph Structure:

```
Original Graph with ALL possible edges:

    1 ——— 2 ——— 5
    |   \   |   /   |
    |    \  |  /    |
    3 ——— 4       6
            |       |
            |       |
            8 ——— 7

All Edges in the Graph:
(1,2), (1,3), (2,4), (2,5), (3,4), (4,8), (5,6), (6,8), (7,8)

Note: Some edges like (1,3) and (5,7) will create cycles
```

Detailed Edge List with Weights:

Edge	Vertices	Weight	Description
E1	(1,2)	1	Connect vertices 1 and 2
E2	(3,4)	2	Connect vertices 3 and 4
E3	(5,6)	3	Connect vertices 5 and 6
E4	(7,8)	4	Connect vertices 7 and 8
E5	(2,4)	5	Connect vertices 2 and 4
E6	(2,5)	6	Connect vertices 2 and 5
E7	(1,3)	7	Connect vertices 1 and 3 ⚠
E8	(6,8)	8	Connect vertices 6 and 8
E9	(5,7)	9	Connect vertices 5 and 7 ⚠

⚠ = These edges will create cycles when processed

Initial State - Universal Set:

```
Universal Set: {1, 2, 3, 4, 5, 6, 7, 8}
Each element is its own set

Graph Representation (No edges included yet):
```

```
1     2     3     4     5     6     7     8
o     o     o     o     o     o     o     o

Disjoint Sets:
{1}   {2}   {3}   {4}   {5}   {6}   {7}   {8}
```

Step-by-step Processing:

Step	Edge	Weight	Find(u)	Find(v)	Same Set?	Action	Result Sets	Graph State
1	(1,2)	1	Set{1}	Set{2}	❌ No	✅ Union	S1={1,2}	1–2 3 4 5 6 7 8
2	(3,4)	2	Set{3}	Set{4}	❌ No	✅ Union	S1={1,2}, S2={3,4}	1–2 3–4 5 6 7 8
3	(5,6)	3	Set{5}	Set{6}	❌ No	✅ Union	S1={1,2}, S2={3,4}, S3={5,6}	1–2 3–4 5–6 7 8
4	(7,8)	4	Set{7}	Set{8}	❌ No	✅ Union	S1={1,2}, S2={3,4}, S3={5,6}, S4={7,8}	1–2 3–4 5–6 7–8
5	(2,4)	5	S1	S2	❌ No	✅ Union	S5={1,2,3,4}, S3={5,6}, S4={7,8}	1–2–4–3 5–6 7–8
6	(2,5)	6	S5	S3	❌ No	✅ Union	S6={1,2,3,4,5,6}, S4={7,8}	1–2–5–6 7–8 | | 4-3
7	(1,3)	7	S6	S6	✅ YES	❌ CYCLE!	No change	Same as step 6
8	(6,8)	8	S6	S4	❌ No	✅ Union	S7={1,2,3,4,5,6,7,8}	All vertices connected
9	(5,7)	9	S7	S7	✅ YES	❌ CYCLE!	No change	Same as step 8

Visual Progress of MST Construction:

```
Step 0: Initial State
1     2     3     4     5     6     7     8
o     o     o     o     o     o     o     o

Step 1: Add Edge (1,2)
1———2     3     4     5     6     7     8
o     o     o     o     o     o     o     o

Step 2: Add Edge (3,4)
1———2     3———4     5     6     7     8
o     o     o     o     o     o     o     o

Step 3: Add Edge (5,6)
1———2     3———4     5———6     7     8
o     o     o     o     o     o     o     o

Step 4: Add Edge (7,8)
1———2     3———4     5———6     7———8
o     o     o     o     o     o     o     o

Step 5: Add Edge (2,4)
1———2     3———4     5———6     7———8
o     o     o     o     o     o     o     o
|     \   /
└———————4

Step 6: Add Edge (2,5)
1———2———5———6     7———8
o     o     o     o     o     o     o     o
|     |
└———4———3

Step 7: Try Edge (1,3)
1———2———5———6     7———8
o     o     o     o     o     o     o     o
|     |
└———4———3
Would create cycle 1→2→4→3→1!

Step 8: Add Edge (6,8)
1———2———5———6———8
o     o     o     o     o     o     o     o
|     |           |     |
└———4———3     └———7
```

```
                                    ❌ REJECT THIS EDGE

Step 9: Try Edge (5,7) → ❌ CYCLE DETECTED!

Final MST (7 edges for 8 vertices):
1———2———5———6———8
|    |              |
└————4———3         7

MST Edges Selected: {(1,2), (3,4), (5,6), (7,8), (2,4), (2,5), (6,8)}
Rejected Edges: {(1,3), (5,7)} - Both would create cycles
Total MST Weight: 1+2+3+4+5+6+8 = 29
```

Graphical Representation

Instead of maintaining actual sets, we use a tree structure where each set has a representative (root). Let's trace through all steps to see how these trees evolve.

Step-by-Step Tree Evolution

Initial State - Each vertex is its own tree (forest of single nodes):

```
Step 0: Universal Set - Each element is its own parent

1     2     3     4     5     6     7     8
o     o     o     o     o     o     o     o
|     |     |     |     |     |     |     |
1     2     3     4     5     6     7     8
(each points to itself)

Sets: {1} {2} {3} {4} {5} {6} {7} {8}
```

Step 1: Process Edge (1,2) - Union of vertices 1 and 2

```
Find(1) = 1, Find(2) = 2 → Different sets → Union them

Before Union:     After Union:
1     2           1
o     o           o
|     |           |
1     2           1
                  ↑
                  2

Tree Structure:   Sets: {1,2} {3} {4} {5} {6} {7} {8}
    1             Remaining: 3  4  5  6  7  8
    |                        o  o  o  o  o  o
    2
```

Step 2: Process Edge (3,4) - Union of vertices 3 and 4

```
Find(3) = 3, Find(4) = 4 → Different sets → Union them

Tree Structures:    Sets: {1,2} {3,4} {5} {6} {7} {8}
    1       3       Remaining: 5  6  7  8
    |       |                  o  o  o  o
    2       4
```

Step 3: Process Edge (5,6) - Union of vertices 5 and 6

```
Find(5) = 5, Find(6) = 6 → Different sets → Union them

Tree Structures:        Sets: {1,2} {3,4} {5,6} {7} {8}
    1       3       5   Remaining: 7  8
    |       |       |              o  o
    2       4       6
```

Step 4: Process Edge (7,8) - Union of vertices 7 and 8

```
Find(7) = 7, Find(8) = 8 → Different sets → Union them

Tree Structures:            Sets: {1,2} {3,4} {5,6} {7,8}
    1       3       5       7
    |       |       |       |
    2       4       6       8
```

Step 5: Process Edge (2,4) - Union of sets containing 2 and 4

```
Find(2): 2 → 1 (root = 1)
Find(4): 4 → 3 (root = 3)
Different roots → Union based on weight

Weight comparison: Set{1,2} has 2 nodes, Set{3,4} has 2 nodes
Equal weights → Make 1 the parent of 3 (arbitrary choice)

Before Union:           After Union:
    1       3               1
    |       |              /|\
    2       4             2 3 4

Tree Structures:            Sets: {1,2,3,4} {5,6} {7,8}
        1           5       7
       /|\          |       |
      2 3 4         6       8
```

Step 6: Process Edge (2,5) - Union of sets containing 2 and 5

```
Find(2): 2 → 1 (root = 1)
Find(5): 5 → 5 (root = 5)
Different roots → Union based on weight

Weight comparison: Set{1,2,3,4} has 4 nodes, Set{5,6} has 2 nodes
Set 1 is larger → Make 1 the parent of 5

Before Union:               After Union:
    1           5               1
```

```
  /|\          |            /|\\
 2 3 4         6           2 3 4 5
                                 |
                                 6
```

```
Tree Structures:                Sets: {1,2,3,4,5,6} {7,8}
       1              7
      /|\\            |
     2 3 4 5         8
           |
           6
```

Step 7: Process Edge (1,3) - CYCLE DETECTION!

```
Find(1): 1 → 1 (root = 1)
Find(3): 3 → 1 (root = 1)
SAME ROOT! → Both belong to same set → CYCLE DETECTED!
```

```
Current Tree:    Path showing cycle:
    1            1 — 2 — 4 — 3 — 1 (would form cycle)
   /|\\          ↑             ↑
  2 3 4 5       └─────────────┘
        |       Adding edge (1,3) creates this cycle
        6
```

❌ REJECT Edge (1,3) - No changes to tree structure

Step 8: Process Edge (6,8) - Union of sets containing 6 and 8

```
Find(6): 6 → 5 → 1 (root = 1)
Find(8): 8 → 7 (root = 7)
Different roots → Union based on weight

Weight comparison: Set{1,2,3,4,5,6} has 6 nodes, Set{7,8} has 2 nodes
Set 1 is larger → Make 1 the parent of 7
```

```
Final Tree Structure:        Sets: {1,2,3,4,5,6,7,8}
       1
      /|\\\
     2 3 4 5 7
           | |
           6 8
```

Step 9: Process Edge (5,7) - CYCLE DETECTION!

```
Find(5): 5 → 1 (root = 1)
Find(7): 7 → 1 (root = 1)
SAME ROOT! → Both belong to same set → CYCLE DETECTED!
```

❌ REJECT Edge (5,7) - No changes to tree structure

Final Tree Representation

```
Complete Disjoint Set Tree:

        1 (Root/Representative)
      /-|-\-\-\
     2  3  4  5  7
           |  |
           6  8

Set: {1,2,3,4,5,6,7,8}
Root: 1
Height: 3
```

Key Concepts Illustrated

Tree Properties:

- **Root**: Representative of the entire set (vertex 1)
- **Parent-Child**: Each child points to its parent
- **Path Compression**: Can be applied to make trees flatter
- **Union by Rank**: Larger trees become parents of smaller trees

Find Operation:

```
Find(6): 6 → 5 → 1 ✓ (result: 1)
Find(8): 8 → 7 → 1 ✓ (result: 1)
Find(3): 3 → 1 ✓     (result: 1)
```

Union Operation Decision Making:

- Compare tree sizes/ranks
- Attach smaller tree under root of larger tree
- Keeps overall tree height minimal

Array Implementation

Data Structure

```
parent[] = array where parent[i] represents parent of element i
parent[i] = -1 means i is a root (representative of its set)
parent[i] = -k means i is root with k elements in its set
```

Initial State - Universal Set

```
Index:  [1] [2] [3] [4] [5] [6] [7] [8]
Value:  -1  -1  -1  -1  -1  -1  -1  -1

Graph visualization of Universal Set:
1     2     3     4     5     6     7     8
o     o     o     o     o     o     o     o
```

```
↑    ↑    ↑    ↑    ↑    ↑    ↑    ↑
-1   -1   -1   -1   -1   -1   -1   -1
(Each element is its own parent/root)

Disjoint Sets Representation:
{1}  {2}  {3}  {4}  {5}  {6}  {7}  {8}
```

Step-by-step Array Changes

Let's trace through each step showing both the array representation and corresponding tree structure.

Initial State - Universal Set:

```
Index:  [1] [2] [3] [4] [5] [6] [7] [8]
Value:  -1  -1  -1  -1  -1  -1  -1  -1

Tree Structure:
1    2    3    4    5    6    7    8
o    o    o    o    o    o    o    o

Explanation: Each element is its own parent (value -1 means root with 1 element)
```

Step 1: Process Edge (1,2)

```
Find(1) = 1 (parent[1] = -1, so 1 is root)
Find(2) = 2 (parent[2] = -1, so 2 is root)
Union(1,2): Make 1 parent of 2, update count

Index:  [1] [2] [3] [4] [5] [6] [7] [8]
Value:  -2   1  -1  -1  -1  -1  -1  -1

Tree Structure:
    1    3    4    5    6    7    8
   /     o    o    o    o    o    o
  2

Explanation:
- parent[1] = -2 (root with 2 elements)
- parent[2] = 1 (2's parent is 1)
```

Step 2: Process Edge (3,4)

```
Find(3) = 3 (parent[3] = -1, so 3 is root)
Find(4) = 4 (parent[4] = -1, so 4 is root)
Union(3,4): Make 3 parent of 4, update count

Index:  [1] [2] [3] [4] [5] [6] [7] [8]
Value:  -2   1  -2   3  -1  -1  -1  -1

Tree Structure:
    1    3    5    6    7    8
   /    /     o    o    o    o
  2    4

Explanation:
```

```
- parent[3] = -2 (root with 2 elements)
- parent[4] = 3 (4's parent is 3)
```

Step 3: Process Edge (5,6)

```
Find(5) = 5 (parent[5] = -1, so 5 is root)
Find(6) = 6 (parent[6] = -1, so 6 is root)
Union(5,6): Make 5 parent of 6, update count

Index:  [1] [2] [3] [4] [5] [6] [7] [8]
Value:  -2   1  -2   3  -2   5  -1  -1

Tree Structure:
    1     3     5     7     8
   /     /     /      o     o
  2     4     6

Explanation:
- parent[5] = -2 (root with 2 elements)
- parent[6] = 5 (6's parent is 5)
```

Step 4: Process Edge (7,8)

```
Find(7) = 7 (parent[7] = -1, so 7 is root)
Find(8) = 8 (parent[8] = -1, so 8 is root)
Union(7,8): Make 7 parent of 8, update count

Index:  [1] [2] [3] [4] [5] [6] [7] [8]
Value:  -2   1  -2   3  -2   5  -2   7

Tree Structure:
    1     3     5     7
   /     /     /     /
  2     4     6     8

Explanation:
- parent[7] = -2 (root with 2 elements)
- parent[8] = 7 (8's parent is 7)
```

Step 5: Process Edge (2,4) - Union of different sets

```
Find(2): 2 → parent[2] = 1 → parent[1] = -2 ✓ (root = 1)
Find(4): 4 → parent[4] = 3 → parent[3] = -2 ✓ (root = 3)
Different roots! Weight comparison: both have -2 (2 elements each)
Union(1,3): Make 1 parent of 3 (arbitrary choice since equal weights)

Index:  [1] [2] [3] [4] [5] [6] [7] [8]
Value:  -4   1   1   3  -2   5  -2   7

Tree Structure:
        1         5     7
      / | \      /     /
     2  3  4    6     8

Explanation:
- parent[1] = -4 (root with 4 elements: 1,2,3,4)
```

```
- parent[3] = 1 (3's parent is now 1)
- Total elements in set 1: 2 + 2 = 4
```

Step 6: Process Edge (2,5) - Union of different sets

```
Find(2): 2 → parent[2] = 1 → parent[1] = -4 ✓ (root = 1)
Find(5): 5 → parent[5] = -2 ✓ (root = 5)
Different roots! Weight comparison: parent[1] = -4, parent[5] = -2
Set 1 is larger (4 > 2), so make 1 parent of 5

Index:  [1] [2] [3] [4] [5] [6] [7] [8]
Value:  -6   1   1   3   1   5  -2   7

Tree Structure:
          1           7
      /  /|\ \       /
     2  3 4  5      8
            |
            6

Explanation:
- parent[1] = -6 (root with 6 elements: 1,2,3,4,5,6)
- parent[5] = 1 (5's parent is now 1)
- Total elements in set 1: 4 + 2 = 6
```

Step 7: Process Edge (1,3) - CYCLE DETECTION!

```
Find(1): parent[1] = -6 ✓ (root = 1)
Find(3): 3 → parent[3] = 1 → parent[1] = -6 ✓ (root = 1)
SAME ROOT! Both belong to set with root 1 → CYCLE DETECTED!

Index:  [1] [2] [3] [4] [5] [6] [7] [8]
Value:  -6   1   1   3   1   5  -2   7
        ↑ NO CHANGES - Edge rejected ↑

Tree Structure: (No changes)
          1           7
      /  /|\ \       /
     2  3 4  5      8
            |
            6

❌ Edge (1,3) would create cycle: 1→2→4→3→1
```

Step 8: Process Edge (6,8) - Union of different sets

```
Find(6): 6 → parent[6] = 5 → parent[5] = 1 → parent[1] = -6 ✓ (root = 1)
Find(8): 8 → parent[8] = 7 → parent[7] = -2 ✓ (root = 7)
Different roots! Weight comparison: parent[1] = -6, parent[7] = -2
Set 1 is larger (6 > 2), so make 1 parent of 7

Index:  [1] [2] [3] [4] [5] [6] [7] [8]
Value:  -8   1   1   3   1   5   1   7

Final Tree Structure:
            1
        / / | \ \ \
```

```
       2 3  4  5  7
              |  |
              6  8

Explanation:
- parent[1] = -8 (root with 8 elements: all vertices)
- parent[7] = 1 (7's parent is now 1)
- All vertices now belong to one connected component
```

Step 9: Process Edge (5,7) - CYCLE DETECTION!

```
Find(5): 5 → parent[5] = 1 → parent[1] = -8 ✓ (root = 1)
Find(7): 7 → parent[7] = 1 → parent[1] = -8 ✓ (root = 1)
SAME ROOT! Both belong to set with root 1 → CYCLE DETECTED!

Index:  [1] [2] [3] [4] [5] [6] [7] [8]
Value:  -8   1   1   3   1   5   1   7
        ↑ NO CHANGES - Edge rejected ↑
```

❌ Edge (5,7) would create another cycle in the graph

Array Representation Summary

Key Array Conventions:

- **Negative values**: Indicate root nodes, magnitude = number of elements in set
- **Positive values**: Point to parent node
- **Find Operation**: Follow parent pointers until reaching negative value (root)
- **Union Operation**: Make root of smaller set point to root of larger set

Final State Analysis:

```
Index:  [1] [2] [3] [4] [5] [6] [7] [8]
Value:  -8   1   1   3   1   5   1   7

Root: 1 (parent[1] = -8, meaning 8 elements)
Tree paths:
- Find(2): 2 → 1 ✓
- Find(3): 3 → 1 ✓
- Find(4): 4 → 3 → 1 ✓
- Find(5): 5 → 1 ✓
- Find(6): 6 → 5 → 1 ✓
- Find(7): 7 → 1 ✓
- Find(8): 8 → 7 → 1 ✓
```

Find Operation Implementation

```python
def find(x):
    if parent[x] < 0:
        return x  # x is root
    else:
        return find(parent[x])  # recursively find root
```

Union Operation Implementation

```
def union(x, y):
    root_x = find(x)
    root_y = find(y)

    if root_x == root_y:
        return "CYCLE DETECTED"

    # Weighted union: smaller tree becomes subtree of larger tree
    if parent[root_x] < parent[root_y]:  # root_x has more elements
        parent[root_x] += parent[root_y]
        parent[root_y] = root_x
    else:
        parent[root_y] += parent[root_x]
        parent[root_x] = root_y
```

Optimizations

1. Union by Rank (Weight)

- Always attach the smaller tree under the root of the larger tree
- Keeps the tree height minimal
- **Time Complexity**: O(log n) for find operation

2. Path Compression (Collapsing Find)

- During find operation, make every node point directly to the root
- Subsequent find operations become O(1)

Before Path Compression:

```
    1
   /
  3
 /
4
```

After finding 4:

```
    1
   /|
  3 4
```

Implementation:

```
def find_with_compression(x):
    if parent[x] < 0:
        return x
    else:
```

```
        parent[x] = find_with_compression(parent[x])  # Path compression
        return parent[x]
```

Combined Time Complexity

With both optimizations: **Nearly O(1)** amortized time for both Union and Find operations.

Applications

1. Kruskal's Minimum Spanning Tree Algorithm

- Sort edges by weight
- For each edge, check if it creates a cycle using Union-Find
- If no cycle, include the edge in MST

2. Network Connectivity

- Determine if two nodes are connected
- Find connected components in a network

3. Image Processing

- Connected component labeling
- Region growing algorithms

4. Social Network Analysis

- Find groups of connected people
- Detect communities in social graphs

Implementation Alternatives

Linked List Representation

- Each set is a linked list
- First element is the representative
- Union by appending one list to another
- **Time Complexity**: O(n) for union in worst case

Comparison Table

Operation	Array (Basic)	Array (Optimized)	Linked List
Find	O(n)	O(α(n)) ≈ O(1)	O(1)
Union	O(n)	O(α(n)) ≈ O(1)	O(n)
Space	O(n)	O(n)	O(n)

α(n) is the inverse Ackermann function, which is practically constant for all reasonable values of n.

Summary

Disjoint Sets (Union-Find) is a powerful data structure for:

- ✅ Cycle detection in undirected graphs
- ✅ Finding connected components
- ✅ Kruskal's MST algorithm
- ✅ Dynamic connectivity queries

Key Takeaways:

1. Two main operations: Find and Union
2. Cycle detected when both endpoints of an edge belong to the same set
3. Optimizations make operations nearly constant time
4. Simple array implementation is sufficient for most use cases

Disjoint Set

Design a disjoint set (also called union-find) data structure that efficiently manages a collection of non-overlapping sets in a dynamic graph environment. Think of it like organizing people into groups where groups can merge over time, and you need to quickly check if two people belong to the same group.

The data structure should support:

- **Initialization**: Create n separate sets, each containing one element
- **Union by Rank**: Merge two sets using rank optimization (keeps trees shallow based on height)
- **Union by Size**: Merge two sets using size optimization (attaches smaller component to larger)
- **Find**: Check if two elements belong to the same set by comparing their ultimate parents

This is particularly useful for **dynamic graphs** - graphs that continuously change their configuration as edges are added one by one.

Examples

Example 1

Input

```
["DisjointSet", "unionByRank", "unionBySize", "find", "find"]
[[5], [0, 1], [2, 3], [0, 1], [0, 3]]
```

Output

```
[null, null, null, true, false]
```

Explanation

```
ds = DisjointSet(5)     # Initialize with 5 elements: {0}, {1}, {2}, {3}, {4}
ds.unionByRank(0, 1)    # Merge sets: {0,1}, {2}, {3}, {4}
ds.unionBySize(2, 3)    # Merge sets: {0,1}, {2,3}, {4}
ds.find(0, 1)           # True - both in same set {0,1}
ds.find(0, 3)           # False - 0 in {0,1}, 3 in {2,3}
```

Example 2

Input

```
["DisjointSet", "unionBySize", "unionBySize", "find", "find"]
[[3], [0, 1], [1, 2], [0, 2], [0, 1]]
```

Output

```
[null, null, null, true, true]
```

Explanation

```
ds = DisjointSet(3)     # Initialize: {0}, {1}, {2}
ds.unionBySize(0, 1)    # Merge: {0,1}, {2}
ds.unionBySize(1, 2)    # Merge all: {0,1,2}
ds.find(0, 2)           # True - all in same set
ds.find(0, 1)           # True - all in same set
```

Think of it like merging friend groups on social media - when two people become friends, their entire friend networks can potentially connect.

Solution

Use arrays to track parent-child relationships and component properties:

- **Parent array**: stores the ultimate parent (root) of each node
- **Rank array**: tracks tree height for union by rank optimization
- **Size array**: tracks component size for union by size optimization
- **Path compression**: flattens tree structure during find operations for efficiency

Intuition

Imagine organizing a company merger scenario:

- **Union by Rank**: When companies merge, the one with deeper management hierarchy becomes the parent (keeps structure shallow)
- **Union by Size**: The larger company absorbs the smaller one (more intuitive and maintains accurate sizes)
- **Path compression**: Employees get direct reporting lines to the CEO after each query (flattens hierarchy)

The key insight is maintaining efficient tree structures while enabling fast connectivity queries. Union by Size is often preferred because it's more intuitive and doesn't get distorted by path compression like ranks do.

Approach Steps

1. **Initialize**:
    - Each element starts as its own parent: `parent[i] = i`
    - Set initial rank = 0 and size = 1 for all elements
2. **Find Operation with Path Compression**:
    - Follow parent pointers until reaching root (self-parent)
    - During backtracking, connect each node directly to root
    - This flattens the tree for future queries
3. **Union by Rank**:
    - Find ultimate parents of both nodes
    - If different, attach tree with smaller rank under tree with larger rank
    - If ranks equal, choose one as parent and increment its rank
    - **Note**: Ranks can become inaccurate after path compression

4. **Union by Size**:
    - Find ultimate parents of both nodes
    - If different, attach smaller component under larger component
    - Update size of new root by adding both component sizes
    - **Advantage**: Sizes remain accurate even with path compression
5. **Find Query**:
    - Use find operation to get ultimate parents of both nodes
    - Return true if ultimate parents are the same

Code

```
class DisjointSet:
    def __init__(self, n: int):
        """
        Initialize Disjoint Set (Union-Find) for n elements.
        Each node is its own parent initially.
        """
        self.parent = [i for i in range(n)]       # parent[i] = parent of i
        self.rank = [0] * n                       # rank[i] = height of tree rooted at i
        self.size = [1] * n                       # size[i] = size of component rooted at i

    def find_ultimate_parent(self, node: int) -> int:
        """
        Find the ultimate parent (root) of a node with path compression.
        Path compression: Connect each node directly to root during backtracking.
        """
        if node == self.parent[node]:
            return node

        # Path compression: directly connect node to its ultimate parent
        self.parent[node] = self.find_ultimate_parent(self.parent[node])
        return self.parent[node]

    def union_by_rank(self, u: int, v: int):
        """
        Union two sets by rank (tree height).
        Attach tree with smaller rank under tree with larger rank.
        """
        parent_u = self.find_ultimate_parent(u)
        parent_v = self.find_ultimate_parent(v)

        if parent_u == parent_v:
            return  # Already in same set

        # Attach smaller rank tree under larger rank tree
        if self.rank[parent_u] < self.rank[parent_v]:
            self.parent[parent_u] = parent_v
        elif self.rank[parent_v] < self.rank[parent_u]:
            self.parent[parent_v] = parent_u
        else:
            # Equal ranks: choose one as parent and increment its rank
            self.parent[parent_v] = parent_u
            self.rank[parent_u] += 1

    def union_by_size(self, u: int, v: int):
```

```
        """
        Union two sets by size (number of elements).
        Attach smaller component under larger component.
        """
        parent_u = self.find_ultimate_parent(u)
        parent_v = self.find_ultimate_parent(v)

        if parent_u == parent_v:
            return  # Already in same set

        # Attach smaller component under larger component
        if self.size[parent_u] < self.size[parent_v]:
            self.parent[parent_u] = parent_v
            self.size[parent_v] += self.size[parent_u]
        else:
            self.parent[parent_v] = parent_u
            self.size[parent_u] += self.size[parent_v]

    def find(self, u: int, v: int) -> bool:
        """
        Check if two nodes belong to the same set.
        Returns True if they have the same ultimate parent.
        """
        return self.find_ultimate_parent(u) == self.find_ultimate_parent(v)

# Example Usage
ds = DisjointSet(5)

# Initial state: {0}, {1}, {2}, {3}, {4}
print(ds.find(0, 1))    # False

ds.union_by_rank(0, 1)   # {0,1}, {2}, {3}, {4}
ds.union_by_size(2, 3)   # {0,1}, {2,3}, {4}

print(ds.find(0, 1))    # True
print(ds.find(0, 3))    # False
print(ds.find(2, 3))    # True

ds.union_by_size(1, 2)   # {0,1,2,3}, {4}
print(ds.find(0, 3))    # True
```

Time and Space Complexity

Time Complexity:

- **Find with Path Compression**: O(α(n)) - Nearly constant time, where α is the inverse Ackermann function
- **Union by Rank/Size**: O(α(n)) - Same as find since we need to find roots first
- **Overall**: O(4α) ≈ O(1) for practical purposes (α(n) ≤ 4 for any reasonable input size)

Space Complexity:

- **O(n)** - We need arrays for parent, rank, and size, each of size n
- **Total**: O(3n) = O(n)

Why Union by Size is Preferred:

- **Intuitive**: Size represents actual number of elements, easier to understand
- **Robust**: Sizes remain accurate even after path compression, unlike ranks

- **Performance**: Same time complexity as union by rank but with more meaningful metadata

Dynamic Graph Advantage: Instead of running DFS/BFS (O(N+E)) every time the graph changes, disjoint set allows:

- Adding edges: O(α(n)) per edge
- Connectivity queries: O(α(n)) per query
- Perfect for scenarios where graph structure changes frequently and connectivity queries are frequent

Simple Disjoint Set

Design a disjoint set (also called union-find) data structure that efficiently manages a collection of non-overlapping sets. Think of it like organizing people into groups - initially everyone is in their own group, but you can merge groups together and quickly find which group someone belongs to.

The data structure should support:

- **Initialization**: Create n separate sets, each containing one element
- **Union by Rank**: Merge two sets using rank optimization (keeps trees shallow)
- **Union by Size**: Merge two sets using size optimization (attaches smaller tree to larger)
- **Find**: Determine which set an element belongs to (find the group leader)

Examples

Input

```
# Create disjoint set with 5 elements (0, 1, 2, 3, 4)
ds = DisjointSet(4)

# Initially: {0}, {1}, {2}, {3}, {4}
print(ds.find(0))  # 0 (0 is its own parent)
print(ds.find(1))  # 1 (1 is its own parent)

# Union 1 and 2
ds.union_by_rank(1, 2)
# Now: {0}, {1, 2}, {3}, {4}

# Union 2 and 3
ds.union_by_rank(2, 3)
# Now: {0}, {1, 2, 3}, {4}
```

Output

```
print(ds.find(0))  # 0 (still its own parent)
print(ds.find(1))  # 1 (representative of {1, 2, 3})
print(ds.find(2))  # 1 (same group as 1)
print(ds.find(3))  # 1 (same group as 1)
```

Explanation

Think of it like a company hierarchy:

- Initially, each person is their own "department head"
- When departments merge, one head becomes the boss of the other
- To find someone's ultimate boss, you keep going up the chain until you reach the top
- Path compression is like creating direct shortcuts to the top boss for faster future lookups

Solution

Use two arrays to track relationships:

- **Parent array**: stores the immediate parent of each node
- **Rank/Size arrays**: help decide which tree should become the parent during union operations
- **Path compression**: flattens the tree structure during find operations for efficiency

Intuition

Imagine you're organizing a tournament bracket:

- **Union by Rank**: When merging teams, put the smaller bracket under the larger one to keep the overall structure shallow
- **Union by Size**: Attach the team with fewer members to the team with more members
- **Path compression**: Once you find the champion, create direct connections so future searches are faster

The key insight is that we want to keep our "family trees" as flat as possible to make searches super fast.

Approach Steps

1. **Initialize**:
    - Each element starts as its own parent
    - Set initial rank = 0 and size = 1 for all elements
2. **Find Operation**:
    - Follow parent pointers until you reach the root (self-parent)
    - Apply path compression: make every node point directly to the root
3. **Union by Rank**:
    - Find roots of both elements
    - If they're the same, do nothing (already connected)
    - Attach the tree with smaller rank under the tree with larger rank
    - If ranks are equal, choose one as parent and increment its rank
4. **Union by Size**:
    - Find roots of both elements
    - Attach the smaller tree under the larger tree
    - Update the size of the new root

Code

```python
class DisjointSet:
    def __init__(self, n: int):
        """
        Initialize Disjoint Set (Union-Find) for n elements.
        Each node is its own parent initially, with rank = 0 and size = 1.
        """
        self.parent = [i for i in range(n + 1)]   # parent[i] = parent of i
        self.rank = [0] * (n + 1)                 # rank[i] = rank of tree rooted at i
        self.size = [1] * (n + 1)                 # size[i] = size of tree rooted at i

    def find(self, node: int) -> int:
        """
        Find the ultimate parent (representative) of a node.
        Uses path compression optimization.
        """
        if node == self.parent[node]:
            return node
        # Path compression: directly connect node to its ultimate parent
        self.parent[node] = self.find(self.parent[node])
        return self.parent[node]

    def union_by_size(self, u: int, v: int):
        """
        Union two sets by size.
        Attach the smaller tree under the larger one.
        """
        parent_u = self.find(u)
        parent_v = self.find(v)

        if parent_u == parent_v:
            return  # already in the same set

        if self.size[parent_u] < self.size[parent_v]:
            self.parent[parent_u] = parent_v
            self.size[parent_v] += self.size[parent_u]
        else:
            self.parent[parent_v] = parent_u
            self.size[parent_u] += self.size[parent_v]

    def union_by_rank(self, u: int, v: int):
        """
        Union two sets by rank.
        Attach the smaller rank tree under the higher rank tree.
        If ranks are equal, attach one under the other and increase rank.
        """
        parent_u = self.find(u)
        parent_v = self.find(v)

        if parent_u == parent_v:
            return  # already in the same set

        if self.rank[parent_u] < self.rank[parent_v]:
            self.parent[parent_u] = parent_v
        elif self.rank[parent_v] < self.rank[parent_u]:
            self.parent[parent_v] = parent_u
        else:
            self.parent[parent_v] = parent_u
            self.rank[parent_u] += 1
```

```python
# Example Usage
ds = DisjointSet(5)
ds.union_by_rank(1, 2)
ds.union_by_rank(3, 4)
ds.union_by_size(2, 4)
print(ds.find(1) == ds.find(4))  # True - they're in the same set
```

Time and Space Complexity

Time Complexity:

- **Find**: O(α(n)) - Nearly constant time due to path compression, where α is the inverse Ackermann function
- **Union**: O(α(n)) - Same as find since we need to find roots first
- In practical terms, α(n) ≤ 4 for any reasonable input size, so it's essentially O(1)

Space Complexity:

- **O(n)** - We need three arrays (parent, rank, size) each of size n

Why it's so fast:

- Path compression flattens the tree during find operations
- Union by rank/size keeps trees balanced and shallow
- Together, these optimizations make operations nearly constant time

Real-world analogy: It's like having a company directory that gets automatically updated with shortcuts every time someone looks up their boss's boss - eventually, everyone has a direct line to the CEO!

Perfect question 🚀. Disjoint Set Union (DSU), also called **Union-Find**, is one of the most useful data structures in Graphs and Network-related problems. Let's go step by step.

🔷 What is a Disjoint Set (Union-Find)?

A **Disjoint Set** is a data structure that keeps track of a **partition of elements into groups (or sets)**.

- Each element belongs to exactly one set.
- You can quickly find which set an element belongs to (`find` operation).
- You can merge two sets together (`union` operation).

This is super useful when you need to know **connectivity**: 👉 "Are these two elements in the same group or not?"

🔷 Core Operations

1. **Find** → Identify which group an element belongs to.
2. **Union** → Merge two groups into one.
3. **Connected?** → Check if two elements are in the same set.

🔷 Real-Time Use Cases with Examples

1. Cycle Detection in Graphs

📌 **Problem:** You are given an undirected graph. Check if it has a cycle.

👉 **How DSU helps:**

- Initially, each node is in its own set.
- For every edge `(u, v)`:
  - If `find(u) == find(v)`, then `u` and `v` are already connected → adding this edge creates a cycle.
  - Otherwise, union them.

✅ **Real-life analogy:** Think of **roads between cities**. If connecting two cities creates a loop of roads where you can return to the starting point without retracing, that's a cycle.

2. Minimum Spanning Tree (MST) – Kruskal's Algorithm

📌 **Problem:** You want to connect all cities with roads such that the total road cost is minimum (no cycles allowed).

👉 **How DSU helps:**

- Sort all edges by weight.
- For each edge `(u, v)`:
  - If `u` and `v` are in different sets → union them and add the edge to MST.
  - If they are in the same set → skip (to avoid cycles).

✅ **Real-life analogy:** Building a **telecom network** or **electricity grid** → connect all stations with minimum wire cost while ensuring no unnecessary loops.

3. Network Connectivity

📌 **Problem:** You are given computers and cables. You need to check if all computers are connected or if extra cables are required.

👉 **How DSU helps:**

- Each computer starts in its own set.
- For each cable `(u, v)`, do union.
- At the end, check if all computers have the same parent (or count unique parents).

✅ **Real-life analogy:** Think of **LAN connections in an office**. If some computers are isolated, they won't communicate → DSU helps detect and fix this.

4. Dynamic Connectivity (Friend Circles / Social Networks)

📌 **Problem:** You have a social network. When two people become friends, they should belong to the same group.

👉 **How DSU helps:**

- Initially, each person is alone.
- On each "friendship" `(u, v)`, do union.
- To check if two people are connected (directly or indirectly), just check `find(u) == find(v)`.

✅ **Real-life analogy:** Facebook/LinkedIn → Checking if **two users are in the same friend circle**.

5. Image Processing (Connected Components in a Grid)

📌 **Problem:** Given a binary image (0 = background, 1 = object), count connected components of `1`s.

👉 **How DSU helps:**

- Treat each pixel as a node.
- If two adjacent pixels are `1`, do union.
- At the end, count distinct parents → number of connected objects.

✅ **Real-life analogy:** Medical Imaging (MRI scans) → Identify separate tumors in an image.

6. Percolation / Clustering Problems

📌 **Problem:** Given points in 2D/3D, group them into clusters if they are close enough.

👉 **How DSU helps:**

- Treat each point as a node.
- If distance between two points < threshold → union them.
- Finally, count number of groups.

✅ **Real-life analogy:** Geographical clustering (villages within a certain distance form a “district”).

7. Accounts Merge Problem (Leetcode 721)

📌 **Problem:** You are given multiple user accounts with emails. If two accounts share at least one email, they belong to the same user.

👉 **How DSU helps:**

- Each account/email is a node.
- Union accounts if they share an email.
- At the end, collect merged groups.

✅ **Real-life analogy:** Bank KYC system → Merge duplicate accounts of the same customer.

🔷 Summary Table

Use Case	Real-world Analogy
Cycle Detection	Detecting loops in road maps
MST (Kruskal)	Building cheapest telecom/electric grid
Network Connectivity	Ensuring computers in a LAN are connected
Social Networks	Friend circles (are two people connected?)
Image Processing	Identifying separate objects in MRI scans
Clustering	Forming districts from nearby villages
Accounts Merge	Merging duplicate user accounts in banking/social media

👉 So, in **one line**: **Disjoint Set is best whenever you need to manage "groups" dynamically and answer queries like: "Are these two in the same group?" or "Merge these two groups."**

Would you like me to also **draw a visual flow (with diagrams) for one of these use cases** (like MST or cycle detection) so you can use it in your YouTube/explaining content?

Minimum Spanning Tree (MST) – Notes

Problem Description

We are given a **weighted, undirected, and connected graph** with `V` vertices (numbered from 0 to V-1) and `E` edges. Each edge is represented as `[ai, bi, wi]`, where:

- `ai` and `bi` are the endpoints of the edge
- `wi` is the weight of the edge

We need to **find the sum of the weights of the edges in the Minimum Spanning Tree (MST)** of the graph.

👉 A **Minimum Spanning Tree (MST)** is a subset of edges of a graph that:

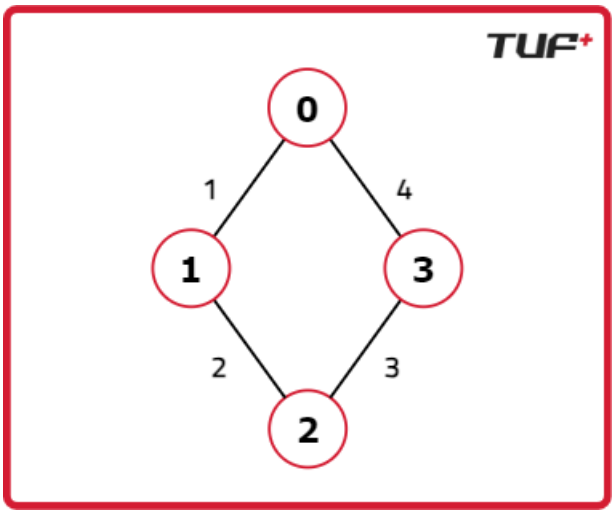
- Connects all vertices
- Contains exactly `V-1` edges (no cycles)
- Has the **minimum possible total edge weight**

Examples

Example 1

Input:

```
V = 4
Edges = [[0, 1, 1], [1, 2, 2], [2, 3, 3], [0, 3, 4]]
```



Output:

```
6
```

Explanation:

- Chosen edges for MST:

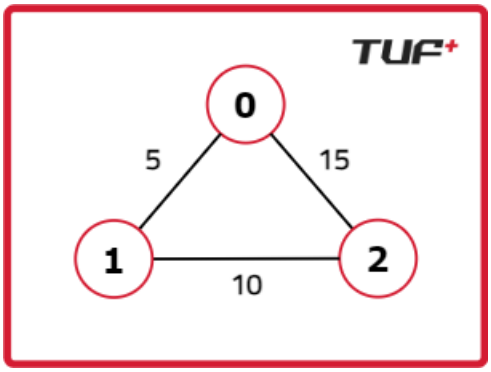
- [0, 1, 1]
- [1, 2, 2]
- [2, 3, 3]

- Total MST weight = 1 + 2 + 3 = **6**

Example 2

Input:

```
V = 3
Edges = [[0, 1, 5], [1, 2, 10], [2, 0, 15]]
```



Output:

```
15
```

Explanation:

- Chosen edges for MST:
  - [0, 1, 5]
  - [1, 2, 10]
- Total MST weight = 5 + 10 = **15**

Solution

We can solve this problem using two famous algorithms:

1. **Prim's Algorithm** – Grow the MST by choosing the smallest edge from the tree to a new vertex.
2. **Kruskal's Algorithm** – Choose the smallest edges one by one, avoiding cycles (using Disjoint Set/Union-Find).

Both algorithms guarantee the minimum spanning tree weight.

Intuition

- The MST must always choose the **smallest possible edges** that connect all nodes without forming cycles.

- **Greedy approach works** here because choosing the smallest available edge at each step still leads to the optimal MST.
-
  - In **Prim's Algorithm**, we start from a node and keep expanding with the smallest available edge.
  - In **Kruskal's Algorithm**, we sort edges by weight and keep picking the smallest edge if it doesn't form a cycle.

Approach Steps

Prim's Algorithm

1. Start with an arbitrary node.
2. Use a **min-heap** to always pick the smallest edge leading to an unvisited node.
3. Mark the node as visited and add the edge weight to the MST sum.
4. Push all edges from this node to the heap.
5. Repeat until all nodes are visited.

Kruskal's Algorithm

1. Extract all edges from the graph.
2. Sort edges by their weight.
3. Initialize a **Disjoint Set (Union-Find)** to keep track of connected components.
4. For each edge in sorted order:
   - If the two nodes are not already connected, add the edge to MST.
   - Union their sets in the disjoint set.
5. Continue until we have exactly `V-1` edges in MST.

Code

Prim's Algorithm (Python)

```python
import heapq

def prim_mst(nodes, graph):
    src = nodes[0]  # Start from the first node
    visited = {i: False for i in nodes}  # Keep track of visited nodes
    queue = []  # Min-heap for edges
    heapq.heappush(queue, (0, src))  # Push starting node with weight 0
    MSTWeight = 0  # Total MST weight

    while queue:
        weight, node = heapq.heappop(queue)  # Get edge with minimum weight
        if visited[node]:
            continue  # Skip if node already visited
        visited[node] = True  # Mark node as visited
        MSTWeight += weight  # Add weight to MST
```

```
        # Push all adjacent edges into heap
        for adj_node, adj_weight in graph[node]:
            if not visited[adj_node]:
                heapq.heappush(queue, (adj_weight, adj_node))
    return MSTWeight

class Solution:
    def spanningTree(self, V, adj):
        return prim_mst(list(range(V)), adj)
```

Kruskal's Algorithm (Python)

```
class DisjointSet:
    def __init__(self, n):
        # Parent array and size array for Union-Find
        self.parent = [i for i in range(n + 1)]
        self.size = [1] * (n + 1)

    def find(self, node):
        # Path compression to find root parent
        if node == self.parent[node]:
            return node
        self.parent[node] = self.find(self.parent[node])
        return self.parent[node]

    def union(self, u, v):
        # Union by size
        parent_u = self.find(u)
        parent_v = self.find(v)
        if parent_u == parent_v:
            return False  # Already connected
        if self.size[parent_u] < self.size[parent_v]:
            self.parent[parent_u] = parent_v
            self.size[parent_v] += self.size[parent_u]
        else:
            self.parent[parent_v] = parent_u
            self.size[parent_u] += self.size[parent_v]
        return True

def kruskal_mst(nodes, graph):
    edges = []
    # Collect all edges (avoid duplicates in undirected graph)
    for node in graph:
        for adj_node, adj_weight in graph[node]:
            if node < adj_node:
                edges.append((adj_weight, node, adj_node))

    # Sort edges by weight
    edges.sort()
    ds = DisjointSet(len(nodes))
    mst_weight = 0

    # Process edges in increasing weight order
    for wt, u, v in edges:
        if ds.union(u, v):  # If u and v are not already connected
            mst_weight += wt  # Add edge weight to MST
    return mst_weight

class Solution:
```

```
    def spanningTree(self, V, adj):
        return kruskal_mst(list(range(V)), adj)
```

Time and Space Complexity

Prim's Algorithm

- **Time Complexity:** `O(E log V)`
  - Each edge can be pushed into the heap → `E log V`
- **Space Complexity:** `O(V + E)`
  - For adjacency list and heap

Kruskal's Algorithm

- **Time Complexity:** `O(E log E)` (sorting edges) + `O(E α(V))` (Union-Find operations)
  - Effectively `O(E log E)`
- **Space Complexity:** `O(V + E)`
  - For edge list and disjoint set

Number of Operations to Make Network Connected

Problem Description

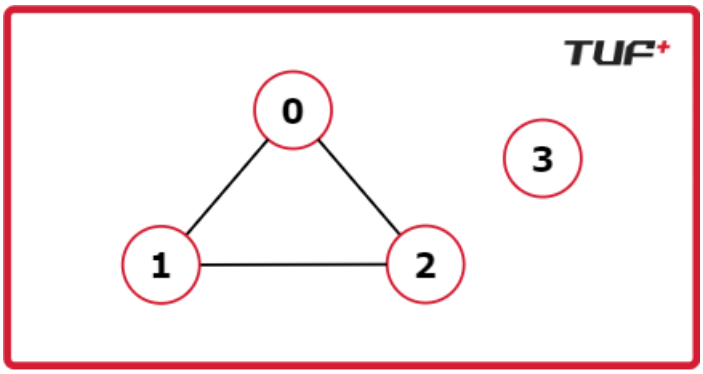
You have a network with `n` computers (vertices) and some cables (edges) connecting them. You can remove any cable and place it between any two computers in one operation. Find the minimum number of operations needed to make all computers connected to each other. If it's impossible, return -1.

Think of it like having several groups of computers that can talk to each other within their group, but different groups can't communicate. You need to "bridge" these groups by moving cables around.

Examples

Input

```
n = 4, edges = [[0,1], [0,2], [1,2]]
```



Output

```
1
```

Explanation

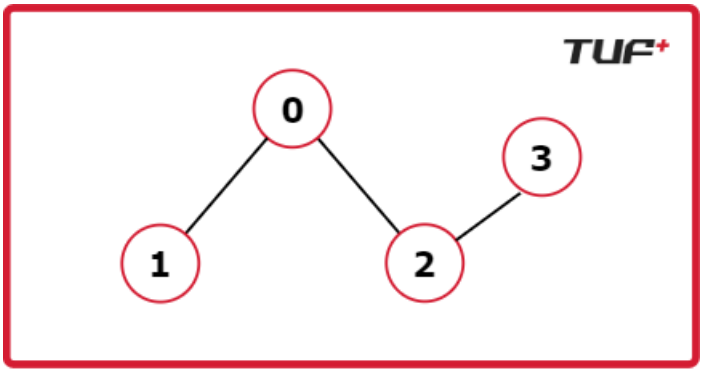
Initially, we have computers 0, 1, 2 connected to each other, but computer 3 is isolated. We have 3 cables but only need 2 to keep computers 0, 1, 2 connected. So we can remove the "extra" cable (1,2) and use it to connect computer 3 to the main group.

Visual representation:

```
Before:     After 1 operation:
0---1       0---1
|\ /|       |   |
| X |  →    |   |
|/ \|       |   |
2   3       2---3
(isolated)  (all connected)
```

Input

```
n = 9, edges = [[0,1],[0,2],[0,3],[1,2],[2,3],[4,5],[5,6],[7,8]]
```



Output ``` 2 ```

Explanation

We have 3 separate groups: {0,1,2,3}, {4,5,6}, and {7,8}. We need 2 operations to connect all 3 groups together.

Solution

Use Union-Find (Disjoint Set) to identify connected components, then calculate how many "bridges" we need to connect all components. We need exactly `(number of components - 1)` operations.

Intuition

Key Insight: To connect `k` separate islands, you need exactly `k-1` bridges.

Detailed Diagram - Why This Works

```
Case 1: Sufficient edges available
Components: A, B, C (3 components)
Need: 3-1 = 2 bridges

A: 0---1---2    B: 3---4    C: 5
   (3 nodes,       (2 nodes,   (1 node,
    2 edges)       1 edge)     0 edges)

Total edges: 2 + 1 + 0 = 3 edges
Minimum needed for connectivity: 3 components - 1 = 2 bridges
Extra edges available: 3 - 2 = 1 extra edge

Since we have at least 2 total edges, we can:
1. Remove 1 extra edge from component A: 0---2
2. Use it to connect A to B: A---B
3. Remove 1 edge from component B: 3---4
4. Use it to connect B to C: B---C

Result: A---B---C (all connected with 2 operations)
```

```
Case 2: Insufficient edges
Components: A, B, C, D (4 components)
Each component has only 1 node: 0, 1, 2, 3
Total edges: 0
Need: 4-1 = 3 bridges
Available: 0 edges

Since 0 < 3, it's impossible → return -1
```

Union-Find Visualization

```
Initial state (n=6, edges=[[0,1],[2,3],[4,5]]):
Parent: [0, 1, 2, 3, 4, 5]
         0  1  2  3  4  5

After processing edge [0,1]:
Parent: [0, 0, 2, 3, 4, 5]
         0  1  2  3  4  5
Component 1: {0,1}

After processing edge [2,3]:
Parent: [0, 0, 2, 2, 4, 5]
         0  1  2  3  4  5
Component 1: {0,1}, Component 2: {2,3}

After processing edge [4,5]:
Parent: [0, 0, 2, 2, 4, 4]
         0  1  2  3  4  5
Component 1: {0,1}, Component 2: {2,3}, Component 3: {4,5}

Counting components (nodes that are their own parent):
- Node 0: parent[0] = 0 ✓ (component root)
- Node 2: parent[2] = 2 ✓ (component root)
- Node 4: parent[4] = 4 ✓ (component root)

Total components: 3
Operations needed: 3 - 1 = 2
```

Approach Steps

1. **Feasibility Check**: If total edges < (n-1), return -1
    - A connected graph with n nodes needs at least n-1 edges
2. **Initialize Union-Find**: Create disjoint set for n vertices
3. **Process All Edges**: For each edge [u,v], union u and v
    - This groups vertices into connected components
4. **Count Components**: Count vertices that are their own parent
    - These represent the "root" of each connected component
5. **Calculate Operations**: Return (number of components - 1)

- This is the minimum bridges needed to connect all components

Code

```
class DisjointSet:
    def __init__(self, n: int):
        """
        Initialize Disjoint Set (Union-Find) for n elements.
        Each node is its own parent initially, with size = 1.
        """
        self.parent = [i for i in range(n)]
        self.size = [1] * n

    def find(self, node: int) -> int:
        """
        Find the ultimate parent (representative) of a node.
        Uses path compression optimization.
        """
        if node == self.parent[node]:
            return node
        # Path compression: directly connect node to its ultimate parent
        self.parent[node] = self.find(self.parent[node])
        return self.parent[node]

    def union_by_size(self, u: int, v: int):
        """
        Union two sets by size.
        Attach the smaller tree under the larger one.
        """
        parent_u = self.find(u)
        parent_v = self.find(v)

        if parent_u == parent_v:
            return  # already in the same set

        if self.size[parent_u] < self.size[parent_v]:
            self.parent[parent_u] = parent_v
            self.size[parent_v] += self.size[parent_u]
        else:
            self.parent[parent_v] = parent_u
            self.size[parent_u] += self.size[parent_v]

def solution(n, edges):
    """
    Find minimum operations to make network connected.

    Args:
        n: number of vertices
        edges: list of edges [u, v]

    Returns:
        minimum operations needed, or -1 if impossible
    """
    # Check if we have enough edges to potentially connect the graph
    if len(edges) < n - 1:
        return -1

    # Initialize Union-Find data structure
    ds = DisjointSet(n)
```

```
        # Process all edges to form connected components
        for u, v in edges:
            ds.union_by_size(u, v)

        # Count number of connected components
        connected_components = 0
        for node in range(n):
            if ds.find(node) == node:  # node is root of its component
                connected_components += 1

        # Operations needed = components - 1
        return connected_components - 1

class Solution:
    def solve(self, n, edges):
        return solution(n, edges)
```

Time and Space Complexity

Time Complexity: O(n + m × α(n))

- Processing m edges with Union-Find operations: O(m × α(n))
- Counting components by checking n vertices: O(n × α(n))
- α(n) is the inverse Ackermann function, practically constant
- **Simplified: O(n + m)** for practical purposes

Space Complexity: O(n)

- Union-Find uses parent array and size array, each of size n
- No additional space proportional to number of edges

Complexity Explanation in Simple Terms

- **Time**: We look at each edge once and each vertex once. Union-Find operations are nearly instant due to optimizations.
- **Space**: We only store information about each vertex (its parent and component size), so space grows linearly with the number of vertices.

Key Insights

- **Minimum edges for connectivity**: Any connected graph with n vertices needs at least n-1 edges
- **Component bridging**: To connect k components, you need exactly k-1 "bridge" edges
- **Edge redistribution**: We can always rearrange existing edges as long as we have enough total edges

Accounts Merge

Problem Description

Given a list of accounts where each account contains a name followed by emails, merge accounts that belong to the same person. Two accounts belong to the same person if they share at least one common email address. People can have the same name but be different individuals, so we only merge based on shared emails.

Examples

Input

```
N = 4
accounts = [
    ["John", "johnsmith@mail.com", "john_newyork@mail.com"],
    ["John", "johnsmith@mail.com", "john00@mail.com"],
    ["Mary", "mary@mail.com"],
    ["John", "johnnybravo@mail.com"]
]
```

Output

```
[
    ["John", "john00@mail.com", "john_newyork@mail.com", "johnsmith@mail.com"],
    ["Mary", "mary@mail.com"],
    ["John", "johnnybravo@mail.com"]
]
```

Explanation

Think of this like organizing scattered business cards. The first two John accounts share "johnsmith@mail.com", so they belong to the same person and get merged. Mary's account stands alone. The third John account has no common emails with others, so it remains separate. It's like finding that two business cards have the same phone number - they must belong to the same person!

Solution

Use **Disjoint Set Union (DSU)** to efficiently group accounts that share emails. Map each email to its first occurrence, and when we see the same email again, union those accounts together.

Intuition

Visual Representation

```
Initial State:
Account 0: John → [johnsmith@mail.com, john_newyork@mail.com]
Account 1: John → [johnsmith@mail.com, john00@mail.com]
Account 2: Mary → [mary@mail.com]
Account 3: John → [johnnybravo@mail.com]

Email Mapping Process:
```

```
┌─────────────────────────────────────────────────────┐
│ Step 1: Process Account 0                           │
│ johnsmith@mail.com     → maps to Account 0          │
│ john_newyork@mail.com  → maps to Account 0          │
└─────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────┐
│ Step 2: Process Account 1                           │
│ johnsmith@mail.com     → ALREADY SEEN! Union(0, 1)  │
│ john00@mail.com        → maps to Account 1          │
│                                                     │
│ DSU State: Account 0 and 1 are now connected        │
└─────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────┐
│ Step 3: Process Account 2                           │
│ mary@mail.com          → maps to Account 2          │
└─────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────┐
│ Step 4: Process Account 3                           │
│ johnnybravo@mail.com   → maps to Account 3          │
└─────────────────────────────────────────────────────┘

Final Connected Components:
Component 1: {Account 0, Account 1} → Same person
Component 2: {Account 2} → Different person
Component 3: {Account 3} → Different person
```

DSU Tree Visualization

```
After Union Operations:
    0 (John)
    │
    └── 1 (John)  [connected via johnsmith@mail.com]

    2 (Mary)      [standalone]

    3 (John)      [standalone]

Final Groups:
Group 1: All emails from accounts 0 & 1
Group 2: All emails from account 2
Group 3: All emails from account 3
```

The key insight is that **shared emails create bridges between accounts**. Like connecting islands with bridges - if Island A connects to Island B, and Island B connects to Island C, then all three islands form one connected landmass!

Approach Steps

1. **Initialize DSU**: Create a Disjoint Set Union structure for all accounts
2. **Map emails to accounts**: For each account, process all its emails:
    - If email seen for first time → map it to current account
    - If email already exists → union current account with the account that has this email

3. **Group emails by parent**: After all unions, group emails by their ultimate parent account
4. **Format result**: For each group, sort emails and prepend the person's name

Code

```python
class DisjointSet:
    def __init__(self, n: int):
        """
        Initialize Disjoint Set (Union-Find) for n elements.
        Each node is its own parent initially, with size = 1.
        """
        self.parent = [i for i in range(n)]
        self.size = [1] * n

    def find(self, node: int) -> int:
        """
        Find the ultimate parent (representative) of a node.
        Uses path compression optimization.
        """
        if node == self.parent[node]:
            return node
        # Path compression: directly connect node to its ultimate parent
        self.parent[node] = self.find(self.parent[node])
        return self.parent[node]

    def union_by_size(self, u: int, v: int):
        """
        Union two sets by size.
        Attach the smaller tree under the larger one.
        """
        parent_u = self.find(u)
        parent_v = self.find(v)

        if parent_u == parent_v:
            return  # already in the same set

        if self.size[parent_u] < self.size[parent_v]:
            self.parent[parent_u] = parent_v
            self.size[parent_v] += self.size[parent_u]
        else:
            self.parent[parent_v] = parent_u
            self.size[parent_u] += self.size[parent_v]

class Solution:
    def accountsMerge(self, accounts):
        n = len(accounts)
        email_to_account = {}  # email -> (account_index, name)
        ds = DisjointSet(n)

        # Step 1: Process each account and union accounts with common emails
        for i in range(n):
            name = accounts[i][0]
            for j in range(1, len(accounts[i])):
                email = accounts[i][j]

                if email in email_to_account:
                    # Found common email - union current account with existing one
                    existing_account = email_to_account[email][0]
                    ds.union_by_size(existing_account, i)
```

```
            else:
                # First time seeing this email
                email_to_account[email] = (i, name)

    # Step 2: Group emails by their ultimate parent account
    merged_accounts = {}  # parent_account -> list of emails
    account_names = {}    # parent_account -> name

    for email, (account_index, name) in email_to_account.items():
        parent = ds.find(account_index)

        if parent not in merged_accounts:
            merged_accounts[parent] = []
            account_names[parent] = name

        merged_accounts[parent].append(email)

    # Step 3: Format the result - sort emails and prepend name
    result = []
    for parent in merged_accounts:
        email_list = sorted(merged_accounts[parent])
        result.append([account_names[parent]] + email_list)

    return result
```

Time and Space Complexity

Time Complexity: O(N × E × α(N) + N × E × log(E))

- **N** = number of accounts, **E** = average emails per account
- **O(N × E × α(N))**: Processing all emails and performing union operations (α is inverse Ackermann function, practically constant)
- **O(N × E × log(E))**: Sorting emails within each merged account
- **Overall**: Effectively **O(N × E × log(E))** since sorting dominates

Space Complexity: O(N + E)

- **O(N)**: DSU parent and size arrays
- **O(E)**: Hash map storing email-to-account mappings
- **O(E)**: Result storage for merged accounts

Simplified Explanation

- **Time**: We visit each email once to build connections, then sort emails within each group. Like organizing mail - first we group letters by recipient, then sort each person's mail alphabetically.
- **Space**: We need memory to track which emails belong to whom and the family tree structure of merged accounts.

Number of islands II

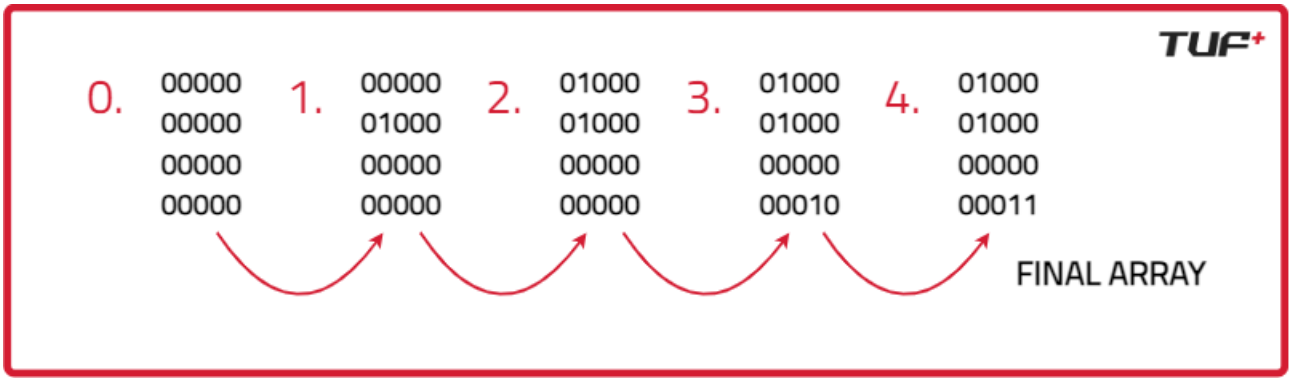
Problem Description

Given an n×m grid initially filled with water (all 0s), you'll receive k operations. Each operation converts a water cell to land (changes 0 to 1) at position [row, col]. After each operation, count how many separate islands exist. An island is a group of connected land cells (connected horizontally or vertically, not diagonally).

Examples

Input

```
n = 4, m = 5, k = 4
A = [[1,1], [0,1], [3,3], [3,4]]
```



Output

```
[1, 1, 2, 2]
```

Explanation

Think of this like dropping stones into a pond to create islands:

Operation 1: Drop stone at (1,1) → Creates 1st island → Count = 1 **Operation 2**: Drop stone at (0,1) → Connects to existing island at (1,1) → Count = 1
Operation 3: Drop stone at (3,3) → Creates separate 2nd island → Count = 2 **Operation 4**: Drop stone at (3,4) → Connects to island at (3,3) → Count = 2

It's like building with LEGO blocks - when you place a new block next to existing ones, they connect to form a larger structure!

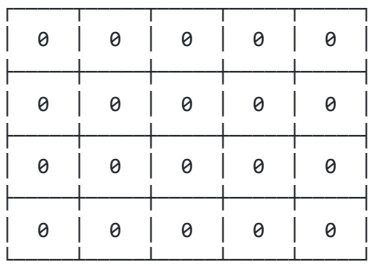
Solution

Use **Disjoint Set Union (DSU)** to dynamically track connected components as we add land cells. Each time we add land, we check if it connects to existing neighboring islands and merge them accordingly.

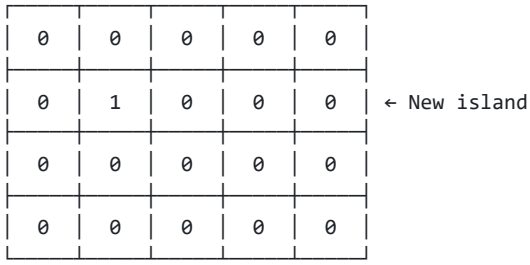
Intuition

Visual Step-by-Step Process

Initial Grid (4×5): All water

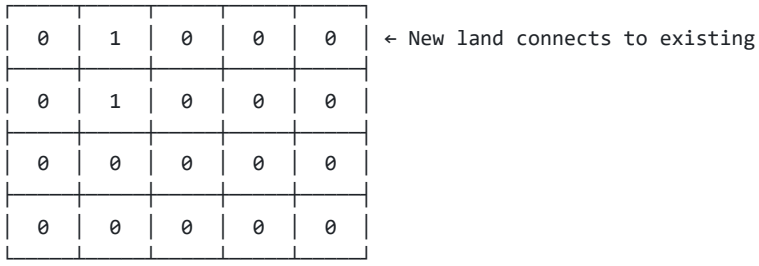


Operation 1: Add land at (1,1)



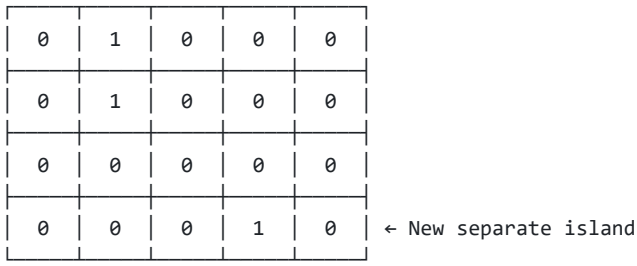
Islands: 1

Operation 2: Add land at (0,1)



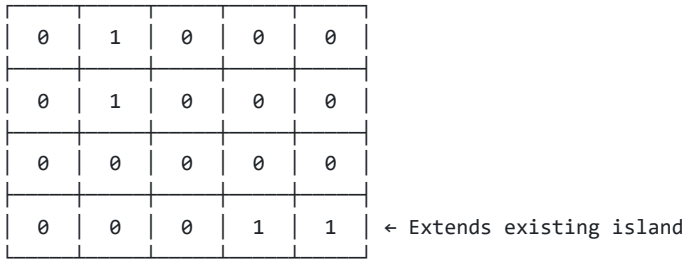
Islands: 1 (merged with existing)

Operation 3: Add land at (3,3)



Islands: 2

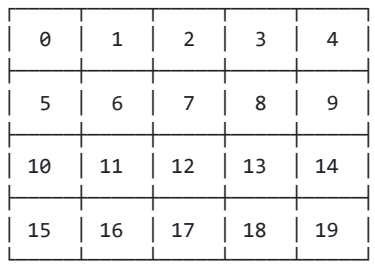
Operation 4: Add land at (3,4)



Islands: 2 (connected to existing)

Cell-to-Node Mapping System

```
Grid Coordinates → Node Numbers (for DSU)
```




```
Formula: Node = row × m + col
Example: Cell (1,1) → Node = 1×5 + 1 = 6
```

DSU Connection Process

```
When adding land at (1,1):
1. Create new island → Count = 1
2. Check neighbors: up(0,1), down(2,1), left(1,0), right(1,2)
3. No land neighbors found → Island count remains 1

When adding land at (0,1):
1. Create new island → Count = 2
2. Check neighbors: up(-1,1)❌, down(1,1)✅, left(0,0)❌, right(0,2)❌
3. Found land neighbor at (1,1) → Union nodes 1 and 6 → Count = 1

Connection Visualization:
Before Union:    After Union:
Node 1: [1]      Node 6: [1, 6]  ← 6 becomes parent
Node 6: [6]      Node 1: points to 6
Count: 2         Count: 1
```

The key insight is that **each new land cell starts as its own island, then merges with neighboring islands**. It's like dropping puzzle pieces - each piece starts separate, but connects to adjacent pieces to form larger pictures!

Approach Steps

1. **Initialize DSU**: Create Disjoint Set for all n×m cells and a visited matrix
2. **For each operation**:
   - Convert water cell to land at given position
   - If already land, skip (add current count to result)
   - If new land, increment island count by 1
   - Check all 4 neighbors (up, down, left, right)
   - For each land neighbor, try to union with current cell
   - If union successful (weren't already connected), decrement island count
   - Add current island count to result
3. **Return result array**

Code

```python
class DisjointSet:
    def __init__(self, n: int):
        """
        Initialize Disjoint Set (Union-Find) for n elements.
        Each node is its own parent initially, with size = 1.
        """
        self.parent = [i for i in range(n)]
        self.size = [1] * n

    def find(self, node: int) -> int:
        """
        Find the ultimate parent (representative) of a node.
        Uses path compression optimization.
        """
        if node == self.parent[node]:
            return node
        # Path compression: directly connect node to its ultimate parent
        self.parent[node] = self.find(self.parent[node])
        return self.parent[node]

    def union_by_size(self, u: int, v: int) -> bool:
        """
        Union two sets by size.
        Returns True if union happened, False if already connected.
        """
        parent_u = self.find(u)
        parent_v = self.find(v)

        if parent_u == parent_v:
            return False  # already in the same set

        # Union by size: attach smaller tree under larger one
        if self.size[parent_u] < self.size[parent_v]:
            self.parent[parent_u] = parent_v
            self.size[parent_v] += self.size[parent_u]
        else:
            self.parent[parent_v] = parent_u
            self.size[parent_u] += self.size[parent_v]

        return True  # successful union

class Solution:
    def numOfIslands(self, n, m, operations):
        # Initialize DSU for all cells and visited matrix
        ds = DisjointSet(n * m)
        visited = [[False] * m for _ in range(n)]

        # Direction vectors for 4-directional movement (up, down, left, right)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

        island_count = 0
        result = []

        for row, col in operations:
            # Skip if cell is already land
            if visited[row][col]:
                result.append(island_count)
                continue
```

```
            # Convert water to land
            visited[row][col] = True
            island_count += 1  # Start as new island

            # Convert 2D coordinates to 1D node number
            current_node = row * m + col

            # Check all 4 neighbors
            for dr, dc in directions:
                new_row, new_col = row + dr, col + dc

                # Check if neighbor is within bounds and is land
                if (0 <= new_row < n and 0 <= new_col < m and
                    visited[new_row][new_col]):

                    neighbor_node = new_row * m + new_col

                    # Try to union current cell with neighbor
                    if ds.union_by_size(current_node, neighbor_node):
                        island_count -= 1  # Successful merge reduces count

            result.append(island_count)

        return result
```

Time and Space Complexity

Time Complexity: O(K × α(N×M))

- **K** = number of operations
- **α(N×M)** = inverse Ackermann function (practically constant)
- For each operation, we check 4 neighbors and perform DSU operations
- **Overall**: Effectively **O(K)** since α is nearly constant

Space Complexity: O(N×M + K)

- **O(N×M)**: DSU parent and size arrays for all cells
- **O(N×M)**: Visited matrix to track land cells
- **O(K)**: Result array storing island count after each operation

Simplified Explanation

- **Time**: For each stone we drop, we check its 4 neighbors and possibly merge islands. Like checking if a new LEGO piece connects to existing structures - very fast per operation.
- **Space**: We need to remember which cells have land and track how islands are connected, plus store the answer for each operation.

Making a large island

Problem Description

Given an n×n binary grid, you can change **at most one** 0 to 1. Find the size of the largest possible island after this operation. An island is a group of connected 1s (connected horizontally or vertically). If the grid is already all 1s, return the total grid size.

Examples

Input

```
grid = [[1,0],[0,1]]
```

Output

```
3
```

Explanation

We can change either 0 to 1. Changing any 0 connects the two separate 1s into one island of size 3. It's like building a bridge between two separate islands to create one large landmass!

Input

```
grid = [[1,1],[1,1]]
```

Output

```
4
```

Explanation

The grid is already completely filled with land - the largest possible island already exists with size 4.

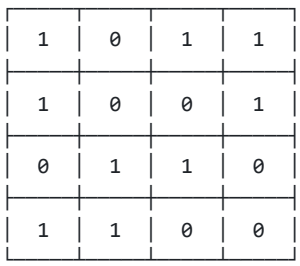
Solution

Use **Disjoint Set Union (DSU)** to pre-compute all existing island sizes, then for each water cell (0), simulate converting it to land and calculate the potential island size by summing unique neighboring island sizes.

Intuition

Visual Problem Breakdown

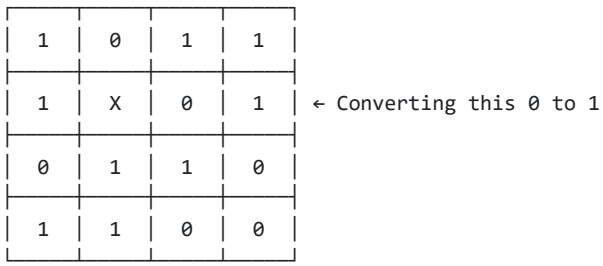
Original Grid:



Initial Islands (before any changes):
Island A: {(0,0), (1,0)} → Size 2
Island B: {(0,2), (0,3), (1,3)} → Size 3
Island C: {(2,1), (2,2), (3,0), (3,1)} → Size 4

Strategy Visualization

Testing position (1,1) - Convert 0 → 1:

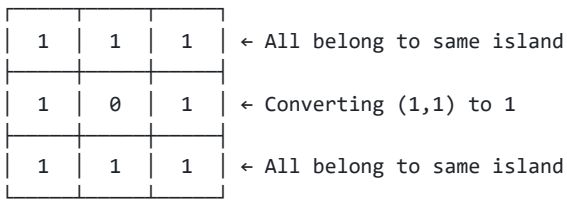


Check neighbors of (1,1):
↑ (0,1): 0 (water) - no connection
↓ (2,1): 1 (land) - connects to Island C (size 4)
← (1,0): 1 (land) - connects to Island A (size 2)
→ (1,2): 0 (water) - no connection

Potential island size = 1 (new cell) + 4 (Island C) + 2 (Island A) = 7

Key Challenge: Avoiding Double Counting

Problematic Case:



Without proper handling:
- Up neighbor: Island size 3
- Down neighbor: Island size 3
- Left neighbor: Island size 3
- Right neighbor: Island size 3
- Total: 1 + 3 + 3 + 3 + 3 = 13 ❌ (WRONG!)

```
With DSU ultimate parent tracking:
- All neighbors have same ultimate parent
- Add island size only once: 1 + 8 = 9 ✅ (CORRECT!)
```

DSU Parent Tracking System

```
Cell Numbering (for 3×3 grid):
┌─────┬─────┬─────┐
│  0  │  1  │  2  │
├─────┼─────┼─────┤
│  3  │  4  │  5  │
├─────┼─────┼─────┤
│  6  │  7  │  8  │
└─────┴─────┴─────┘

DSU After Initial Processing:
parent[0] → 0, parent[1] → 0, parent[2] → 0  (merged into component 0)
parent[3] → 0, parent[5] → 0  (merged into component 0)
parent[6] → 0, parent[7] → 0, parent[8] → 0  (merged into component 0)

Ultimate Parent: 0 (represents the entire island of size 8)
```

The brilliant insight is that **DSU automatically handles the merging complexity** - we just need to track unique parent components when calculating potential island sizes!

Approach Steps

1. **Build initial islands**: Use DSU to connect all existing land cells (1s) into their respective islands
2. **For each water cell (0)**:
   - Assume we convert it to land (adds +1 to size)
   - Check all 4 neighbors
   - For each land neighbor, find its ultimate parent (island representative)
   - Use a set to track unique neighboring islands (avoids double counting)
   - Sum sizes of all unique neighboring islands
3. **Track maximum**: Keep track of the largest possible island size found
4. **Handle edge case**: If no water cells exist, return total grid size

Code

```python
class DisjointSet:
    def __init__(self, n: int):
        """
        Initialize Disjoint Set (Union-Find) for n elements.
        Each node is its own parent initially, with size = 1.
        """
        self.parent = [i for i in range(n)]
        self.size = [1] * n

    def find(self, node: int) -> int:
        """
        Find the ultimate parent (representative) of a node.
        Uses path compression optimization.
```

```
        """
        if node == self.parent[node]:
            return node
        # Path compression: directly connect node to its ultimate parent
        self.parent[node] = self.find(self.parent[node])
        return self.parent[node]

    def union_by_size(self, u: int, v: int):
        """
        Union two sets by size.
        Returns True if union happened, False if already connected.
        """
        parent_u = self.find(u)
        parent_v = self.find(v)

        if parent_u == parent_v:
            return False  # already in the same set

        # Union by size: attach smaller tree under larger one
        if self.size[parent_u] < self.size[parent_v]:
            self.parent[parent_u] = parent_v
            self.size[parent_v] += self.size[parent_u]
        else:
            self.parent[parent_v] = parent_u
            self.size[parent_u] += self.size[parent_v]

        return True

class Solution:
    def largestIsland(self, grid):
        n = len(grid)
        ds = DisjointSet(n * n)

        # Direction vectors for 4-directional movement (up, down, left, right)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

        # Step 1: Build initial islands by connecting adjacent land cells
        for row in range(n):
            for col in range(n):
                if grid[row][col] == 1:
                    current_node = row * n + col

                    # Check all 4 neighbors
                    for dr, dc in directions:
                        new_row, new_col = row + dr, col + dc

                        # If neighbor is within bounds and is land, union them
                        if (0 <= new_row < n and 0 <= new_col < n and
                            grid[new_row][new_col] == 1):
                            neighbor_node = new_row * n + new_col
                            ds.union_by_size(current_node, neighbor_node)

        # Step 2: Try converting each water cell to land
        max_island_size = 0
        has_water = False

        for row in range(n):
            for col in range(n):
                if grid[row][col] == 0:
                    has_water = True

                    # Set to store unique neighboring island parents
```

```
                unique_parents = set()
                potential_size = 1  # Adding the current cell

                # Check all 4 neighbors
                for dr, dc in directions:
                    new_row, new_col = row + dr, col + dc

                    # If neighbor is within bounds and is land
                    if (0 <= new_row < n and 0 <= new_col < n and
                        grid[new_row][new_col] == 1):
                        neighbor_node = new_row * n + new_col
                        parent = ds.find(neighbor_node)

                        # Add to set to avoid double counting
                        unique_parents.add(parent)

                # Sum sizes of all unique neighboring islands
                for parent in unique_parents:
                    potential_size += ds.size[parent]

                max_island_size = max(max_island_size, potential_size)

        # Step 3: Handle edge case - if grid has no water (all 1s)
        if not has_water:
            return n * n

        return max_island_size
```

Time and Space Complexity

Time Complexity: $O(N^2 \times \alpha(N^2))$

- N^2 = total cells in the grid
- $\alpha(N^2)$ = inverse Ackermann function (practically constant)
- **Phase 1**: $O(N^2)$ to traverse grid and perform union operations
- **Phase 2**: $O(N^2)$ to check each water cell and its neighbors
- **Overall**: Effectively $O(N^2)$ since α is nearly constant

Space Complexity: $O(N^2)$

- $O(N^2)$: DSU parent and size arrays for all N^2 cells
- **O(1)**: Additional space for sets and variables (constant per operation)

Simplified Explanation

- **Time**: We visit each cell twice - once to build initial islands, once to test conversion possibilities. Like surveying land twice - first to map existing territories, then to plan optimal expansion.
- **Space**: We need to track the "family tree" of connected cells and remember which cells belong to which island group.

Most stones removed with same row or column

Problem Description

Given n stones placed on a 2D plane at integer coordinates, find the maximum number of stones that can be removed. A stone can only be removed if it shares the same row OR column with another stone that hasn't been removed yet. The goal is to remove as many stones as possible while following this rule.

Examples

Input

```
n = 6
stones = [[0,0], [0,1], [1,0], [1,2], [2,1], [2,2]]
```

Output

```
5
```

Explanation

We can remove 5 stones, leaving only 1 stone remaining. One possible removal sequence: Remove [0,0] (shares row with [0,1]), then [1,0] (shares column with [0,0]), and so on. The last stone cannot be removed as it would have no stones sharing its row or column.

Input

```
n = 6
stones = [[0,0], [0,2], [1,3], [3,1], [3,2], [4,3]]
```

Output

```
4
```

Explanation

We can remove 4 stones from the 6 total stones. The stones form 2 separate groups, and from each group we must leave at least 1 stone.

Solution

Use **Disjoint Set Union (DSU)** to group stones that are connected through shared rows or columns. The maximum stones we can remove equals **total stones minus the number of connected components** (since we must leave at least 1 stone per component).

Intuition

Core Insight: Connected Components

```
Key Formula: Max Removable = Total Stones - Number of Components

Why? In each connected group, we can remove all stones except the last one!
```

Visual Problem Analysis

```
Example 1 Visualization:
stones = [[0,0], [0,1], [1,0], [1,2], [2,1], [2,2]]

Grid Representation:
    Col0  Col1  Col2
Row0  ●     ●     ○     ← Stones at (0,0) and (0,1)
Row1  ●     ○     ●     ← Stones at (1,0) and (1,2)
Row2  ○     ●     ●     ← Stones at (2,1) and (2,2)

Connection Analysis:
- (0,0) connects to (0,1) via Row 0
- (0,0) connects to (1,0) via Col 0
- (0,1) connects to (2,1) via Col 1
- (1,0) connects to (1,2) via Row 1
- (1,2) connects to (2,2) via Col 2
- (2,1) connects to (2,2) via Row 2

All stones are transitively connected → 1 component
Max removable = 6 - 1 = 5 stones
```

Revolutionary Approach: Row-Column Union

Instead of connecting stones directly, we connect **rows to columns**!

```
Traditional Approach (Complex):
Stone (0,0) ↔ Stone (0,1) ↔ Stone (1,0) ↔ ...
Need to check every pair of stones = O(N²)

DSU Approach (Elegant):
Row 0 ↔ Col 0 (because stone at (0,0))
Row 0 ↔ Col 1 (because stone at (0,1))
Row 1 ↔ Col 0 (because stone at (1,0))
...

Node Mapping System:
Rows: 0, 1, 2, 3, ...
Cols: maxRow+1, maxRow+2, maxRow+3, ... (offset to avoid collision)

Example with maxRow = 2:
Row nodes: 0, 1, 2
Col nodes: 3, 4, 5 (representing cols 0, 1, 2)
```

Step-by-Step DSU Process

```
Stone Positions: [[0,0], [0,1], [1,0], [1,2], [2,1], [2,2]]
Max Row = 2, Max Col = 2
Node mapping: Rows = {0,1,2}, Cols = {3,4,5}

Step 1: Process stone (0,0)
Union(Row0=0, Col0=3)
Components: {0,3}, {1}, {2}, {4}, {5}

Step 2: Process stone (0,1)
Union(Row0=0, Col1=4)
Components: {0,3,4}, {1}, {2}, {5}

Step 3: Process stone (1,0)
Union(Row1=1, Col0=3) → But 3 is already connected to 0
Components: {0,1,3,4}, {2}, {5}

Step 4: Process stone (1,2)
Union(Row1=1, Col2=5) → 1 is connected to main component
Components: {0,1,3,4,5}, {2}

Step 5: Process stone (2,1)
Union(Row2=2, Col1=4) → 4 is connected to main component
Components: {0,1,2,3,4,5}

Step 6: Process stone (2,2)
Union(Row2=2, Col2=5) → Both already in main component
Components: {0,1,2,3,4,5}

Final: 1 connected component
Answer: 6 stones - 1 component = 5 removable stones
```

Why This Works: Transitive Connection Magic

```
Row-Column bridges create transitive connections:

Stone A(0,0) → Row0 ← Stone B(0,1) → Col1 ← Stone C(2,1)

Even though A and C don't share row/column directly,
they're connected through the Row0-Col1 bridge!

This is like a relay race where runners pass the baton:
Stone → Row → Column → Stone → Row → Column → ...
```

Approach Steps

1. **Find grid dimensions**: Determine maxRow and maxCol from all stone positions
2. **Initialize DSU**: Create DSU with enough nodes for all rows and columns
3. **Map coordinates to nodes**:
   - Row i → Node i
   - Column j → Node (maxRow + 1 + j)
4. **Union row-column pairs**: For each stone at (r,c), union row node r with column node (maxRow + 1 + c)

5. **Count unique components**: Find how many unique parent nodes exist among all stone-occupied rows and columns
6. **Calculate result**: Return total stones minus number of components

Code

```python
class DisjointSet:
    def __init__(self, n: int):
        """
        Initialize Disjoint Set (Union-Find) for n elements.
        Each node is its own parent initially, with size = 1.
        """
        self.parent = [i for i in range(n)]
        self.size = [1] * n

    def find(self, node: int) -> int:
        """
        Find the ultimate parent (representative) of a node.
        Uses path compression optimization.
        """
        if node == self.parent[node]:
            return node
        # Path compression: directly connect node to its ultimate parent
        self.parent[node] = self.find(self.parent[node])
        return self.parent[node]

    def union_by_size(self, u: int, v: int):
        """
        Union two sets by size.
        Returns True if union happened, False if already connected.
        """
        parent_u = self.find(u)
        parent_v = self.find(v)

        if parent_u == parent_v:
            return False  # already in the same set

        # Union by size: attach smaller tree under larger one
        if self.size[parent_u] < self.size[parent_v]:
            self.parent[parent_u] = parent_v
            self.size[parent_v] += self.size[parent_u]
        else:
            self.parent[parent_v] = parent_u
            self.size[parent_u] += self.size[parent_v]

        return True

class Solution:
    def maxRemove(self, stones, n):
        # Step 1: Find the maximum row and column indices
        max_row = max_col = 0
        for row, col in stones:
            max_row = max(max_row, row)
            max_col = max(max_col, col)

        # Step 2: Initialize DSU with enough nodes for all rows and columns
        # Rows: 0 to max_row, Columns: (max_row + 1) to (max_row + 1 + max_col)
        total_nodes = max_row + max_col + 2
        ds = DisjointSet(total_nodes)
```

```
        # Step 3: Union each stone's row with its column
        for row, col in stones:
            row_node = row
            col_node = max_row + 1 + col  # Offset columns to avoid collision
            ds.union_by_size(row_node, col_node)

        # Step 4: Count unique connected components
        # Only consider nodes that actually have stones
        unique_parents = set()
        for row, col in stones:
            row_node = row
            col_node = max_row + 1 + col

            # Add ultimate parents of both row and column nodes
            unique_parents.add(ds.find(row_node))
            unique_parents.add(ds.find(col_node))

        # Step 5: Calculate result
        # Each component can remove all stones except 1
        num_components = len(unique_parents)
        return n - num_components

# Alternative cleaner implementation
class SolutionOptimized:
    def maxRemove(self, stones, n):
        max_row = max_col = 0
        for row, col in stones:
            max_row = max(max_row, row)
            max_col = max(max_col, col)

        ds = DisjointSet(max_row + max_col + 2)
        stone_nodes = set()  # Track nodes that have stones

        # Union rows with columns and track stone nodes
        for row, col in stones:
            row_node = row
            col_node = max_row + 1 + col
            ds.union_by_size(row_node, col_node)
            stone_nodes.add(row_node)
            stone_nodes.add(col_node)

        # Count unique components among stone nodes
        unique_components = len(set(ds.find(node) for node in stone_nodes))

        return n - unique_components
```

Time and Space Complexity

Time Complexity: O(N × α(R+C))

- **N** = number of stones
- **R** = maximum row index, **C** = maximum column index
- **α(R+C)** = inverse Ackermann function (practically constant)
- **Processing**: O(N) to process all stones with DSU operations
- **Counting**: O(N) to count unique components
- **Overall**: Effectively **O(N)** since α is nearly constant

Space Complexity: O(R + C)

- **O(R + C)**: DSU parent and size arrays for all row and column nodes
- **O(N)**: Set to track unique parent components
- **Overall**: **O(R + C + N)**

Simplified Explanation

- **Time**: We visit each stone once to build connections, then count components. Like organizing a relay team - first we connect teammates, then count how many complete teams we have.
- **Space**: We need memory to track the "family tree" of connected rows and columns, plus remember which nodes actually contain stones.

Strongly Connected Graph

Definition

A **strongly connected graph** is a directed graph where there exists a path from every vertex to every other vertex. In other words, for any two vertices u and v, there must be a directed path from u to v AND a directed path from v to u.

Key Properties

- Every vertex can reach every other vertex
- The entire graph forms one single strongly connected component
- If you can traverse from any vertex and return to it while visiting all other vertices, the graph is strongly connected

Visual Example

```
Strongly Connected Graph:
    A → B
    ↑   ↓
    D ← C

Paths exist:
A→B→C→D→A (main cycle)
A→B, A→C, A→D, B→A, B→C, B→D, C→A, C→B, C→D, D→A, D→B, D→C


NOT Strongly Connected Graph:
    A → B → C
        ↓
        D

Missing paths: B→A, C→A, C→B, D→A, D→B, D→C (among others)
No way to return to earlier vertices
```

Strongly Connected Components

Definition

A **Strongly Connected Component (SCC)** is a maximal set of vertices in a directed graph such that:

1. Every vertex in the component can reach every other vertex in the same component
2. No vertex outside the component can be added while maintaining the strongly connected property

Key Characteristics

- **Maximal**: You cannot add any more vertices to an SCC without breaking the strongly connected property
- **Partition**: SCCs partition the vertices of a directed graph (every vertex belongs to exactly one SCC)
- **Single Vertex**: An isolated vertex (or vertex with no incoming/outgoing edges within its component) forms its own SCC

Visual Example

```
Graph with Multiple SCCs:

    SCC1: {A, B}     SCC2: {C}     SCC3: {D, E}

    A → B  →  C  →  D → E
    ↑   ↓           ↑   ↓
    └───┘           └───┘

    - SCC1: A and B form a cycle (A→B→A)
    - SCC2: C is alone (can't reach back to A,B and can't be reached from D,E)
    - SCC3: D and E form a cycle (D→E→D)
```

SCC in Directed vs Undirected Graphs

Directed Graphs

- SCCs are the main concept we work with
- Direction of edges matters critically
- A graph can have multiple SCCs
- Each SCC is a maximal strongly connected subgraph

```
Directed Graph Example:
    1 → 2 → 3
    ↑       ↓
    6 ← 5 ← 4

SCCs: {1,2,3,4,5,6} - This forms ONE SCC because:
1→2→3→4→5→6→1 (cycle exists)
```

Undirected Graphs

- In undirected graphs, the concept becomes **Connected Components**
- Every edge can be traversed in both directions
- If there's a path between two vertices, they're in the same connected component
- Each connected component in an undirected graph would be an SCC if we consider each undirected edge as two directed edges

```
Undirected Graph Example:
    1—2—3    7—8
    |   |    |
    6—5—4    9

Connected Components: {1,2,3,4,5,6} and {7,8,9}
If converted to directed: each edge becomes two directed edges
```

Brute Force Solution using Floyd-Warshall

Algorithm Overview

Floyd-Warshall finds shortest paths between all pairs of vertices. We can use it to check strong connectivity by verifying if every vertex can reach every other vertex.

Steps for SCC Detection using Floyd-Warshall

1. **Create Reachability Matrix**: Use Floyd-Warshall to find if vertex i can reach vertex j
2. **Check Strong Connectivity**: A graph is strongly connected if distance[i][j] != float('inf') for all pairs (i,j)
3. **Find SCCs**: Group vertices that can reach each other

Algorithm Implementation Logic

```
1. Initialize reachability matrix with direct edges
2. Apply Floyd-Warshall to find transitive closure
3. For each vertex pair (i,j):
   - If reachability[i][j] AND reachability[j][i] are not equal to infinity,
     then i and j are in the same SCC
4. Group vertices into SCCs based on mutual reachability
```

Time Complexity

- **Time**: $O(V^3)$ where V is number of vertices
- **Space**: $O(V^2)$ for reachability matrix
- **Not optimal** for large graphs (better algorithms exist like Tarjan's or Kosaraju's)

Graph Transpose and SCC Behavior

What is Graph Transpose?

The **transpose of a directed graph G** (denoted as G^T) is obtained by reversing the direction of all edges.

```
Original Graph G:        Transpose G^T:
    A → B                   A ← B
    ↓   ↓                   ↑   ↑
    D ← C                   D → C
```

Key Properties of Transpose

1. **Same SCCs**: G and G^T have exactly the same strongly connected components
2. **Reversed Edges**: Only edge directions are reversed, not the connectivity within SCCs
3. **SCC Structure Preserved**: The internal structure of each SCC remains the same

Why SCCs Remain Same in Transpose?

Proof by Logic:

- If vertices u and v are in the same SCC in G:
  - There exists path u → v in G
  - There exists path v → u in G
- In G^T:
  - Path u → v in G becomes v → u in G^T
  - Path v → u in G becomes u → v in G^T
  - So u and v can still reach each other in G^T
  - Therefore, they remain in the same SCC

Visual Example of Transpose Effect

```
Original Graph:
    A → B → C
    ↑       ↓
    E ← D ←┘

SCC: {A,B,C,D,E} - All vertices form one cycle: A→B→C→D→E→A

Transpose Graph:
    A ← B ← C
    ↓       ↑
    E → D →┘

SCC: {A,B,C,D,E} - Same vertices, cycle becomes: A→E→D→C→B→A

The SCC structure is preserved!
```

Applications of Transpose in SCC Algorithms

Kosaraju's Algorithm uses this property:

1. Perform DFS on original graph to get finish times
2. Create transpose graph
3. Perform DFS on transpose in decreasing order of finish times
4. Each DFS tree in step 3 gives one SCC

Visual Examples

Example 1: Simple SCC

```
Graph:
    1 → 2
    ↑   ↓
    4 ← 3

Analysis:
- Path 1→2→3→4→1 exists
```

```
- All vertices can reach all others
- This is ONE SCC: {1,2,3,4}
```

Example 2: Multiple SCCs

```
Graph:
    1 → 2 → 3 → 7 → 8
    ↑       ↓   ↑   ↓
    6 ← 5 ← 4   10← 9

Analysis:
SCC1: {1,2,3,4,5,6} - cycle: 1→2→3→4→5→6→1
SCC2: {7,8,9,10} - cycle: 7→8→9→10→7
```

Example 3: Linear Chain

```
Graph: 1 → 2 → 3 → 4 → 5

Analysis:
- No vertex can reach back to previous vertices
- Each vertex forms its own SCC
- SCCs: {1}, {2}, {3}, {4}, {5}
```

Key Takeaways

1. **Strongly Connected Graph**: Every vertex can reach every other vertex
2. **SCC**: Maximal strongly connected subgraph
3. **Direction Matters**: Only applies to directed graphs
4. **Floyd-Warshall Approach**: O(V^3) brute force method using reachability
5. **Transpose Property**: G and G^T have identical SCCs
6. **Practical Applications**: Social networks, web page ranking, circuit design

When to Use Each Approach

- **Floyd-Warshall**: Small graphs, when you also need shortest paths
- **Tarjan's Algorithm**: O(V+E), single pass, optimal for most cases
- **Kosaraju's Algorithm**: O(V+E), easy to understand, uses transpose property

Kosaraju's algorithm

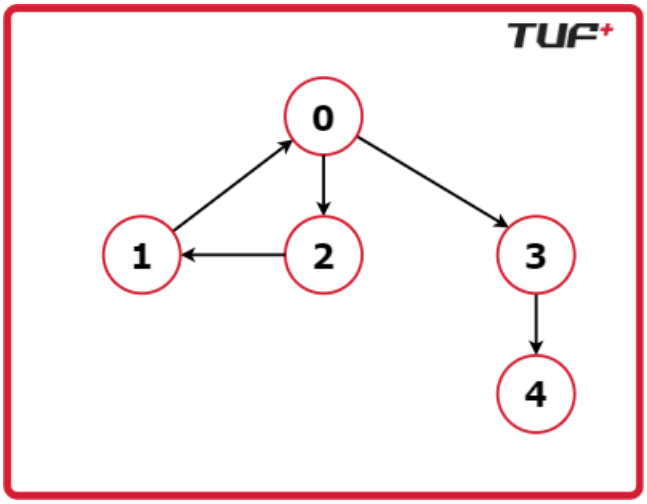
Problem Description

Given a directed graph with V vertices (numbered 0 to V-1) and its adjacency list, find the number of **Strongly Connected Components (SCCs)**. An SCC is a maximal set of vertices where every vertex is reachable from every other vertex within that set through directed edges.

Examples

Input

```
V = 5
Adj = [[2,3], [0], [1], [4], []]
```



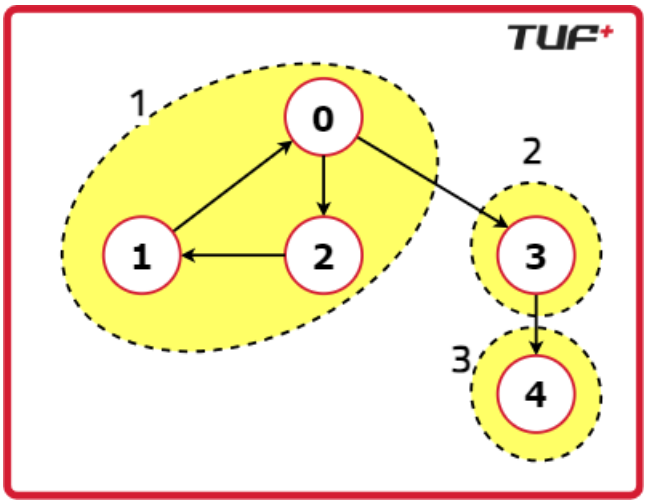
Output ``` 3 ```

Explanation

The graph has 3 strongly connected components: {0,1,2}, {3}, and {4}. Within the first component, you can reach any vertex from any other vertex.

Input

```
V = 8
Adj = [[1], [2], [0,3], [4], [5,7], [6], [4,7], []]
```



Output ``` 4 ```

Explanation

Four strongly connected components exist in this graph.

Solution

Use **Kosaraju's Algorithm**: a two-pass DFS approach that first determines finishing order, then performs DFS on the transposed graph to count SCCs.

Intuition

The Core Problem: Why Simple DFS Fails

```
Original Graph Visualization:
    SCC1          SCC2          SCC3
┌─────────┐   ┌─────────┐   ┌─────────┐
│ 0 → 1   │──→│ 3 → 4   │──→│ 6 → 7   │
│ ↑   ↓   │   │ ↑   ↓   │   │ ↑   ↓   │
│ 3 ← 2   │   │ 6 ← 5   │   │ 9 ← 8   │
└─────────┘   └─────────┘   └─────────┘

Problem: Starting DFS from node 0 visits ALL nodes across all SCCs!
We can't distinguish where one SCC ends and another begins.
```

The Brilliant Solution: Reverse the Graph!

```
Key Insight: If we reverse ALL edges, SCCs remain the same,
but we can no longer "leak" from one SCC to another!

Transposed (Reversed) Graph:
    SCC1          SCC2          SCC3
┌─────────┐   ┌─────────┐   ┌─────────┐
│ 0 ← 1   │←──│ 3 ← 4   │←──│ 6 ← 7   │
│ ↓   ↑   │   │ ↓   ↑   │   │ ↓   ↑   │
│ 3 → 2   │   │ 6 → 5   │   │ 9 → 8   │
└─────────┘   └─────────┘   └─────────┘
```

```
Now: Starting DFS from any node only visits its own SCC!
Number of DFS calls = Number of SCCs
```

But Wait! Which Node Should We Start From?

```
Problem: If we start DFS randomly on transposed graph,
we might start from the "wrong" SCC and miss others.

Example: Starting from SCC3 in transposed graph won't reach SCC1 or SCC2!

Solution: Start from the SCC that finishes LAST in original graph
(because it will be topologically "first" in the transpose)
```

The Magic of Finishing Times

```
Why Finishing Times Matter:

In original graph: DFS explores SCCs in topological order
SCC1 → SCC2 → SCC3 (following forward edges)

Finishing order: Nodes in SCC3 finish first, then SCC2, then SCC1
Stack after DFS: [SCC1_nodes..., SCC2_nodes..., SCC3_nodes...]
                  ↑ (top)                                  ↑ (bottom)
                Last to finish                        First to finish

In transposed graph: We need reverse topological order
Start with SCC1 (finished last), then SCC2, then SCC3

Stack gives us perfect order: Pop from top → SCC1, SCC2, SCC3 ✓
```

Step-by-Step Visual Walkthrough

```
Original Graph Example:
0 → 1 → 2 → 0 (SCC1: {0,1,2})
2 → 3 → 4     (SCC2: {3,4} - but 4↛3 missing, so separate SCCs)
4 → (end)     (SCC3: {4})

Step 1: First DFS on Original Graph
Start from 0: 0 → 1 → 2 → 3 → 4
Finishing order: 4 finishes first, then 3, then 2, then 1, then 0
Stack: [0, 1, 2, 3, 4] (top to bottom)

Step 2: Create Transposed Graph
0 ← 1 ← 2 ← 0 (still SCC1: {0,1,2})
2 ← 3 ← 4     (connections reversed)

Step 3: Second DFS on Transposed Graph
Pop 0 from stack: DFS from 0 visits {0,1,2} → SCC count = 1
Pop 1 from stack: Already visited → skip
Pop 2 from stack: Already visited → skip
Pop 3 from stack: DFS from 3 visits {3} → SCC count = 2
Pop 4 from stack: DFS from 4 visits {4} → SCC count = 3
```

```
Final Answer: 3 SCCs
```

Why This Algorithm is Genius

```
🔍 First DFS: "Survey the landscape"
   - Discovers the topological structure
   - Finishing times tell us the "exit order" from SCCs
   - Stack preserves reverse topological order

🔄 Transpose: "Reverse the flow"
   - Same SCCs, but no inter-SCC leakage
   - Each SCC becomes isolated island

🎯 Second DFS: "Count the islands"
   - Process SCCs in correct order (reverse topological)
   - Each DFS call = one complete SCC
   - No cross-contamination between SCCs

It's like exploring a river system:
1. First, follow downstream to map all tributaries
2. Reverse the water flow
3. Start from sources and count separate watersheds!
```

Approach Steps

1. **First DFS on Original Graph**:
    - Perform DFS to get finishing times of all vertices
    - Push vertices to stack when their DFS completes
    - Stack will contain vertices in reverse finishing order
2. **Create Transposed Graph**:
    - Reverse all edges in the original graph
    - If original had edge u→v, transpose has v→u
3. **Second DFS on Transposed Graph**:
    - Pop vertices from stack one by one
    - For each unvisited vertex, start new DFS
    - Each new DFS call represents one SCC
    - Count the number of DFS calls

Code

```python
class Solution:
    def kosaraju(self, V, adj):
        # Step 1: First DFS to get finishing order
        def dfs1(node, graph, visited, stack):
            visited[node] = True
            # Visit all adjacent nodes
```

```
            for neighbor in graph[node]:
                if not visited[neighbor]:
                    dfs1(neighbor, graph, visited, stack)
            # Add to stack when finishing (post-order)
            stack.append(node)

        # Step 2: Create transposed graph
        def create_transpose(V, adj):
            transpose = [[] for _ in range(V)]
            for u in range(V):
                for v in adj[u]:
                    transpose[v].append(u)  # Reverse the edge
            return transpose

        # Step 3: Second DFS on transposed graph
        def dfs2(node, graph, visited):
            visited[node] = True
            for neighbor in graph[node]:
                if not visited[neighbor]:
                    dfs2(neighbor, graph, visited)

        # Execute Step 1: First DFS on original graph
        visited = [False] * V
        stack = []

        for i in range(V):
            if not visited[i]:
                dfs1(i, adj, visited, stack)

        # Execute Step 2: Create transposed graph
        transpose_adj = create_transpose(V, adj)

        # Execute Step 3: Second DFS on transposed graph
        visited = [False] * V
        scc_count = 0

        # Process vertices in reverse finishing order
        while stack:
            node = stack.pop()
            if not visited[node]:
                dfs2(node, transpose_adj, visited)
                scc_count += 1  # Each DFS call = one SCC

        return scc_count

# Alternative implementation with cleaner structure
class SolutionOptimized:
    def kosaraju(self, V, adj):
        # Phase 1: Get finishing order
        visited = [False] * V
        finish_stack = []

        def dfs_finish_time(node):
            visited[node] = True
            for neighbor in adj[node]:
                if not visited[neighbor]:
                    dfs_finish_time(neighbor)
            finish_stack.append(node)  # Post-order addition

        # Run DFS on all unvisited nodes
        for i in range(V):
            if not visited[i]:
```

```
            dfs_finish_time(i)

    # Phase 2: Create transpose and count SCCs
    transpose = [[] for _ in range(V)]
    for u in range(V):
        for v in adj[u]:
            transpose[v].append(u)

    visited = [False] * V
    scc_count = 0

    def dfs_scc(node):
        visited[node] = True
        for neighbor in transpose[node]:
            if not visited[neighbor]:
                dfs_scc(neighbor)

    # Process in reverse finishing order
    while finish_stack:
        node = finish_stack.pop()
        if not visited[node]:
            dfs_scc(node)
            scc_count += 1

    return scc_count
```

Time and Space Complexity

Time Complexity: O(V + E)

- **First DFS**: O(V + E) to traverse all vertices and edges
- **Transpose creation**: O(E) to reverse all edges
- **Second DFS**: O(V + E) to traverse transposed graph
- **Overall**: O(V + E) - highly efficient!

Space Complexity: O(V + E)

- **Original graph storage**: O(V + E)
- **Transposed graph storage**: O(V + E)
- **Stack for finishing order**: O(V)
- **Visited arrays**: O(V)
- **Recursion stack**: O(V) in worst case

Simplified Explanation

- **Time**: We visit each vertex and edge exactly twice - once in each DFS phase. Like reading a book twice - first to understand the plot, second to appreciate the details.
- **Space**: We need to store both the original and reversed versions of the graph, plus some bookkeeping arrays. Like keeping both the original map and its mirror image.

Real-World Use Cases of Kosaraju's Algorithm

Kosaraju's Algorithm is used to find **Strongly Connected Components (SCCs)** in a directed graph. SCCs are subsets of nodes where each node is reachable from every other node in that subset. This concept has many **practical, real-world applications**:

1. Deadlock Detection in Operating Systems

- In resource allocation graphs (used in OS), processes and resources are represented as nodes, and edges denote requests/assignments.
- A **deadlock** occurs if a group of processes is waiting on each other in a cycle.
- Detecting SCCs helps identify such cycles efficiently.
- **Use case:** Preventing system freeze due to deadlocks.

2. Web Crawling and Search Engines

- The internet can be represented as a directed graph: web pages are nodes, and hyperlinks are edges.
- SCCs help identify clusters of websites that heavily reference each other.
- Search engines can:
  - Detect communities of interest.
  - Optimize crawling.
  - Improve ranking algorithms like **PageRank**.

3. Social Network Analysis

- In social media, users are nodes, and follow/connection relations are directed edges.
- SCCs can identify **mutually connected communities**:
  - A group of people all following each other.
  - Sub-networks with strong interaction loops.
- **Use case:** Community detection, targeted recommendations.

4. Recommendation Systems

- In e-commerce, nodes represent products and edges represent "people who bought X also bought Y."
- SCCs help discover **tightly linked products** that are frequently purchased together.
- **Use case:** Amazon, Flipkart suggesting bundles.

5. Compiler Design (Function Dependency Analysis)

- Functions in code can be represented as nodes, and function calls as edges.

- SCCs identify **mutually recursive functions**.
- Helps in:
  - Detecting infinite loops.
  - Optimizing function inlining.
- **Use case:** Efficient program analysis and optimization in compilers.

6. Electric Circuits / Network Analysis

- Circuits can be modeled as directed graphs.
- SCCs represent strongly connected sub-networks where current can flow both ways.
- **Use case:** Analyzing feedback loops in circuits.

7. Database Query Optimization

- In query dependency graphs, nodes represent queries/tables and edges represent dependency.
- SCCs can identify groups of queries that depend on each other.
- **Use case:** Optimizing execution order of SQL queries.

8. Transportation & Navigation Systems

- Cities (nodes) and one-way roads (edges) form a directed graph.
- SCCs can:
  - Detect regions where every city is reachable from every other city.
  - Identify isolated clusters.
- **Use case:** Planning routes, identifying bottlenecks.

9. Software Package Management

- In package managers (like npm, pip), packages are nodes and dependencies are edges.
- SCCs identify **circular dependencies**.
- **Use case:** Preventing infinite dependency loops.

10. Telecommunication Networks

- Routers/servers are nodes, and data transfer links are edges.
- SCCs show subnetworks where data can circulate in both directions.
- **Use case:** Ensuring fault tolerance, redundancy in network design.

🔑 Key Insight

Kosaraju's algorithm is **not just about theory**. Anywhere you have a **directed graph with cycles** (dependencies, flows, references), SCCs are essential to:

- Detect **cycles**.
- Identify **clusters**.
- Optimize **processing order**.
- Improve **robustness** of systems.

Kosaraju's Algorithm - Why DFS, Reverse, and DFS Again?

Kosaraju's Algorithm is used to find **Strongly Connected Components (SCCs)** in a directed graph.

A **Strongly Connected Component (SCC)** is a subgraph where every node is reachable from every other node in the same subgraph.

The algorithm works in three main steps:

1. **DFS to record finishing times**
2. **Reverse the graph**
3. **DFS in the order of decreasing finishing times**

Let's break this down with intuition, reasoning, and examples.

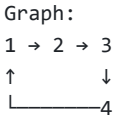
Step 1: DFS to Record Finishing Times

- We first do a **DFS traversal** on the original graph.
- The important part is the **finishing time** (when a node's DFS call is completed).
- We push nodes into a stack (or list) when they finish.

👉 Why?

- The finishing time ensures that we process **the right components first** when we move to the reversed graph.
- A node that finishes last is the one that does **not depend on any other node outside its component**.

📌 Example:



- If we run DFS:
    - Start at 1 → visit 2 → visit 3 → visit 4 → backtrack.

    - Finishing order: [2, 4, 3, 1]

- Here, node `1` finishes last, meaning it should be processed first in the reversed graph.

Step 2: Reverse the Graph

- We reverse the direction of every edge in the graph.
- If the original graph has `u → v`, the reversed graph has `v → u`.

👉 Why?

- In the original graph, DFS follows outward edges, so finishing times tell us which node leads strongly connected groups.
- In the reversed graph, DFS now explores **inward connections**, which allows us to capture the entire SCC.

📌 Example (reversing the above graph):

```
Original:    Reversed:
1 → 2        2 → 1
2 → 3        3 → 2
3 → 4        4 → 3
4 → 1        1 → 4
```

- Notice: The reversed graph still keeps the cycle, but DFS starting in the right order will now reveal SCCs directly.

Step 3: DFS in Decreasing Finishing Time Order

- Now we pop nodes from the stack (based on finishing time order from Step 1).
- For each unvisited node, perform DFS in the **reversed graph**.
- Every DFS call will give exactly **one SCC**.

👉 Why?

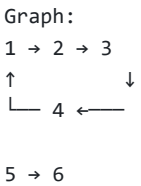
- Because the reversed graph ensures we can only reach nodes **inside the same SCC**.
- Processing nodes in the order of finishing time ensures that we discover larger SCCs first, and we never miss one.

📌 Continuing Example:

- Finishing order from Step 1: [2, 4, 3, 1]
- Start with `1` in reversed graph → DFS gives SCC = {1, 2, 3, 4}
- No other nodes left.
- Answer: one SCC = {1, 2, 3, 4}

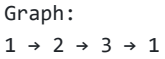
Different Cases

Case 1: Multiple SCCs



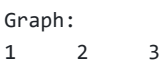
- SCCs: {1,2,3,4} and {5,6}
- Step 1: DFS gives finishing order: [2,3,4,1,6,5]
- Step 2: Reverse edges.
- Step 3: DFS on reversed graph in finishing order:
  - Start 5 → {5,6}
  - Start 1 → {1,2,3,4}
- Answer: Two SCCs found.

Case 2: Single SCC (all connected)



- All nodes form one SCC.
- Any DFS in reversed graph will capture all nodes at once.

Case 3: Completely Disconnected Graph



- Each node is its own SCC.
- Algorithm correctly identifies {1}, {2}, {3}.

Summary (Why the Three Steps?)

1. **DFS first (finishing times):**
   - Helps identify a correct processing order.
   - Nodes that finish last are “roots” of SCCs.
2. **Reverse graph:**

- Changes direction so that DFS explores entire SCCs instead of moving out of them.

3. **DFS again in finishing order:**
   - Ensures we capture SCCs one by one without overlap.

💡 Intuition:

- Step 1 tells us the order to consider nodes.
- Step 2 makes sure that we don't escape the SCC.
- Step 3 extracts SCCs cleanly.

Thus, Kosaraju's algorithm is essentially: **Finish times → Reverse → Extract SCCs.**

Bridges in graph / Targan's Algorithm

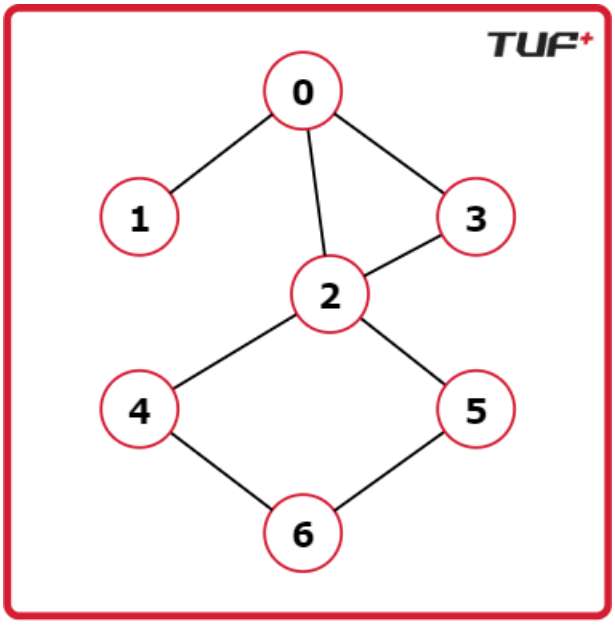
Problem Description

Given an undirected connected graph with V vertices and E edges, find all **bridges** in the graph. A bridge is an edge whose removal increases the number of connected components in the graph (i.e., disconnects the graph or makes some vertices unreachable from others).

Examples

Input

```
V = 4
edges = [[0,1], [1,2], [2,0], [1,3]]
```



Output

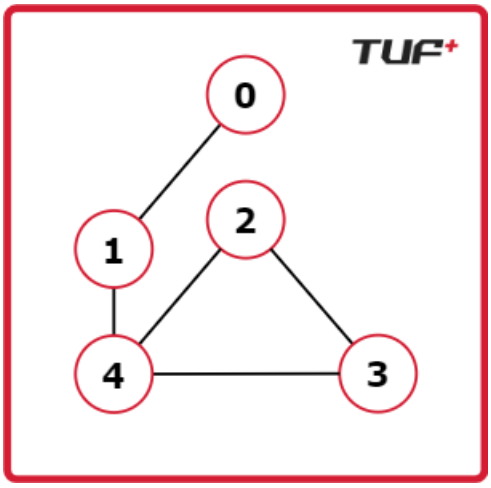
```
[[1,3]]
```

Explanation

The edge [1,3] is a bridge because removing it splits the graph into two components: {0,1,2} and {3}. All other edges are part of cycles and can be removed without disconnecting the graph.

Input

```
V = 3
edges = [[0,1], [1,2], [2,0]]
```



Output ``` [] ```

Explanation

This graph forms a triangle (3-cycle). No edge is a bridge because removing any edge still leaves the graph connected through the other two edges.

Solution

Use **Tarjan's Algorithm**: a DFS-based approach that tracks discovery times and lowest reachable times to identify bridges in O(V+E) time.

Intuition

The Core Insight: What Makes an Edge a Bridge?

```
Key Principle: An edge (u,v) is a bridge if and only if there's NO alternative path
between u and v that doesn't use this edge.

In other words: If removing edge (u,v) isolates v (and its subtree) from u,
then (u,v) is a bridge.
```

Visual Understanding: Bridge vs Non-Bridge

```
Case 1: Bridge Edge
    0---1---3
        |
        2

Edge (1,3) is a bridge because:
- Removing it isolates node 3
- No alternative path exists from {0,1,2} to {3}

Case 2: Non-Bridge Edge
    0---1---3
    |   |   |
    2---+---4

Edge (1,3) is NOT a bridge because:
- Alternative path exists: 1→0→2→4→3
- Removing (1,3) doesn't disconnect the graph
```

The Brilliant Tarjan's Algorithm Concept

Tarjan's algorithm tracks two crucial timestamps for each node during DFS:

```
1. Discovery Time (tin[v]): When node v was first visited in DFS
2. Low Time (low[v]): Earliest discovered node reachable from v's subtree

Bridge Condition: Edge (u,v) is a bridge if low[v] > tin[u]
```

Why This Condition Works: Deep Dive

```
Understanding low[v] > tin[u]:

If low[v] > tin[u], it means:
- Node v and its entire subtree cannot reach any ancestor of u
- The only way to reach v from u's ancestors is through edge (u,v)
- Therefore, removing (u,v) would disconnect v's subtree from the rest

Visual Example:
    DFS Tree with discovery times:

         0 (tin=1)
         |
         1 (tin=2)
         |
         2 (tin=3)
         |
         3 (tin=4)

If low[3] = 4 and tin[2] = 3:
- low[3] > tin[2] means node 3 cannot reach node 2 or earlier
- Edge (2,3) is a bridge!

If low[3] = 1 and tin[2] = 3:
```

```
- low[3] < tin[2] means node 3 can reach node 0 (discovered at time 1)
- There's a back edge! Edge (2,3) is NOT a bridge
```

Step-by-Step Algorithm Walkthrough

```
Example Graph:
    0---1---3
        |
        2

Step 1: Start DFS from node 0
timer = 1
tin[0] = low[0] = 1
visited[0] = true

Step 2: Visit neighbor 1
timer = 2
tin[1] = low[1] = 2
visited[1] = true

Step 3: From node 1, visit neighbor 2
timer = 3
tin[2] = low[2] = 3
visited[2] = true

Step 4: From node 2, check neighbors
- Neighbor 1 is parent, skip
- No other unvisited neighbors
- Return to node 1
- Update: low[1] = min(low[1], low[2]) = min(2, 3) = 2
- Check bridge condition: low[2] > tin[1]? → 3 > 2? → YES!
- Edge (1,2) is a bridge ✓

Step 5: From node 1, visit neighbor 3
timer = 4
tin[3] = low[3] = 4
visited[3] = true

Step 6: From node 3, check neighbors
- Neighbor 1 is parent, skip
- Return to node 1
- Update: low[1] = min(low[1], low[3]) = min(2, 4) = 2
- Check bridge condition: low[3] > tin[1]? → 4 > 2? → YES!
- Edge (1,3) is a bridge ✓

Final Result: Bridges = [(1,2), (1,3)]
```

Handling Back Edges: The Non-Bridge Case

```
Example with Back Edge (no bridges):
    0---1
    |   |
    2---+

DFS Traversal:
Step 1: Visit 0 (tin[0]=1, low[0]=1)
```

```
Step 2: Visit 1 (tin[1]=2, low[1]=2)
Step 3: Visit 2 (tin[2]=3, low[2]=3)
Step 4: From 2, find neighbor 0 (already visited, not parent)
        This is a BACK EDGE!
        Update: low[2] = min(low[2], tin[0]) = min(3, 1) = 1
Step 5: Return to 1: low[1] = min(low[1], low[2]) = min(2, 1) = 1
        Check: low[2] > tin[1]? → 1 > 2? → NO!
        Edge (1,2) is NOT a bridge
Step 6: Return to 0: low[0] = min(low[0], low[1]) = min(1, 1) = 1
        Check: low[1] > tin[0]? → 1 > 1? → NO!
        Edge (0,1) is NOT a bridge

Result: No bridges (cycle detected via back edge)
```

Why Tarjan's Algorithm is Genius

```
🎯 Single DFS Pass: Unlike naive approaches that try removing each edge,
   Tarjan's algorithm finds all bridges in just one DFS traversal!

🧠 Smart Timestamps: The low[] array cleverly tracks reachability without
   explicitly finding all paths.

⚡ Optimal Complexity: O(V+E) time - you can't do better than visiting
   each vertex and edge once!

🔗 Handles All Cases: Automatically distinguishes between:
   - Tree edges (potential bridges)
   - Back edges (cycle indicators, prevent bridges)
   - Cross edges (in directed graphs)

The algorithm essentially asks: "If I remove this edge, can the subtree
still reach the ancestors?" If not, it's a bridge!
```

Approach Steps

1. **Initialize data structures**:
    - Build adjacency list from edge list
    - Create arrays: `tin[]` (discovery time), `low[]` (lowest reachable time), `visited[]`
    - Initialize timer counter and result list
2. **Start DFS from each unvisited node**:
    - Set discovery and low time for current node
    - Mark as visited and increment timer
3. **For each neighbor of current node**:
    - Skip if neighbor is parent (avoid immediate backtrack)
    - If unvisited: recursively call DFS
    - If visited: update low time (back edge detected)
4. **Check bridge condition after each recursive call**:

- If `low[neighbor] > tin[current]`, edge is a bridge
- Add bridge to result list

5. **Update low time**:
   - `low[current] = min(low[current], low[neighbor])`

Code

```
class Solution:
    def criticalConnections(self, n, connections):
        # Step 1: Build adjacency list
        graph = [[] for _ in range(n)]
        for u, v in connections:
            graph[u].append(v)
            graph[v].append(u)

        # Step 2: Initialize arrays
        tin = [0] * n      # Discovery time
        low = [0] * n      # Lowest reachable time
        visited = [False] * n
        bridges = []
        timer = [0]        # Use list to make it mutable in nested function

        def dfs(node, parent):
            # Mark current node as visited
            visited[node] = True
            timer[0] += 1
            tin[node] = low[node] = timer[0]

            # Explore all neighbors
            for neighbor in graph[node]:
                # Skip parent to avoid immediate backtrack
                if neighbor == parent:
                    continue

                if not visited[neighbor]:
                    # Tree edge: recursively visit unvisited neighbor
                    dfs(neighbor, node)

                    # Update low time after returning from recursion
                    low[node] = min(low[node], low[neighbor])

                    # Check bridge condition
                    if low[neighbor] > tin[node]:
                        bridges.append([node, neighbor])

                else:
                    # Back edge: neighbor is visited and not parent
                    low[node] = min(low[node], tin[neighbor])

        # Step 3: Start DFS from each unvisited node (handles disconnected components)
        for i in range(n):
            if not visited[i]:
                dfs(i, -1)  # -1 indicates no parent for starting node

        return bridges

# Alternative implementation with cleaner structure
```

```python
class SolutionOptimized:
    def criticalConnections(self, n, connections):
        # Build adjacency list using sets for faster lookups
        graph = [set() for _ in range(n)]
        for u, v in connections:
            graph[u].add(v)
            graph[v].add(u)

        tin = [0] * n
        low = [0] * n
        visited = [False] * n
        bridges = []
        time = 0

        def tarjan_dfs(u, parent):
            nonlocal time
            visited[u] = True
            time += 1
            tin[u] = low[u] = time

            for v in graph[u]:
                if v == parent:
                    continue

                if visited[v]:
                    # Back edge
                    low[u] = min(low[u], tin[v])
                else:
                    # Tree edge
                    tarjan_dfs(v, u)
                    low[u] = min(low[u], low[v])

                    # Bridge condition
                    if low[v] > tin[u]:
                        bridges.append([u, v])

        # Handle all connected components
        for i in range(n):
            if not visited[i]:
                tarjan_dfs(i, -1)

        return bridges
```

Time and Space Complexity

Time Complexity: O(V + E)

- **Single DFS traversal**: Each vertex visited exactly once
- **Edge processing**: Each edge examined exactly twice (once from each endpoint)
- **Bridge checking**: Constant time per edge
- **Overall**: Optimal - cannot do better than visiting each vertex and edge

Space Complexity: O(V + E)

- **Adjacency list**: O(V + E) for storing the graph
- **Arrays**: O(V) for tin[], low[], visited[] arrays
- **Recursion stack**: O(V) in worst case (linear graph)

- **Result storage**: O(E) in worst case (when all edges are bridges)

Simplified Explanation

- **Time**: We visit each vertex once and examine each edge twice (from both endpoints). Like surveying a road network - you drive each road segment once in each direction to map it completely.
- **Space**: We need to store the road network map, plus some bookkeeping about when we first visited each intersection and the earliest intersection reachable from each location.

Articulation Point

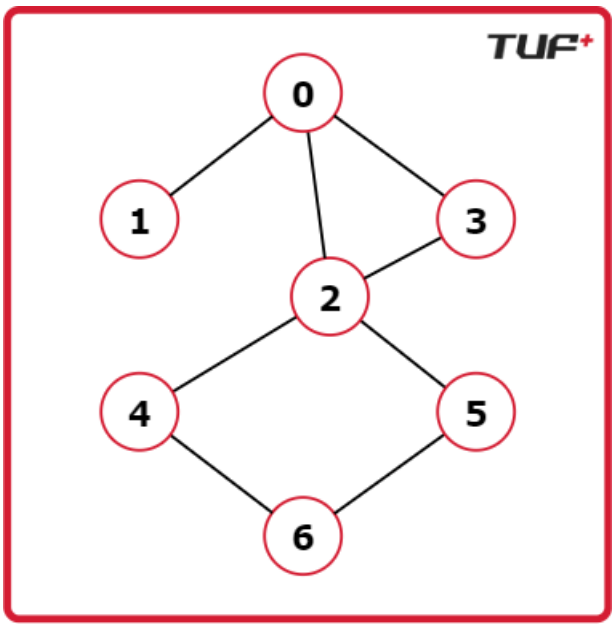
Problem Description

Given an undirected graph with V vertices and adjacency list, find all **articulation points** (or cut vertices). An articulation point is a vertex whose removal (along with all its incident edges) increases the number of connected components in the graph. Return vertices in ascending order, or [-1] if no articulation points exist.

Examples

Input

```
V = 7
adj = [[1,2,3], [0], [0,3,4,5], [2,0], [2,6], [5,2,6], [4,5]]
```



Output

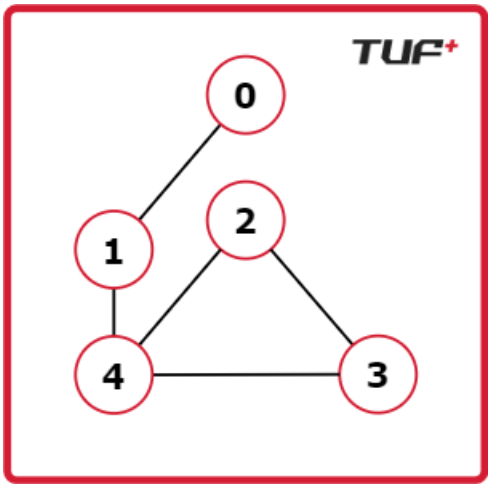
```
[0, 2]
```

Explanation

Removing vertex 0 disconnects vertices 1 from the rest. Removing vertex 2 separates vertices {4,5,6} from {0,1,3}. Both are critical for connectivity.

Input

```
V = 5
adj = [[1], [0,4], [3,4], [2,4], [1,2,3]]
```



Output

```
[1, 4]
```

Explanation

Removing vertex 1 isolates vertex 0. Removing vertex 4 separates the graph into multiple components. Both are articulation points.

Solution

Use **Tarjan's Algorithm for Articulation Points**: a DFS-based approach using discovery times and low values to identify critical vertices in O(V+E) time.

Intuition

Core Difference: Articulation Points vs Bridges

```
Bridge Problem: "Which EDGES are critical for connectivity?"
Articulation Point: "Which VERTICES are critical for connectivity?"

Key Challenge: When we remove a vertex, we also remove ALL its incident edges!
This makes the problem more complex than finding bridges.
```

What Makes a Vertex an Articulation Point?

```
A vertex u is an articulation point if:
1. It's the root of DFS tree AND has more than one child, OR
2. It's a non-root vertex AND removing it disconnects some part of the graph

Visual Example:
       1
      / \
     0   2---3
         |   |
         4---5

- Vertex 2 is articulation point: removing it separates {0,1} from {3,4,5}
- Vertex 0 is NOT articulation point: graph remains connected without it
- Vertex 1 is articulation point IF it's the DFS root with children 0 and 2
```

The Articulation Point Condition

```
For a non-root vertex u with child v in DFS tree:
u is an articulation point if low[v] >= tin[u]

Why this works:
- low[v] >= tin[u] means v's subtree cannot reach any ancestor of u
- Without vertex u, v's subtree would be disconnected from u's ancestors
- Therefore, u is critical for connectivity

Compare with Bridge Condition:
- Bridge: low[v] > tin[u] (strictly greater)
- Articulation Point: low[v] >= tin[u] (greater or equal)

The difference: Equal case means v can reach u but not u's ancestors!
```

Special Case: Root of DFS Tree

```
Root vertex is articulation point if and only if it has more than 1 child in DFS tree.

Why?
- If root has only 1 child: removing root leaves one connected component
- If root has 2+ children: removing root separates children into different components

Example:
    Root (0)
    /   |   \
   1    2    3

Removing root 0 creates 3 separate components: {1}, {2}, {3}
So root 0 is an articulation point.
```

Step-by-Step Algorithm Walkthrough

```
Example Graph:
    0---1---4
```

```
    |       |
    2---3---+

DFS Tree starting from 0:
    0 (root)
    |
    1
    |
    4
    |
    3
    |
    2 (back edge to 0)

Step-by-step execution:

1. Visit 0: tin[0] = low[0] = 1, children = 0

2. Visit 1: tin[1] = low[1] = 2, parent = 0

3. Visit 4: tin[4] = low[4] = 3, parent = 1

4. Visit 3: tin[3] = low[3] = 4, parent = 4

5. Visit 2: tin[2] = low[2] = 5, parent = 3

6. From 2, find neighbor 0 (visited, not parent):
   Back edge! low[2] = min(5, 1) = 1

7. Return to 3: low[3] = min(4, 1) = 1
   Check articulation: low[2] >= tin[3]? → 1 >= 4? → NO

8. Return to 4: low[4] = min(3, 1) = 1
   Check articulation: low[3] >= tin[4]? → 1 >= 3? → NO

9. Return to 1: low[1] = min(2, 1) = 1
   Check articulation: low[4] >= tin[1]? → 1 >= 2? → NO

10. Return to 0: low[0] = min(1, 1) = 1, children = 1
    Root check: children > 1? → 1 > 1? → NO

Result: No articulation points (graph is 2-edge-connected cycle)
```

Complex Example with Multiple Articulation Points

```
Graph:
    1---0---2---3
            |   |
            4---5

DFS Tree from 0:
    0 (root)
   /|\
  1 2 4
    | |
    3 5

Execution:
1. Visit 0: tin[0] = 1, children = 0
```

```
2. Visit 1: tin[1] = 2, check later
3. Visit 2: tin[2] = 3
4. Visit 3: tin[3] = 4
   Return to 2: low[2] = min(3,4) = 3
   Check: low[3] >= tin[2]? → 4 >= 3? → YES!
   2 is articulation point ✓
5. Visit 4: tin[4] = 5
6. Visit 5: tin[5] = 6
   Return to 4: low[4] = min(5,6) = 5
   Check: low[5] >= tin[4]? → 6 >= 5? → YES!
   4 is articulation point ✓
   Return to 2: low[2] = min(3,5) = 3
   Check: low[4] >= tin[2]? → 5 >= 3? → YES!
   2 is articulation point ✓ (confirmed again)
7. Return to 0: children = 3
   Root check: children > 1? → 3 > 1? → YES!
   0 is articulation point ✓

Result: Articulation points = [0, 2, 4]
```

Why the Algorithm Works: Deep Insight

```
The genius of Tarjan's algorithm for articulation points:

🎯 DFS Tree Structure: Creates a hierarchical view where parent-child
   relationships represent dependencies

🔍 Back Edge Detection: low[] values track alternative paths that could
   bypass the current vertex

⚡ Single Pass Efficiency: Computes both conditions (root and non-root)
   simultaneously during one DFS traversal

🧠 Critical Insight: The condition low[v] >= tin[u] captures exactly when
   removing u would isolate v's subtree

The algorithm transforms the complex question "What vertices disconnect
the graph?" into the simpler question "What vertices have no alternative
paths around them?" - which DFS can answer efficiently!
```

Approach Steps

1. **Initialize data structures**:
   - Arrays: `tin[]` (discovery time), `low[]` (lowest reachable time), `visited[]`
   - Timer counter and result list for articulation points
2. **Start DFS from each unvisited vertex**:
   - Set discovery and low time for current vertex
   - Mark as visited and increment timer
   - Track number of children for root case
3. **For each neighbor of current vertex**:

- Skip if neighbor is parent
- If unvisited: recursively call DFS, then check articulation condition
- If visited: update low time (back edge)

4. **Check articulation point conditions**:
   - **Non-root**: If `low[child] >= tin[current]`, current is articulation point
   - **Root**: If number of children > 1, root is articulation point
5. **Return sorted unique articulation points** (or [-1] if none exist)

Code

```
class Solution:
    def articulationPoints(self, n, adj):
        # Initialize data structures
        tin = [0] * n          # Discovery times
        low = [0] * n          # Lowest reachable times
        visited = [False] * n   # Visited markers
        ap_points = []         # Articulation points
        timer = [0]            # Timer (use list for mutability)

        def dfs(u, parent):
            # Mark current vertex as visited
            visited[u] = True
            timer[0] += 1
            tin[u] = low[u] = timer[0]

            children = 0  # Count children in DFS tree

            # Explore all adjacent vertices
            for v in adj[u]:
                if v == parent:
                    continue  # Skip parent to avoid immediate backtrack

                if not visited[v]:
                    # Tree edge: v is a child of u
                    children += 1
                    dfs(v, u)

                    # Update low value after returning from recursion
                    low[u] = min(low[u], low[v])

                    # Check articulation point condition for non-root
                    if parent != -1 and low[v] >= tin[u]:
                        ap_points.append(u)

                else:
                    # Back edge: update low value
                    low[u] = min(low[u], tin[v])

            # Check articulation point condition for root
            if parent == -1 and children > 1:
                ap_points.append(u)

        # Run DFS for all unvisited vertices (handle disconnected components)
        for i in range(n):
            if not visited[i]:
```

```
                dfs(i, -1)

        # Return sorted unique articulation points, or [-1] if none exist
        if ap_points:
            return sorted(set(ap_points))
        else:
            return [-1]

# Alternative implementation with cleaner structure
class SolutionOptimized:
    def articulationPoints(self, n, adj):
        tin = [0] * n
        low = [0] * n
        visited = [False] * n
        is_articulation = [False] * n  # Use boolean array to avoid duplicates
        time = 0

        def tarjan_ap(u, parent):
            nonlocal time
            visited[u] = True
            time += 1
            tin[u] = low[u] = time
            children = 0

            for v in adj[u]:
                if v == parent:
                    continue

                if visited[v]:
                    # Back edge
                    low[u] = min(low[u], tin[v])
                else:
                    # Tree edge
                    children += 1
                    tarjan_ap(v, u)
                    low[u] = min(low[u], low[v])

                    # Articulation point condition
                    if (parent == -1 and children > 1) or (parent != -1 and low[v] >= tin[u]):
                        is_articulation[u] = True

        # Process all connected components
        for i in range(n):
            if not visited[i]:
                tarjan_ap(i, -1)

        # Collect articulation points
        result = [i for i in range(n) if is_articulation[i]]
        return result if result else [-1]
```

Time and Space Complexity

Time Complexity: O(V + E)

- **Single DFS traversal**: Each vertex visited exactly once
- **Edge processing**: Each edge examined exactly twice (once from each endpoint)
- **Condition checking**: Constant time per vertex
- **Sorting**: O(V log V) for final result, but dominated by DFS time

Space Complexity: O(V + E)

- **Adjacency list**: O(V + E) to store the graph
- **Arrays**: O(V) for tin[], low[], visited[], and result arrays
- **Recursion stack**: O(V) in worst case (linear graph)
- **Overall**: O(V + E) dominated by graph storage

Simplified Explanation

- **Time**: We visit each vertex once and examine each edge twice. Like inspecting a building - you check each room once and each doorway from both sides to understand the structural dependencies.
- **Space**: We need to store the building blueprint plus some notes about when we first visited each room and the earliest room reachable from each location through the hallway network.