**Graph**

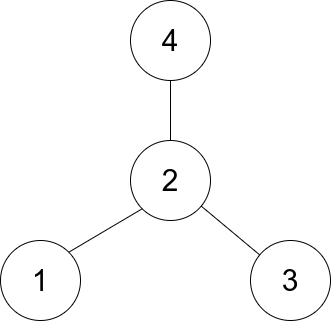
# **Very Easy:**

## **[Find Center of Star Graph](https://leetcode.com/problems/find-center-of-star-graph)**

There is an undirected star graph consisting of n nodes labeled from 1 to n. A star graph is a graph where there is one center node and exactly n - 1 edges that connect the center node with every other node.

You are given a 2D integer array edges where each edges[i] = [ui, vi] indicates that there is an edge between the nodes ui and vi. Return the center of the given star graph.

**Example 1:**



**Input**: edges = [[1,2],[2,3],[4,2]]

**Output**: 2

**Explanation**: As shown in the figure above, node 2 is connected to every other node, so 2 is the center.

**Example 2:**

**Input**: edges = [[1,2],[5,1],[1,3],[1,4]]

**Output**: 1

**Constraints:**

* 3 <= n <= 1e5
* edges.length == n - 1
* edges[i].length == 2
* 1 <= ui, vi <= n
* ui != vi
* The given edges represent a valid star graph.

## **[Find the Town Judge](https://leetcode.com/problems/find-the-town-judge/)**

In a town, there are n people labeled from 1 to n. There is a rumor that one of these people is secretly the town judge.

If the town judge exists, then:

1. The town judge trusts nobody.
2. Everybody (except for the town judge) trusts the town judge.
3. There is exactly one person that satisfies properties 1 and 2.

You are given an array trust where trust[i] = [ai, bi] representing that the person labeled ai trusts the person labeled bi. If a trust relationship does not exist in trust array, then such a trust relationship does not exist.

Return the label of the town judge if the town judge exists and can be identified, or return -1 otherwise.

**Example 1:**

**Input**: n = 2, trust = [[1,2]]

**Output**: 2

**Example 2:**

**Input**: n = 3, trust = [[1,3],[2,3]]

**Output**: 3

**Example 3:**

**Input**: n = 3, trust = [[1,3],[2,3],[3,1]]

**Output**: -1

**Constraints**:

* 1 <= n <= 1000
* 0 <= trust.length <= 1e4
* trust[i].length == 2
* All the pairs of trust are unique.
* ai != bi
* 1 <= ai, bi <= n

## **Flood Fill - [link](https://leetcode.com/problems/flood-fill/description/)**

You are given an image represented by an m x n grid of integers image, where image[i][j] represents the pixel value of the image. You are also given three integers sr, sc, and color. Your task is to perform a flood fill on the image starting from the pixel image[sr][sc].

To perform a flood fill:

Begin with the starting pixel and change its color to color.

Perform the same process for each pixel that is directly adjacent (pixels that share a side with the original pixel, either horizontally or vertically) and shares the same color as the starting pixel.

Keep repeating this process by checking neighboring pixels of the updated pixels and modifying their color if it matches the original color of the starting pixel.

The process stops when there are no more adjacent pixels of the original color to update.

Return the modified image after performing the flood fill.

**Example 1**:

**Input**: image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2

**Output**: [[2,2,2],[2,2,0],[2,0,1]]

**Explanation**:

From the center of the image with position (sr, sc) = (1, 1) (i.e., the red pixel), all pixels connected by a path of the same color as the starting pixel (i.e., the blue pixels) are colored with the new color.

**Note** the bottom corner is not colored 2, because it is not horizontally or vertically connected to the starting pixel.

**Example 2:**

**Input**: image = [[0,0,0],[0,0,0]], sr = 0, sc = 0, color = 0

**Output**: [[0,0,0],[0,0,0]]

**Explanation**:

The starting pixel is already colored with 0, which is the same as the target color. Therefore, no changes are made to the image.

**Constraints**:

* m == image.length
* n == image[i].length
* 1 <= m, n <= 50
* 0 <= image[i][j], color < 2^16
* 0 <= sr < m
* 0 <= sc < n

# **Easy**

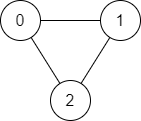
## **[Find if Path Exists in Graph](https://leetcode.com/problems/find-if-path-exists-in-graph/)**

There is a bi-directional graph with n vertices, where each vertex is labeled from 0 to n - 1 (inclusive). The edges in the graph are represented as a 2D integer array edges, where each edges[i] = [ui, vi] denotes a bi-directional edge between vertex ui and vertex vi. Every vertex pair is connected by at most one edge, and no vertex has an edge to itself.

You want to determine if there is a valid path that exists from vertex source to vertex destination.

Given edges and the integers n, source, and destination, return true if there is a valid path from source to destination, or false otherwise.

**Example 1:**



**Input**: n = 3, edges = [[0,1],[1,2],[2,0]], source = 0, destination = 2

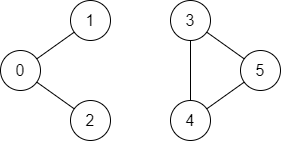
**Output**: true

**Explanation**: There are two paths from vertex 0 to vertex 2:

- 0 → 1 → 2

- 0 → 2

**Example 2:**



**Input**: n = 6, edges = [[0,1],[0,2],[3,5],[5,4],[4,3]], source = 0, destination = 5

**Output**: false

**Explanation**: There is no path from vertex 0 to vertex 5.

**Constraints:**

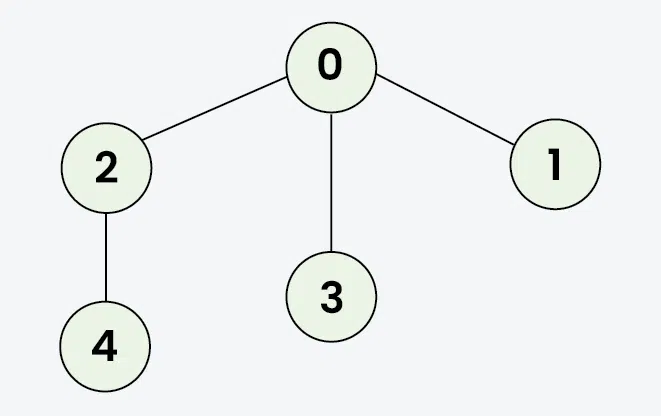
* 1 <= n <= 2 \* 1e5
* 0 <= edges.length <= 2 \* 1e5
* edges[i].length == 2
* 0 <= ui, vi <= n - 1
* ui != vi
* 0 <= source, destination <= n - 1
* There are no duplicate edges.
* There are no self edges.

## **BFS of graph [link](https://www.geeksforgeeks.org/problems/bfs-traversal-of-graph/1)**

Given a connected undirected graph represented by an adjacency list adj, which is a vector of vectors where each adj[i] represents the list of vertices connected to vertex i. Perform a Breadth First Traversal (BFS) starting from vertex 0, visiting vertices from left to right according to the adjacency list, and return a list containing the BFS traversal of the graph.

**Note**: Do traverse in the same order as they are in the adjacency list.

**Example** 1:



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

**Output**: [0, 2, 3, 1, 4]

**Explanation**: Starting from 0, the BFS traversal will follow these steps:

Visit 0 → Output: 0

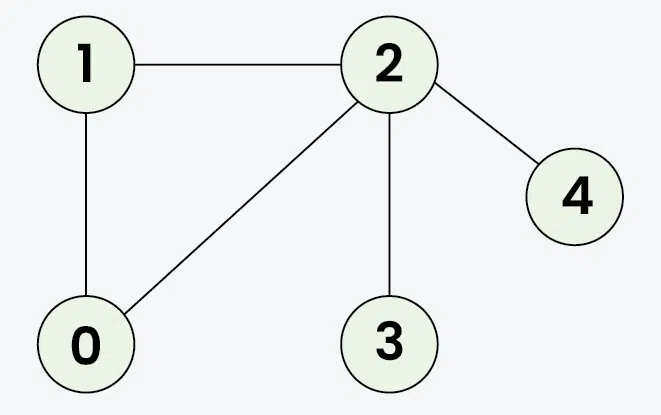
Visit 2 (first neighbor of 0) → Output: 0, 2

Visit 3 (next neighbor of 0) → Output: 0, 2, 3

Visit 1 (next neighbor of 0) → Output: 0, 2, 3,

Visit 4 (neighbor of 2) → Final Output: 0, 2, 3, 1, 4

**Example 2**



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

**Output**: [0, 1, 2, 3, 4]

**Explanation**: Starting from 0, the BFS traversal proceeds as follows:

Visit 0 → Output: 0

Visit 1 (the first neighbor of 0) → Output: 0, 1

Visit 2 (the next neighbor of 0) → Output: 0, 1, 2

Visit 3 (the first neighbor of 2 that hasn't been visited yet) → Output: 0, 1, 2, 3

Visit 4 (the next neighbor of 2) → Final Output: 0, 1, 2, 3, 4

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

Output: [0, 1, 2, 3, 4]

Explanation: Starting the BFS from vertex 0:

Visit vertex 0 → Output: [0]

Visit vertex 1 (first neighbor of 0) → Output: [0, 1]

Visit vertex 2 (first unvisited neighbor of 1) → Output: [0, 1, 2]

Visit vertex 3 (next neighbor of 1) → Output: [0, 1, 2, 3]

Visit vertex 4 (neighbor of 3) → Final Output: [0, 1, 2, 3, 4]

**Constraints**:

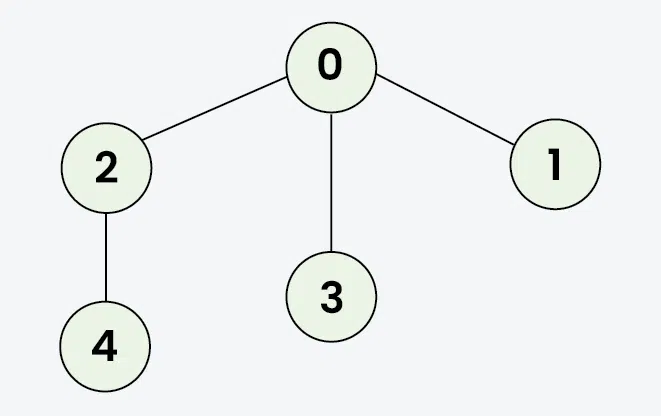
* 1 ≤ adj.size() ≤ 1e4
* 1 ≤ adj[i][j] ≤ 1e4

## **[DFS of Graph](https://www.geeksforgeeks.org/problems/depth-first-traversal-for-a-graph/1)**

Given a connected undirected graph represented by an adjacency list adj, which is a vector of vectors where each adj[i] represents the list of vertices connected to vertex i. Perform a Depth First Traversal (DFS) starting from vertex 0, visiting vertices from left to right as per the adjacency list, and return a list containing the DFS traversal of the graph.

Note: Do traverse in the same order as they are in the adjacency list.

**Example 1:**



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

Output: [0, 2, 4, 3, 1]

Explanation: Starting from 0, the DFS traversal proceeds as follows:

Visit 0 → Output: 0

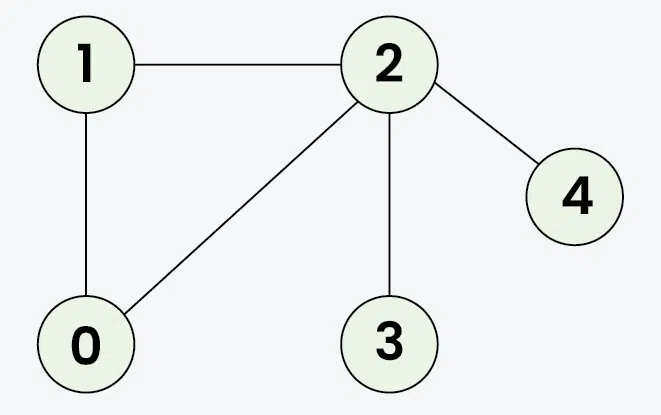
Visit 2 (the first neighbor of 0) → Output: 0, 2

Visit 4 (the first neighbor of 2) → Output: 0, 2, 4

Backtrack to 2, then backtrack to 0, and visit 3 → Output: 0, 2, 4, 3

Finally, backtrack to 0 and visit 1 → Final Output: 0, 2, 4, 3, 1

**Example 2:**



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

Output: [0, 1, 2, 3, 4]

Explanation: Starting from 0, the DFS traversal proceeds as follows:

Visit 0 → Output: 0

Visit 1 (the first neighbor of 0) → Output: 0, 1

Visit 2 (the first neighbor of 1) → Output: 0, 1, 2

Visit 3 (the first neighbor of 2) → Output: 0, 1, 2, 3

Backtrack to 2 and visit 4 → Final Output: 0, 1, 2, 3, 4

**Constraints:**

* 1 ≤ adj.size() ≤ 1e4
* 1 ≤ adj[i][j] ≤ 1e4

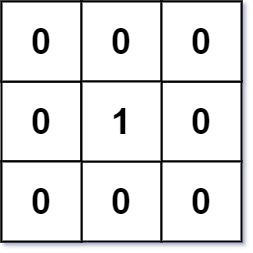
# **Medium**

## **[01 Matrix](https://leetcode.com/problems/01-matrix/)**

Given an m x n binary matrix mat, return the distance of the nearest 0 for each cell.

The distance between two adjacent cells is 1.

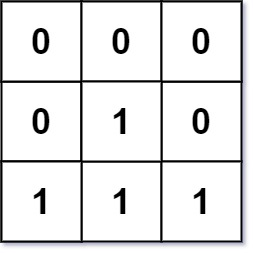
**Example** 1:



**Input**: mat = [[0,0,0],[0,1,0],[0,0,0]]

**Output**: [[0,0,0],[0,1,0],[0,0,0]]

**Example 2:**



**Input**: mat = [[0,0,0],[0,1,0],[1,1,1]]

**Output**: [[0,0,0],[0,1,0],[1,2,1]]

**Constraints:**

* m == mat.length
* n == mat[i].length
* 1 <= m, n <= 104
* 1 <= m \* n <= 104
* mat[i][j] is either 0 or 1.
* There is at least one 0 in mat.

## **[Course Schedule II](https://leetcode.com/problems/rotting-oranges/)**

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

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

Return the ordering of courses you should take to finish all courses. If there are many valid answers, return any of them. If it is impossible to finish all courses, return an empty array.

**Example 1:**

**Input**: numCourses = 2, prerequisites = [[1,0]]

**Output**: [0,1]

**Explanation**: There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is [0,1].

**Example** 2:

**Input**: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]

**Output**: [0,2,1,3]

**Explanation**: There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0.

So one correct course order is [0,1,2,3]. Another correct ordering is [0,2,1,3].

**Example** 3:

**Input**: numCourses = 1, prerequisites = []

**Output**: [0]

**Constraints:**

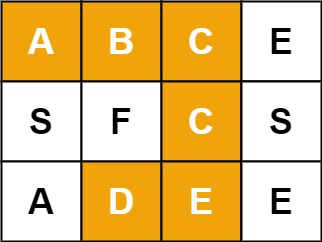
* 1 <= numCourses <= 2000
* 0 <= prerequisites.length <= numCourses \* (numCourses - 1)
* prerequisites[i].length == 2
* 0 <= ai, bi < numCourses
* ai != bi
* All the pairs [ai, bi] are distinct.

## **[Word Search](https://leetcode.com/problems/word-search/)**

Given an m x n grid of characters board and a string word, return true if word exists in the grid.

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

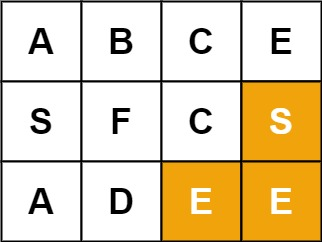
**Example** 1:



**Input**: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"

**Output**: true

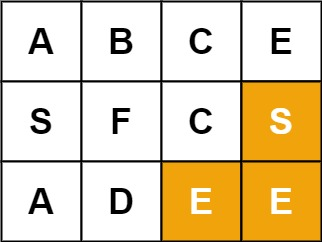
**Example** 2:



**Input**: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"

**Output**: true

**Example** 3:



**Input**: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB"

**Output**: false

**Constraints**:

* m == board.length
* n = board[i].length
* 1 <= m, n <= 6
* 1 <= word.length <= 15
* board and word consists of only lowercase and uppercase English letters.

**Follow up**: Could you use search pruning to make your solution faster with a larger board?

## **[Minimum Height Trees](https://leetcode.com/problems/minimum-height-trees/)**

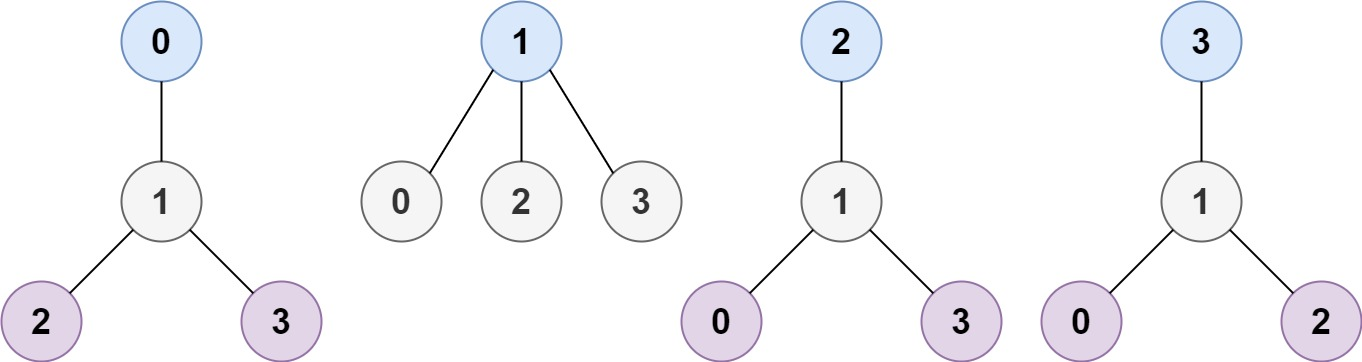
A tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, any connected graph without simple cycles is a tree.

Given a tree of n nodes labelled from 0 to n - 1, and an array of n - 1 edges where edges[i] = [ai, bi] indicates that there is an undirected edge between the two nodes ai and bi in the tree, you can choose any node of the tree as the root. When you select a node x as the root, the result tree has height h. Among all possible rooted trees, those with minimum height (i.e. min(h)) are called minimum height trees (MHTs).

Return a list of all MHTs' root labels. You can return the answer in any order.

The height of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

**Example 1:**

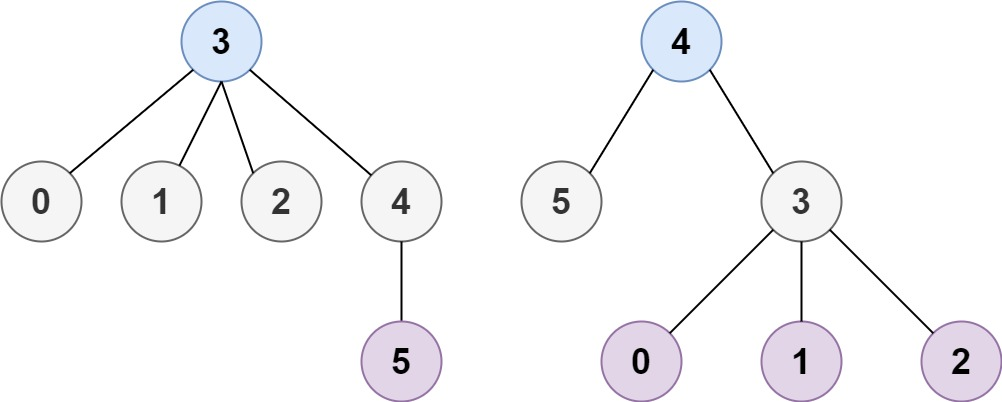


**Input**: n = 4, edges = [[1,0],[1,2],[1,3]]

**Output**: [1]

**Explanation**: As shown, the height of the tree is 1 when the root is the node with label 1 which is the only MHT.

**Example 2:**



**Input**: n = 6, edges = [[3,0],[3,1],[3,2],[3,4],[5,4]]

**Output**: [3,4]

**Constraints:**

* 1 <= n <= 2 \* 1e4
* edges.length == n - 1
* 0 <= ai, bi < n
* ai != bi
* All the pairs (ai, bi) are distinct.
* The given input is guaranteed to be a tree and there will be no repeated edges.

# **Hard**

## **[Accounts Merge](https://leetcode.com/problems/accounts-merge/description/)**

Given a list of accounts where each element accounts[i] is a list of strings, where the first element accounts[i][0] is a name, and the rest of the elements are emails representing emails of the account.

Now, we would like to merge these accounts. Two accounts definitely belong to the same person if there is some common email to both accounts. Note that even if two accounts have the same name, they may belong to different people as people could have the same name. A person can have any number of accounts initially, but all of their accounts definitely have the same name.

After merging the accounts, return the accounts in the following format: the first element of each account is the name, and the rest of the elements are emails in sorted order. The accounts themselves can be returned in any order.

**Example 1:**

**Input**: 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:**

The first and second John's are the same person as they have the common email "johnsmith@mail.com".

The third John and Mary are different people as none of their email addresses are used by other accounts.

We could return these lists in any order, for example the answer [['Mary', 'mary@mail.com'], ['John', 'johnnybravo@mail.com'],

['John', 'john00@mail.com', 'john\_newyork@mail.com', 'johnsmith@mail.com']] would still be accepted.

**Example 2:**

**Input**: accounts = [["Gabe","Gabe0@m.co","Gabe3@m.co","Gabe1@m.co"],["Kevin","Kevin3@m.co","Kevin5@m.co","Kevin0@m.co"],["Ethan","Ethan5@m.co","Ethan4@m.co","Ethan0@m.co"],["Hanzo","Hanzo3@m.co","Hanzo1@m.co","Hanzo0@m.co"],["Fern","Fern5@m.co","Fern1@m.co","Fern0@m.co"]]

**Output**: [["Ethan","Ethan0@m.co","Ethan4@m.co","Ethan5@m.co"],["Gabe","Gabe0@m.co","Gabe1@m.co","Gabe3@m.co"],["Hanzo","Hanzo0@m.co","Hanzo1@m.co","Hanzo3@m.co"],["Kevin","Kevin0@m.co","Kevin3@m.co","Kevin5@m.co"],["Fern","Fern0@m.co","Fern1@m.co","Fern5@m.co"]]

**Constraints:**

* 1 <= accounts.length <= 1000
* 2 <= accounts[i].length <= 10
* 1 <= accounts[i][j].length <= 30
* accounts[i][0] consists of English letters.
* accounts[i][j] (for j > 0) is a valid email.

## **[Rotting Oranges](https://leetcode.com/problems/rotting-oranges/)**

You are given an m x n grid where each cell can have one of three values:

0 representing an empty cell,

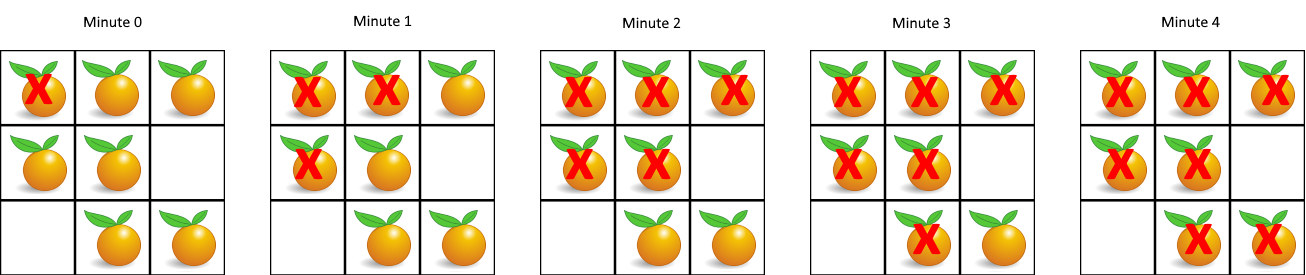
1 representing a fresh orange, or

2 representing a rotten orange.

Every minute, any fresh orange that is 4-directionally adjacent to a rotten orange becomes rotten.

Return the minimum number of minutes that must elapse until no cell has a fresh orange. If this is impossible, return -1.

**Example 1:**



**Input**: grid = [[2,1,1],[1,1,0],[0,1,1]]

**Output**: 4

**Example 2:**

**Input**: grid = [[2,1,1],[0,1,1],[1,0,1]]

**Output**: -1

**Explanation**: The orange in the bottom left corner (row 2, column 0) is never rotten, because rotting only happens 4-directionally.

**Example 3:**

**Input**: grid = [[0,2]]

**Output**: 0

**Explanation**: Since there are already no fresh oranges at minute 0, the answer is just 0.

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 10
* grid[i][j] is 0, 1, or 2.

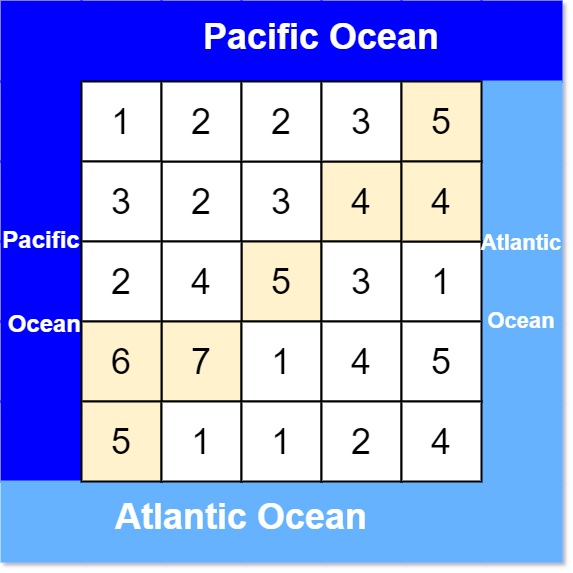
## **[Pacific Atlantic Water Flow](https://leetcode.com/problems/pacific-atlantic-water-flow/description/)**

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

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

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

Return a 2D list of grid coordinates result where result[i] = [ri, ci] denotes that rain water can flow from cell (ri, ci) to both the Pacific and Atlantic oceans.

**Example 1:**

**Input**: heights = [[1,2,2,3,5],[3,2,3,4,4],[2,4,5,3,1],[6,7,1,4,5],[5,1,1,2,4]]

**Output**: [[0,4],[1,3],[1,4],[2,2],[3,0],[3,1],[4,0]]

**Explanation**: The following cells can flow to the Pacific and Atlantic oceans, as shown below:

[0,4]: [0,4] -> Pacific Ocean

[0,4] -> Atlantic Ocean

[1,3]: [1,3] -> [0,3] -> Pacific Ocean

[1,3] -> [1,4] -> Atlantic Ocean

[1,4]: [1,4] -> [1,3] -> [0,3] -> Pacific Ocean

[1,4] -> Atlantic Ocean

[2,2]: [2,2] -> [1,2] -> [0,2] -> Pacific Ocean

[2,2] -> [2,3] -> [2,4] -> Atlantic Ocean

[3,0]: [3,0] -> Pacific Ocean

[3,0] -> [4,0] -> Atlantic Ocean

[3,1]: [3,1] -> [3,0] -> Pacific Ocean

[3,1] -> [4,1] -> Atlantic Ocean

[4,0]: [4,0] -> Pacific Ocean

[4,0] -> Atlantic Ocean

Note that there are other possible paths for these cells to flow to the Pacific and Atlantic oceans.

**Example 2:**

**Input**: heights = [[1]]

**Output**: [[0,0]]

**Explanation**: The water can flow from the only cell to the Pacific and Atlantic oceans.

**Constraints:**

* m == heights.length
* n == heights[r].length
* 1 <= m, n <= 200
* 0 <= heights[r][c] <= 105

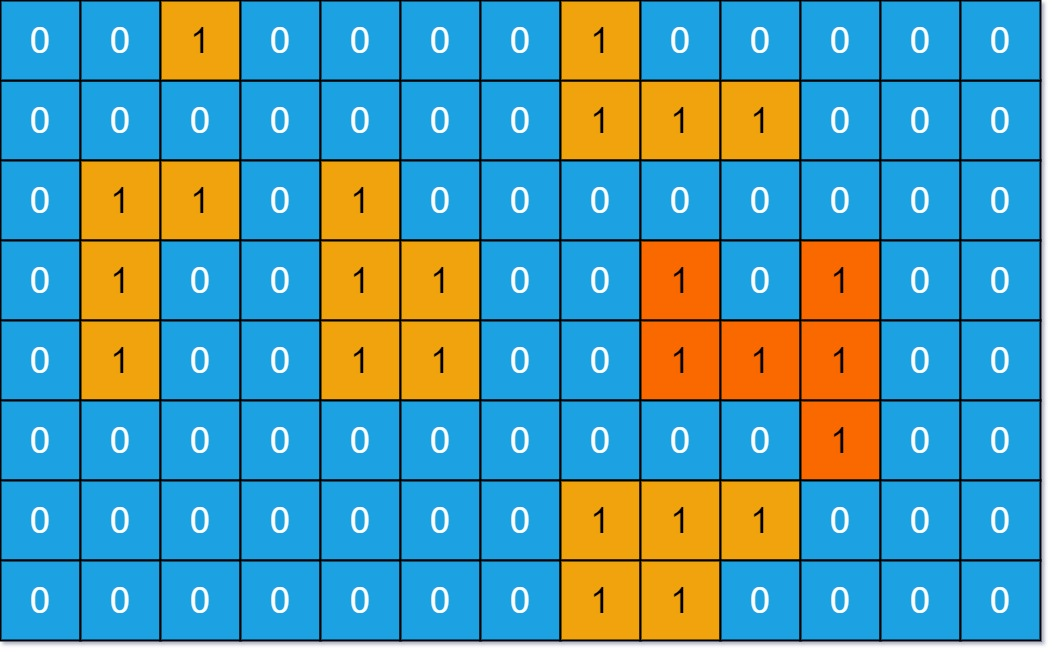
## **[Max Area of Island](https://leetcode.com/problems/max-area-of-island/)**

You are given an m x n binary matrix grid. An island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

The area of an island is the number of cells with a value 1 in the island.

Return the maximum area of an island in grid. If there is no island, return 0.

**Example 1:**



**Input**: grid = [[0,0,1,0,0,0,0,1,0,0,0,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,1,1,0,1,0,0,0,0,0,0,0,0],[0,1,0,0,1,1,0,0,1,0,1,0,0],[0,1,0,0,1,1,0,0,1,1,1,0,0],[0,0,0,0,0,0,0,0,0,0,1,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,0,0,0,0,0,0,1,1,0,0,0,0]]

**Output**: 6

**Explanation**: The answer is not 11, because the island must be connected 4-directionally.

**Example 2:**

**Input**: grid = [[0,0,0,0,0,0,0,0]]

**Output**: 0

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 50
* grid[i][j] is either 0 or 1.

## **[Evaluate Division](https://leetcode.com/problems/evaluate-division/)**

You are given an array of variable pairs equations and an array of real numbers values, where equations[i] = [Ai, Bi] and values[i] represent the equation Ai / Bi = values[i]. Each Ai or Bi is a string that represents a single variable.

You are also given some queries, where queries[j] = [Cj, Dj] represents the jth query where you must find the answer for Cj / Dj = ?.

Return the answers to all queries. If a single answer cannot be determined, return -1.0.

Note: The input is always valid. You may assume that evaluating the queries will not result in division by zero and that there is no contradiction.

Note: The variables that do not occur in the list of equations are undefined, so the answer cannot be determined for them.

**Example 1:**

**Input**: equations = [["a","b"],["b","c"]], values = [2.0,3.0], queries = [["a","c"],["b","a"],["a","e"],["a","a"],["x","x"]]

**Output**: [6.00000,0.50000,-1.00000,1.00000,-1.00000]

**Explanation:**

Given: a / b = 2.0, b / c = 3.0

queries are: a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ?

return: [6.0, 0.5, -1.0, 1.0, -1.0 ]

note: x is undefined => -1.0

**Example 2:**

**Input**: equations = [["a","b"],["b","c"],["bc","cd"]], values = [1.5,2.5,5.0], queries = [["a","c"],["c","b"],["bc","cd"],["cd","bc"]]

**Output**: [3.75000,0.40000,5.00000,0.20000]

**Example 3:**

**Input**: equations = [["a","b"]], values = [0.5], queries = [["a","b"],["b","a"],["a","c"],["x","y"]]

**Output**: [0.50000,2.00000,-1.00000,-1.00000]

**Constraints:**

* 1 <= equations.length <= 20
* equations[i].length == 2
* 1 <= Ai.length, Bi.length <= 5
* values.length == equations.length
* 0.0 < values[i] <= 20.0
* 1 <= queries.length <= 20
* queries[i].length == 2
* 1 <= Cj.length, Dj.length <= 5
* Ai, Bi, Cj, Dj consist of lower case English letters and digits.

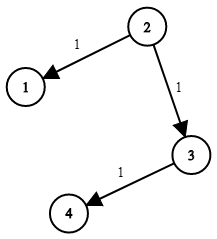
# **Very Hard**

## **[Network Delay Time](https://leetcode.com/problems/network-delay-time/description/)**

You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times as directed edges times[i] = (ui, vi, wi), where ui is the source node, vi is the target node, and wi is the time it takes for a signal to travel from source to target.

We will send a signal from a given node k. Return the minimum time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1.

**Example 1:**



**Input**: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2

**Output**: 2

**Example 2:**

**Input**: times = [[1,2,1]], n = 2, k = 1

**Output**: 1

**Example 3:**

**Input**: times = [[1,2,1]], n = 2, k = 2

**Output**: -1

**Constraints:**

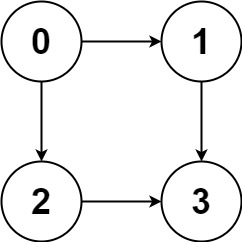
* 1 <= k <= n <= 100
* 1 <= times.length <= 6000
* times[i].length == 3
* 1 <= ui, vi <= n
* ui != vi
* 0 <= wi <= 100
* All the pairs (ui, vi) are unique. (i.e., no multiple edges.)

## **[All Paths From Source to Target](https://leetcode.com/problems/all-paths-from-source-to-target/description/)**

Given a directed acyclic graph (DAG) of n nodes labeled from 0 to n - 1, find all possible paths from node 0 to node n - 1 and return them in any order.

The graph is given as follows: graph[i] is a list of all nodes you can visit from node i (i.e., there is a directed edge from node i to node graph[i][j]).

**Example 1:**

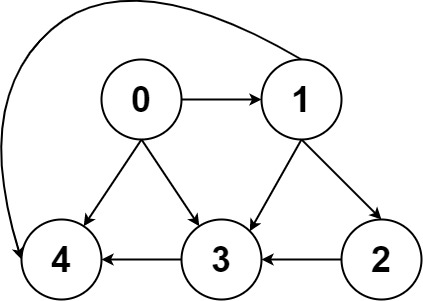


**Input**: graph = [[1,2],[3],[3],[]]

**Output**: [[0,1,3],[0,2,3]]

**Explanation**: There are two paths: 0 -> 1 -> 3 and 0 -> 2 -> 3.

**Example 2:**



**Input**: graph = [[4,3,1],[3,2,4],[3],[4],[]]

**Output**: [[0,4],[0,3,4],[0,1,3,4],[0,1,2,3,4],[0,1,4]]

**Constraints:**

* n == graph.length
* 2 <= n <= 15
* 0 <= graph[i][j] < n
* graph[i][j] != i (i.e., there will be no self-loops).
* All the elements of graph[i] are unique.
* The input graph is guaranteed to be a DAG.

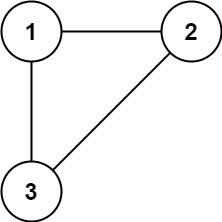
## **[Redundant Connection](https://leetcode.com/problems/redundant-connection/description/)**

In this problem, a tree is an undirected graph that is connected and has no cycles.

You are given a graph that started as a tree with n nodes labeled from 1 to n, with one additional edge added. The added edge has two different vertices chosen from 1 to n, and was not an edge that already existed. The graph is represented as an array edges of length n where edges[i] = [ai, bi] indicates that there is an edge between nodes ai and bi in the graph.

Return an edge that can be removed so that the resulting graph is a tree of n nodes. If there are multiple answers, return the answer that occurs last in the input.

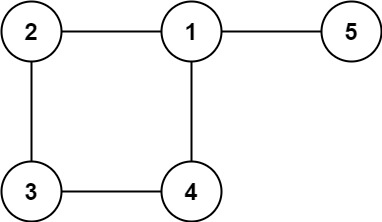
**Example 1:**



**Input**: edges = [[1,2],[1,3],[2,3]]

**Output**: [2,3]

**Example 2:**



**Input**: edges = [[1,2],[2,3],[3,4],[1,4],[1,5]]

**Output**: [1,4]

**Constraints**:

* n == edges.length
* 3 <= n <= 1000
* edges[i].length == 2
* 1 <= ai < bi <= edges.length
* ai != bi
* There are no repeated edges.
* The given graph is connected.

## **[Shortest Path in Binary Matrix](https://leetcode.com/problems/shortest-path-in-binary-matrix/)**

Shortest Path in Binary MatrixGiven an n x n binary matrix grid, return the length of the shortest clear path in the matrix. If there is no clear path, return -1.

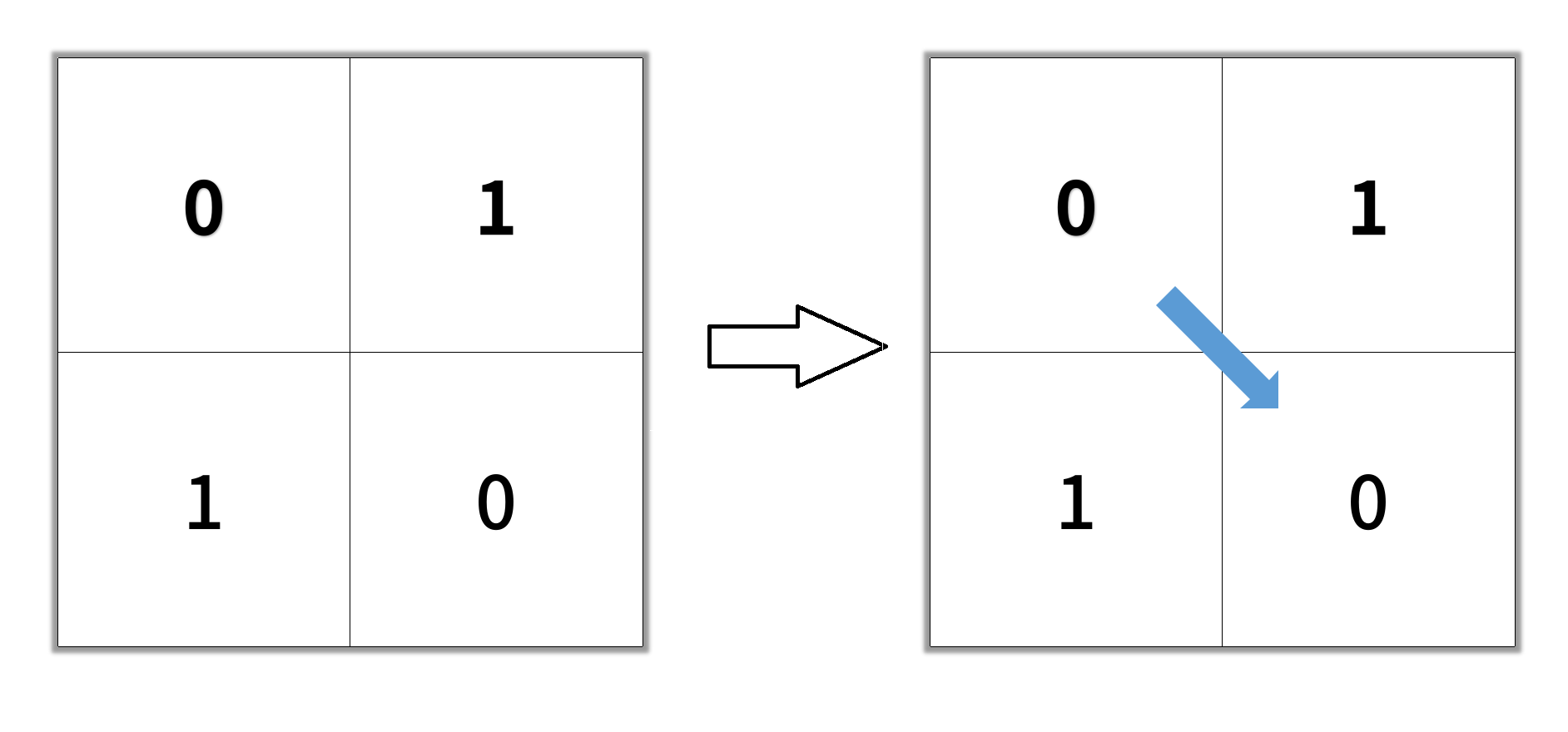
A clear path in a binary matrix is a path from the top-left cell (i.e., (0, 0)) to the bottom-right cell (i.e., (n - 1, n - 1)) such that:

All the visited cells of the path are 0.

All the adjacent cells of the path are 8-directionally connected (i.e., they are different and they share an edge or a corner).

The length of a clear path is the number of visited cells of this path.

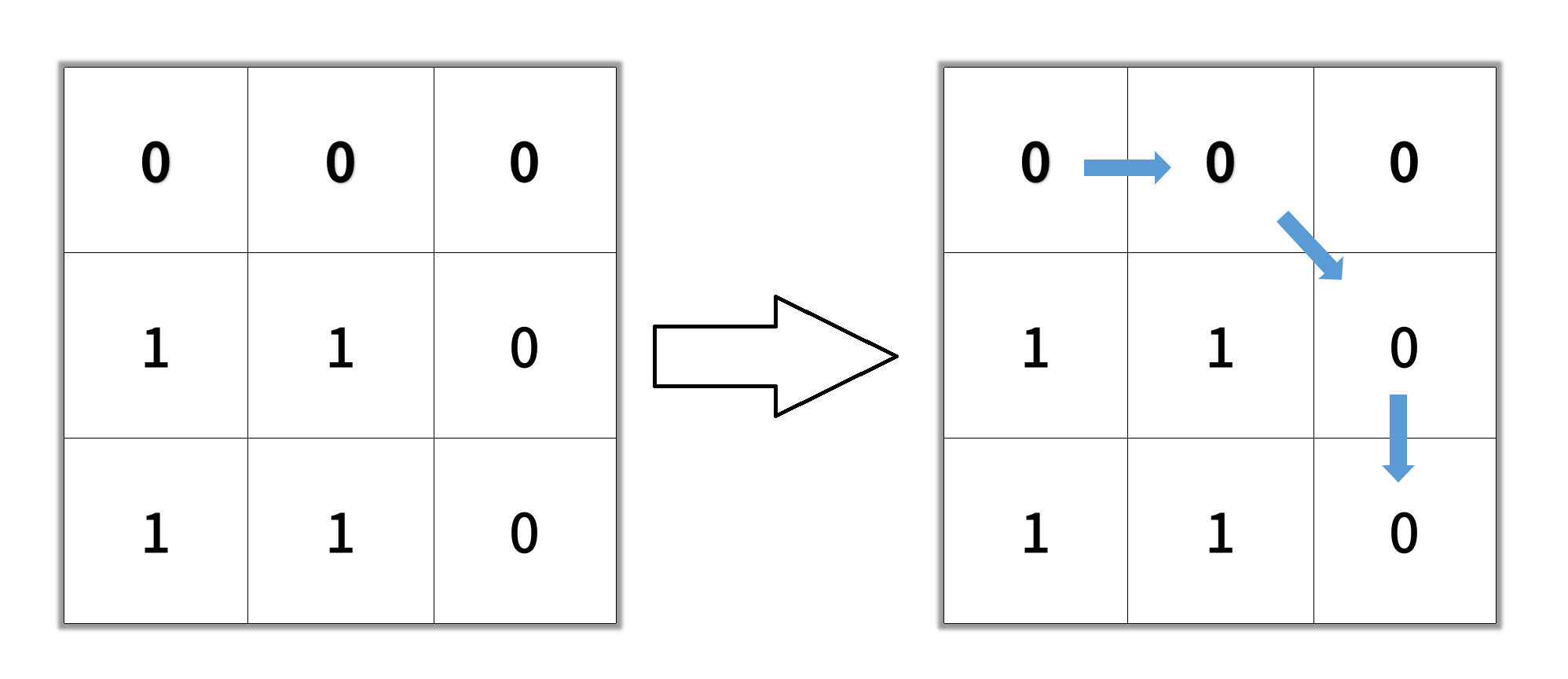
**Example 1:**



**Input**: grid = [[0,1],[1,0]]

**Output**: 2

**Example 2:**



**Input**: grid = [[0,0,0],[1,1,0],[1,1,0]]

**Output**: 4

**Example 3:**

**Input**: grid = [[1,0,0],[1,1,0],[1,1,0]]

**Output**: -1

**Constraints:**

* n == grid.length
* n == grid[i].length
* 1 <= n <= 100
* grid[i][j] is 0 or 1

## **Remove Max Number of Edges to Keep Graph Fully Traversable - [link](https://leetcode.com/problems/remove-max-number-of-edges-to-keep-graph-fully-traversable/description)**

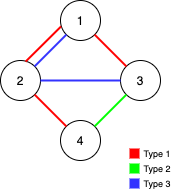
Alice and Bob have an undirected graph of n nodes and three types of edges:

* Type 1: Can be traversed by Alice only.
* Type 2: Can be traversed by Bob only.
* Type 3: Can be traversed by both Alice and Bob.

Given an array edges where edges[i] = [typei, ui, vi] represents a bidirectional edge of type typei between nodes ui and vi, find the maximum number of edges you can remove so that after removing the edges, the graph can still be fully traversed by both Alice and Bob. The graph is fully traversed by Alice and Bob if starting from any node, they can reach all other nodes.

Return the maximum number of edges you can remove, or return -1 if Alice and Bob cannot fully traverse the graph.

**Example 1:**

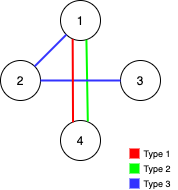


**Input**: n = 4, edges = [[3,1,2],[3,2,3],[1,1,3],[1,2,4],[1,1,2],[2,3,4]]

**Output**: 2

**Explanation**: If we remove the 2 edges [1,1,2] and [1,1,3]. The graph will still be fully traversable by Alice and Bob. Removing any additional edge will not make it so. So the maximum number of edges we can remove is 2.

**Example 2:**

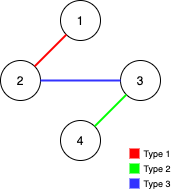


**Input**: n = 4, edges = [[3,1,2],[3,2,3],[1,1,4],[2,1,4]]

**Output**: 0

**Explanation**: Notice that removing any edge will not make the graph fully traversable by Alice and Bob.

**Example 3:**



**Input**: n = 4, edges = [[3,2,3],[1,1,2],[2,3,4]]

**Output**: -1

**Explanation**: In the current graph, Alice cannot reach node 4 from the other nodes. Likewise, Bob cannot reach 1. Therefore it's impossible to make the graph fully traversable.

**Constraints:**

* 1 <= n <= 105
* 1 <= edges.length <= min(105, 3 \* n \* (n - 1) / 2)
* edges[i].length == 3
* 1 <= typei <= 3
* 1 <= ui < vi <= n
* All tuples (typei, ui, vi) are distinct.