126. Word Ladder II

A **transformation sequence** from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord -> s1 -> s2 -> ... -> sk` such that:

- Every adjacent pair of words differs by a single letter.
- Every `si` for `1 <= i <= k` is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- `sk == endWord`

Given two words, `beginWord` and `endWord`, and a dictionary `wordList`, return *all the **shortest transformation sequences** from* `beginWord` *to* `endWord`*, or an empty list if no such sequence exists. Each sequence should be returned as a list of the words* `[beginWord, s1, s2, ..., sk]`.

Example 1:

```
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log",
Output: [["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]
Explanation: There are 2 shortest transformation sequences:
"hit" -> "hot" -> "dot" -> "dog" -> "cog"
"hit" -> "hot" -> "lot" -> "log" -> "cog"
```

Example 2:

```
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
Output: []
Explanation: The endWord "cog" is not in wordList, therefore there is no valid transf
```

Constraints:

- `1 <= beginWord.length <= 5`
- `endWord.length == beginWord.length`
- `1 <= wordList.length <= 500`
- `wordList[i].length == beginWord.length`
- `beginWord`, `endWord`, and `wordList[i]` consist of lowercase English letters.
- `beginWord != endWord`
- All the words in `wordList` are **unique**.
- The **sum** of all shortest transformation sequences does not exceed `10^5`.

```
from collections import defaultdict, deque
from typing import List

class Solution:
    def findLadders(self, beginWord: str, endWord: str, wordList: List[str]) -> Lis
t[List[str]]:
        wordSet = set(wordList)
        if endWord not in wordSet:
            return []

        # Build graph using BFS for shortest paths
        graph = defaultdict(list)
        found = False
        queue = deque([beginWord])
        visited = set([beginWord])

        while queue and not found:
            local_visited = set()
            for _ in range(len(queue)):
                word = queue.popleft()
                for i in range(len(word)):
                    for c in 'abcdefghijklmnopqrstuvwxyz':
                        if word[i] == c:
                            continue
                        newWord = word[:i] + c + word[i+1:]
                        if newWord in wordSet:
                            if newWord not in visited:
                                if newWord == endWord:
                                    found = True
                                if newWord not in local_visited:
                                    queue.append(newWord)
                                    local_visited.add(newWord)
                                graph[word].append(newWord)
            visited.update(local_visited)

        # Backtrack all paths using DFS
        res = []
        path = [beginWord]

        def dfs(current):
            if current == endWord:
                res.append(path[:])
                return
            for neighbor in graph[current]:
                path.append(neighbor)
                dfs(neighbor)
                path.pop()
```

```
        dfs(beginWord)
        return res
```

---

127. Word Ladder

A **transformation sequence** from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord -> s1 -> s2 -> ... -> sk` such that:

- Every adjacent pair of words differs by a single letter.
- Every `si` for `1 <= i <= k` is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- `sk == endWord`

Given two words, `beginWord` and `endWord`, and a dictionary `wordList`, return *the* ***number of words*** *in the* ***shortest transformation sequence*** *from* `beginWord` *to* `endWord`, *or* `0` *if no such sequence exists.*

Example 1:

```
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log",
Output: 5
Explanation: One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog"
```

Example 2:

```
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
Output: 0
Explanation: The endWord "cog" is not in wordList, therefore there is no valid transf
```

Constraints:

- `1 <= beginWord.length <= 10`
- `endWord.length == beginWord.length`
- `1 <= wordList.length <= 5000`
- `wordList[i].length == beginWord.length`
- `beginWord`, `endWord`, and `wordList[i]` consist of lowercase English letters.
- `beginWord != endWord`
- All the words in `wordList` are **unique**.

```
from collections import deque
class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) -> in
t:
        que = deque([[beginWord,1]])
        hashmap = set(wordList)
        if endWord not in hashmap:
            return 0
        while que :
            cur_word,step = que.popleft()
            if cur_word == endWord:return step
            for i in range(len(cur_word)):
                for char in range(97,123):
                    new_word = cur_word[:i] + chr(char)+cur_word[i+1:]
                    if new_word in hashmap:
                        hashmap.remove(new_word)
                        que.append([new_word,step+1])
        return 0
```

130. Surrounded Regions

You are given an `m x n` matrix `board` containing **letters** `'X'` and `'O'`, **capture regions** that are **surrounded**:

- **Connect**: A cell is connected to adjacent cells horizontally or vertically.
- **Region**: To form a region **connect every** `'O'` cell.
- **Surround**: The region is surrounded with `'X'` cells if you can **connect the region** with `'X'` cells and none of the region cells are on the edge of the `board`.

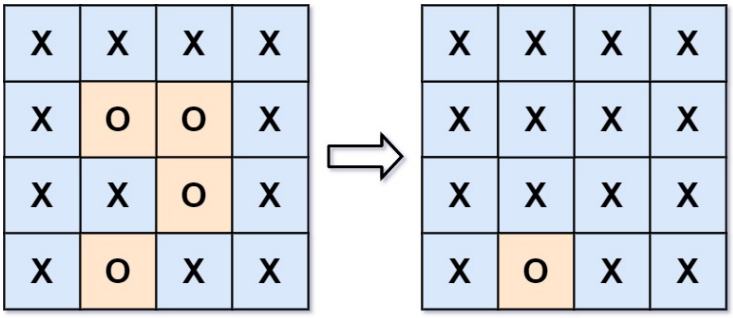
To capture a **surrounded region**, replace all `'O'`s with `'X'`s **in-place** within the original board. You do not need to return anything.

Example 1:

Input: board = [["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:



In the above diagram, the bottom region is not captured because it is on the edge of the board and cannot be surrounded.

Example 2:

Input: board = [["X"]]

Output: [["X"]]

Constraints:

- `m == board.length`
- `n == board[i].length`
- `1 <= m, n <= 200`
- `board[i][j]` is `'X'` or `'O'`.

```
class Solution:
    def solve(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """

        if not board:
            return

        m, n = len(board), len(board[0])
        dirs = [(-1,0), (1,0), (0,-1), (0,1)]

        def dfs(i, j):
            if i < 0 or i >= m or j < 0 or j >= n or board[i][j] != 'O':
                return
            board[i][j] = 'E'  # Mark as escaped
            for dx, dy in dirs:
                dfs(i + dx, j + dy)

        # Step 1: Mark border-connected 'O's
        for i in range(m):
            if board[i][0] == 'O':
                dfs(i, 0)
            if board[i][n-1] == 'O':
                dfs(i, n-1)

        for j in range(n):
            if board[0][j] == 'O':
                dfs(0, j)
            if board[m-1][j] == 'O':
                dfs(m-1, j)

        # Step 2: Flip and revert
        for i in range(m):
            for j in range(n):
                if board[i][j] == 'O':
                    board[i][j] = 'X'
                elif board[i][j] == 'E':
                    board[i][j] = 'O'
```

207. Course Schedule

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] = [a_i, b_i]` indicates that you **must** take course `b_i` first if you want to take course `a_i`.

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

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

Example 1:

```
Input: numCourses = 2, prerequisites = [[1,0]]
Output: true
Explanation: There are a total of 2 courses to take.
To take course 1 you should have finished course 0. So it is possible.
```

Example 2:

```
Input: numCourses = 2, prerequisites = [[1,0],[0,1]]
Output: false
Explanation: There are a total of 2 courses to take.
To take course 1 you should have finished course 0, and to take course 0 you should a
```

Constraints:

- `1 <= numCourses <= 2000`
- `0 <= prerequisites.length <= 5000`
- `prerequisites[i].length == 2`
- `0 <= a_i, b_i < numCourses`
- All the pairs prerequisites[i] are **unique**.

```
from collections import defaultdict
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        adj = defaultdict(list)
        for u,v in prerequisites:
            adj[v].append(u)
        #print(adj)
        path_vis = [0]*numCourses
        visited = set()
        def dfs(node):
            path_vis[node] = 1
            visited.add(node)
            for neigh in adj[node]:
                if neigh not in visited:
                    if dfs(neigh):
                        return True
                elif path_vis[neigh] == 1:
                    return True
            path_vis[node] = 0
            return False

        for i in range(numCourses):
            if dfs(i):
                return not True

        return True
```

210. Course Schedule II

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] = [$a_i$, $b_i$]` indicates that you **must** take course `$b_i$` first if you want to take course `$a_i$`.

- 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
```

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
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 <= a_i, b_i < numCourses`
- `a_i != b_i`
- All the pairs `[a_i, b_i]` are **distinct**.

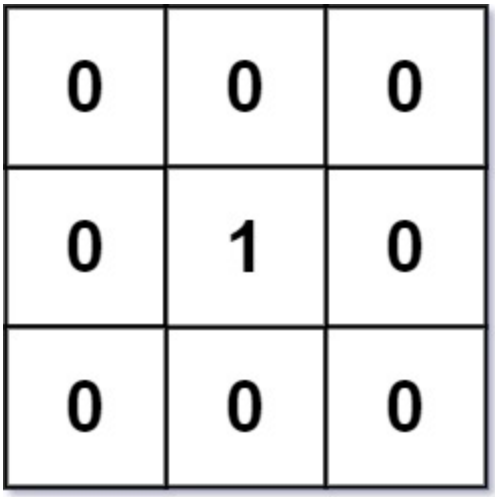
```
from collections import defaultdict,deque
class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List[in
t]:
        vis_set = set()
        adj = defaultdict(list)
        vis = 0
        que = deque([])
        indegree = [0]*numCourses
        for x,y in prerequisites:
            adj[y].append(x)
            indegree[x] +=1

        for course in range(numCourses):
            if indegree[course] == 0:
                que.append(course)
        ans = []
        while que:
            node = que.popleft()
            vis_set.add(node)
            vis+=1
            ans.append(node)
            for neigh in adj[node]:
                indegree[neigh]-=1
                if indegree[neigh] == 0:
                    if neigh not in vis_set:
                        que.append(neigh)
        return [] if vis != numCourses else ans
```

542. 01 Matrix

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

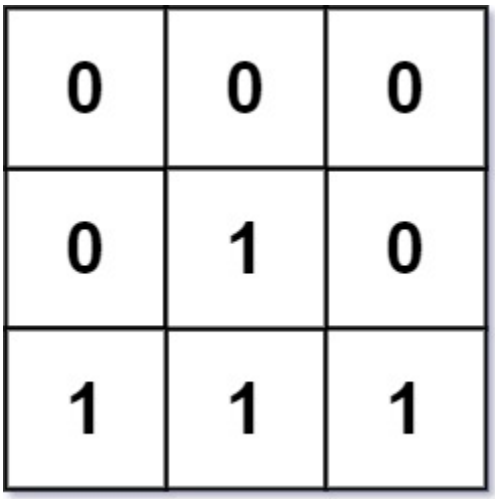
The distance between two cells sharing a common edge is `1`.

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 <= 10^4`
- `1 <= m * n <= 10^4`
- `mat[i][j]` is either `0` or `1`.
- There is at least one `0` in `mat`.

Note: This question is the same as 1765: https://leetcode.com/problems/map-of-highest-peak/ (https://leetcode.com/problems/map-of-highest-peak/description/)

```
from collections import deque
class Solution:
    def updateMatrix(self, mat: List[List[int]]) -> List[List[int]]:
        m = len(mat)
        n = len(mat[0])
        que = deque()
        dirs = [(1,0),(0,1),(-1,0),(0,-1)]
        for i in range(m):
            for j in range(n):
                if mat[i][j] == 0:
                    que.append([i,j])
                else:
                    mat[i][j] = -1
        while que:
            x,y = que.popleft()
            for nx,ny in dirs:
                nx,ny = x+nx,y+ny
                if 0<=nx<m and 0<=ny<n and mat[nx][ny] == -1 :
                    mat[nx][ny]= mat[x][y]+1
                    que.append((nx,ny))
        return mat
```

547. Number of Provinces

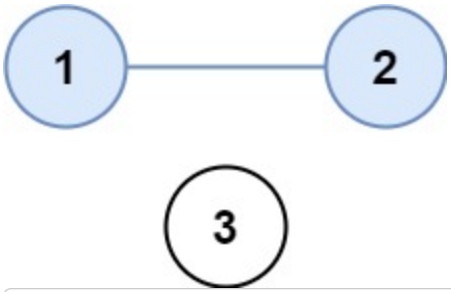
There are `n` cities. Some of them are connected, while some are not. If city `a` is connected directly with city `b`, and city `b` is connected directly with city `c`, then city `a` is connected indirectly with city `c`.

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

You are given an `n x n` matrix `isConnected` where `isConnected[i][j] = 1` if the i^{th} city and the j^{th} city are directly connected, and `isConnected[i][j] = 0` otherwise.

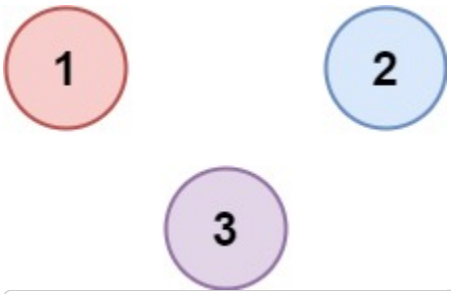
Return *the total number of* ***provinces***.

Example 1:




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

Example 2:




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

Constraints:

- `1 <= n <= 200`
- `n == isConnected.length`
- `n == isConnected[i].length`
- `isConnected[i][j]` is `1` or `0`.
- `isConnected[i][i] == 1`
- `isConnected[i][j] == isConnected[j][i]`

```
class Solution:
    def findCircleNum(self, nums: List[List[int]]) -> int:
        n=len(nums)
        adj_list = [[] for _ in range(n)]
        visited = [False] * n

        for i in range(n):
            for j in range(len(nums[0])):
                if i!=j and nums[i][j]==1:
                    adj_list[i].append(j)

        def dfs(node: int):
            visited[node]=True
            for neighbours in adj_list[node]:
                if not visited[neighbours]:
                    dfs(neighbours)
        ans=0
        for i in range(n):
            if not visited[i]:
                dfs(i)
                ans+=1
        return  ans
```

---

721. Accounts Merge

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","joh
Output: [["John","john00@mail.com","john_newyork@mail.com","johnsmith@mail.com"],["Ma
Explanation:
The first and second John's are the same person as they have the common email "johnsm
The third John and Mary are different people as none of their email addresses are use
We could return these lists in any order, for example the answer [['Mary', 'mary@mail
['John', 'john00@mail.com', 'john_newyork@mail.com', 'johnsmith@mail.com']] would sti
```

Example 2:

```
Input: accounts = [["Gabe","Gabe0@m.co","Gabe3@m.co","Gabe1@m.co"],["Kevin","Kevin3@m
Output: [["Ethan","Ethan0@m.co","Ethan4@m.co","Ethan5@m.co"],["Gabe","Gabe0@m.co","Ga
```

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.

```
from typing import List
from collections import defaultdict

class UnionFind:
    def __init__(self):
        self.parent = {}

    def find(self, x):
        if x != self.parent.setdefault(x, x):
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        self.parent[self.find(x)] = self.find(y)

class Solution:
    def accountsMerge(self, accounts: List[List[str]]) -> List[List[str]]:
        uf = UnionFind()
        email_to_name = {}

        # Step 1: Union emails within the same account
        for account in accounts:
            name = account[0]
            first_email = account[1]
            for email in account[1:]:
                uf.union(first_email, email)
                email_to_name[email] = name

        # Step 2: Group emails by their root parent
        groups = defaultdict(list)
        for email in email_to_name:
            root = uf.find(email)
            groups[root].append(email)

        # Step 3: Build the result
        result = []
        for root, emails in groups.items():
            name = email_to_name[root]
            result.append([name] + sorted(emails))

        return result
```

733. Flood Fill

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**:

1. Begin with the starting pixel and change its color to `color`.
2. 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.
3. 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.
4. The process **stops** when there are **no more** adjacent pixels of the original color to update.

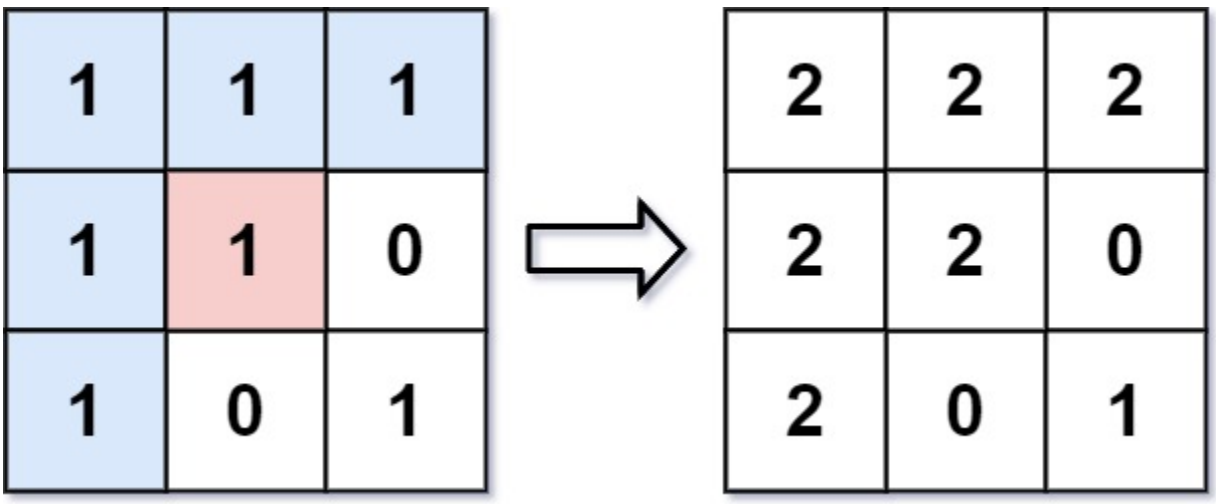
Return the **modified** image after performing the flood fill.

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`

```
from collections import deque
class Solution:
    def floodFill(self, image: List[List[int]], sr: int, sc: int, color: int) -> Li
st[List[int]]:
        que = deque([[sr,sc]])
        dirs = [(-1,0),(0,-1),(1,0),(0,1)]
        m = len(image)
        n = len(image[0])
        col = image[sr][sc]
        if image[sr][sc] == color:
            return image
        while que:
            x,y = que.popleft()
            image[x][y] = color
            for nx,ny in dirs:
                nx,ny = x+nx,y+ny
                if 0 <=nx <m and 0 <=ny < n and col == image[nx][ny]:
                    que.append([nx,ny])
        return image
```

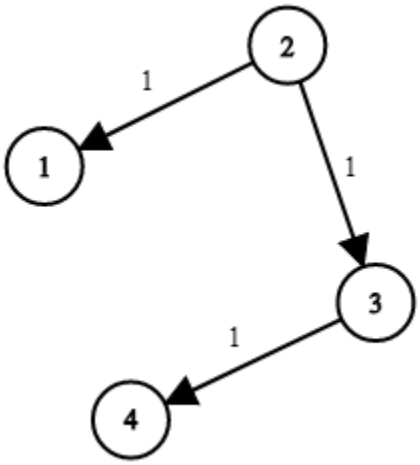
743. Network Delay Time

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] = (u_i, v_i, w_i)`, where `u_i` is the source node, `v_i` is the target node, and `w_i` 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 <= u_i, v_i <= n`
- `u_i != v_i`
- `0 <= w_i <= 100`
- All the pairs `(u_i, v_i)` are **unique**. (i.e., no multiple edges.)

```
from collections import defaultdict
import heapq
class Solution:
    def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:
        adj = defaultdict(list)
        dist = [float("inf")]*(n+1)
        dist[k] = 0
        for x,y,z in times:
            adj[x].append([y,z])

        heap = [[0,k]]
        while heap:
            d,node = heapq.heappop(heap)
            for neigh,weight in adj[node]:
                if  weight+dist[node] < dist[neigh]:
                    dist[neigh] = weight+dist[node]
                    heapq.heappush(heap,[dist[neigh],neigh])
        maxx = -float("inf")
        for i in range(1,n+1):
            if dist[i] == float("inf"):
                return -1
            maxx = max(maxx,dist[i])
        return maxx
```

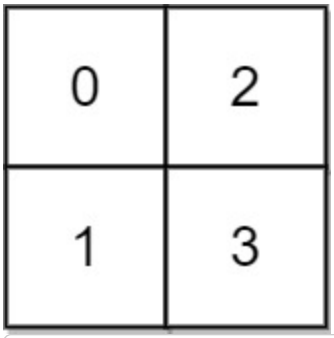
778. Swim in Rising Water

You are given an `n x n` integer matrix `grid` where each value `grid[i][j]` represents the elevation at that point `(i, j)`.

The rain starts to fall. At time `t`, the depth of the water everywhere is `t`. You can swim from a square to another 4-directionally adjacent square if and only if the elevation of both squares individually are at most `t`. You can swim infinite distances in zero time. Of course, you must stay within the boundaries of the grid during your swim.

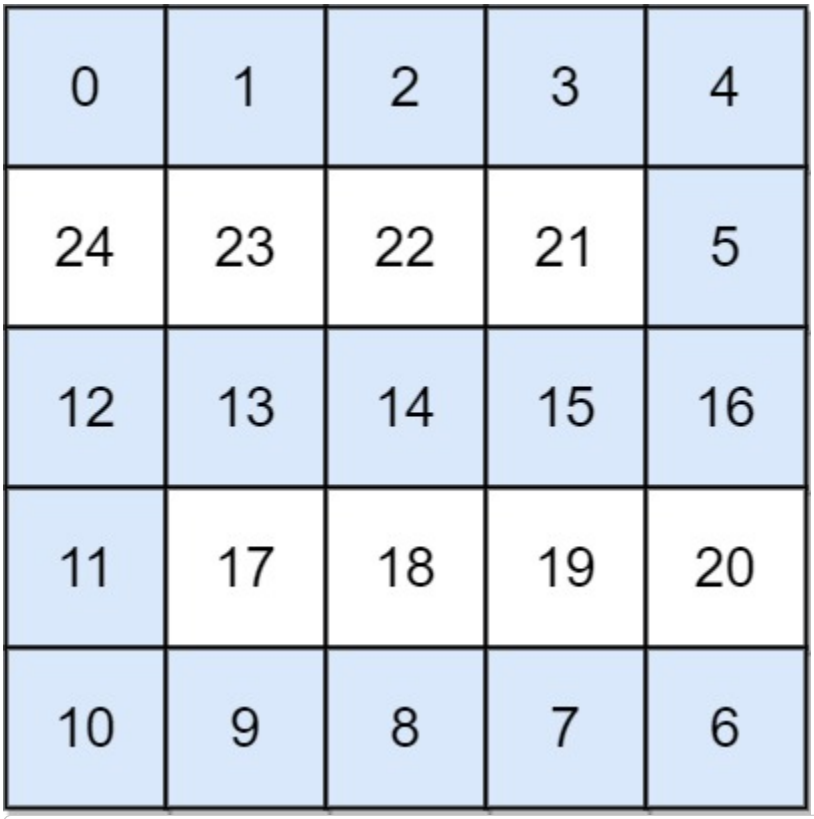
Return *the least time until you can reach the bottom right square* `(n - 1, n - 1)` *if you start at the top left square* `(0, 0)`.

Example 1:




```
Input: grid = [[0,2],[1,3]]
Output: 3
Explanation:
At time 0, you are in grid location (0, 0).
You cannot go anywhere else because 4-directionally adjacent neighbors have a higher
You cannot reach point (1, 1) until time 3.
When the depth of water is 3, we can swim anywhere inside the grid.
```

Example 2:




```
Input: grid = [[0,1,2,3,4],[24,23,22,21,5],[12,13,14,15,16],[11,17,18,19,20],[10,9,8,
Output: 16
Explanation: The final route is shown.
We need to wait until time 16 so that (0, 0) and (4, 4) are connected.
```

Constraints:

- `n == grid.length`
- `n == grid[i].length`
- `1 <= n <= 50`
- `0 <= grid[i][j] < n^2`
- Each value `grid[i][j]` is **unique**.

```
class Solution:
    def swimInWater(self, grid: List[List[int]]) -> int:
        time = 0
        dirs = [(0,1),(1,0),(-1,0),(0,-1)]
        zero = grid[0][0]
        heap = [(zero,0,0)]
        time = 0
        n = len(grid)
        while heap:
            weight,x,y = heapq.heappop(heap)
            grid[x][y] = "#"
            if time < weight:
                time = weight
            if x == n-1 and y == n-1:
                return time
            for nx,ny in dirs:
                nx,ny = x+nx,y+ny
                if 0<=nx<n and 0<=ny<n and grid[nx][ny] != "#":
                    weight = grid[nx][ny]
                    heapq.heappush(heap,(weight,nx,ny))
        return -1
```

785. Is Graph Bipartite?

There is an **undirected** graph with `n` nodes, where each node is numbered between `0` and `n - 1`. You are given a 2D array `graph`, where `graph[u]` is an array of nodes that node `u` is adjacent to. More formally, for each `v` in `graph[u]`, there is an undirected edge between node `u` and node `v`. The graph has the following properties:

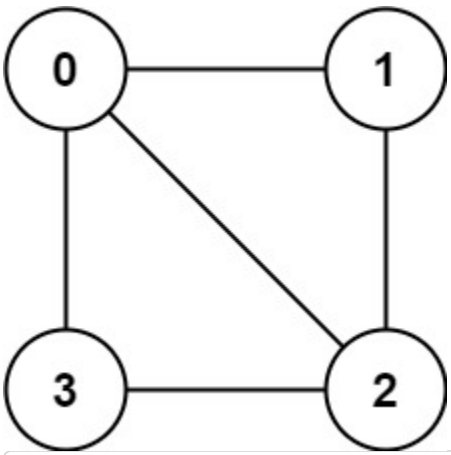
- There are no self-edges (`graph[u]` does not contain `u`).
- There are no parallel edges (`graph[u]` does not contain duplicate values).
- If `v` is in `graph[u]`, then `u` is in `graph[v]` (the graph is undirected).
- The graph may not be connected, meaning there may be two nodes `u` and `v` such that there is no path between them.

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`.

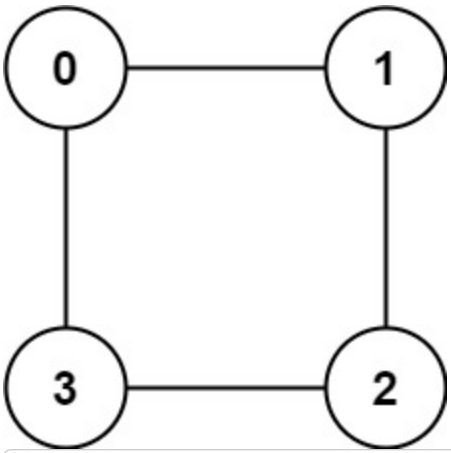
Return `true` *if and only if it is* ***bipartite***.

Example 1:




```
Input: graph = [[1,2,3],[0,2],[0,1,3],[0,2]]
Output: false
Explanation: There is no way to partition the nodes into two independent sets such th
```

Example 2:




```
Input: graph = [[1,3],[0,2],[1,3],[0,2]]
Output: true
Explanation: We can partition the nodes into two sets: {0, 2} and {1, 3}.
```

Constraints:

- `graph.length == n`
- `1 <= n <= 100`
- `0 <= graph[u].length < n`

- `0 <= graph[u][i] <= n - 1`
- `graph[u]` does not contain `u`.
- All the values of `graph[u]` are **unique**.
- If `graph[u]` contains `v`, then `graph[v]` contains `u`.

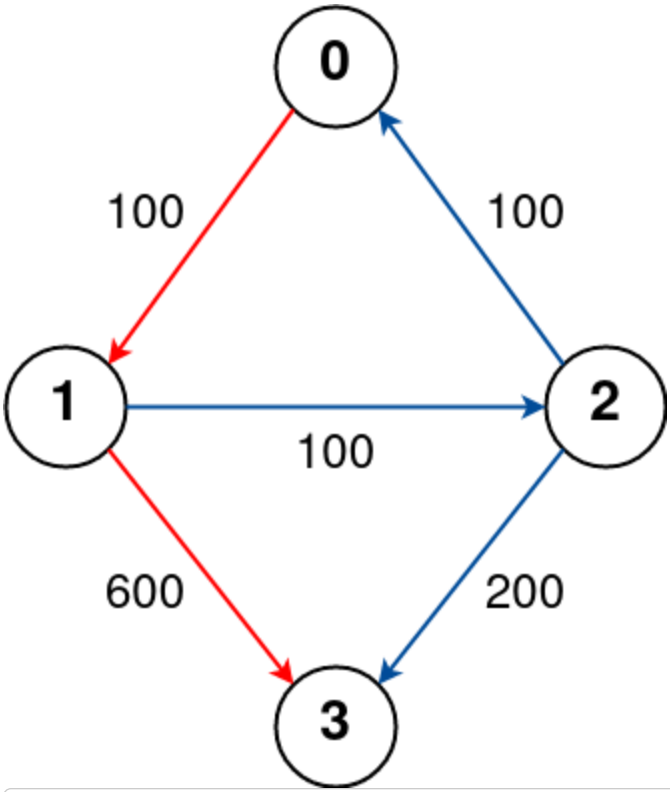
```
from collections import deque
class Solution:
    def isBipartite(self, graph: List[List[int]]) -> bool:
        n = len(graph)
        color = [-1]*n
        for start in range(n):
            if color[start] == -1:
                que = deque([start])
                color[start] = 0
                while que:
                    node = que.popleft()
                    for neigh in graph[node]:
                        if color[neigh] == -1:
                            color[neigh] = 1-color[node]
                            que.append(neigh)
                        elif color[neigh] == color[node]:
                            return False
        return True
```

787. Cheapest Flights Within K Stops

There are `n` cities connected by some number of flights. You are given an array `flights` where `flights[i] = [from_i, to_i, price_i]` indicates that there is a flight from city `from_i` to city `to_i` with cost `price_i`.

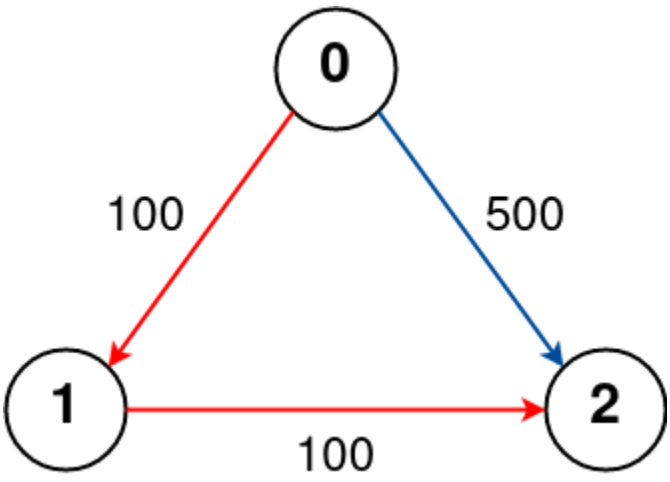
You are also given three integers `src`, `dst`, and `k`, return ***the cheapest price*** *from* `src` *to* `dst` *with at most* `k` *stops.* If there is no such route, return `-1`.

Example 1:



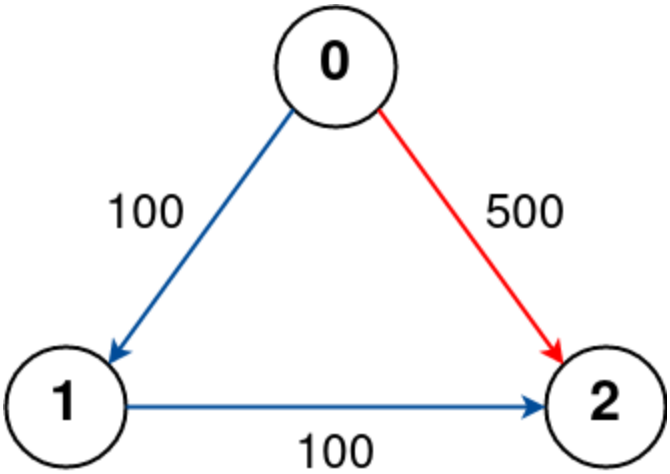

```
Input: n = 4, flights = [[0,1,100],[1,2,100],[2,0,100],[1,3,600],[2,3,200]], src = 0,
Output: 700
Explanation:
The graph is shown above.
The optimal path with at most 1 stop from city 0 to 3 is marked in red and has cost 1
Note that the path through cities [0,1,2,3] is cheaper but is invalid because it uses
```

Example 2:




```
Input: n = 3, flights = [[0,1,100],[1,2,100],[0,2,500]], src = 0, dst = 2, k = 1
Output: 200
Explanation:
The graph is shown above.
The optimal path with at most 1 stop from city 0 to 2 is marked in red and has cost 1
```

Example 3:




```
Input: n = 3, flights = [[0,1,100],[1,2,100],[0,2,500]], src = 0, dst = 2, k = 0
Output: 500
Explanation:
The graph is shown above.
The optimal path with no stops from city 0 to 2 is marked in red and has cost 500.
```

Constraints:

- `1 <= n <= 100`
- `0 <= flights.length <= (n * (n - 1) / 2)`
- `flights[i].length == 3`
- `0 <= from_i, to_i < n`
- `from_i != to_i`
- `1 <= price_i <= 10^4`
- There will not be any multiple flights between two cities.
- `0 <= src, dst, k < n`
- `src != dst`

```
import heapq
from collections import defaultdict
from typing import List

class Solution:
    def findCheapestPrice(self, n: int, flights: List[List[int]], src: int, dst: in
t, k: int) -> int:
        adj = defaultdict(list)
        for u, v, w in flights:
            adj[u].append((v, w))

        # Min-heap: (total cost so far, current city, stops used so far)
        heap = [(0, src, 0)]
        # best[node] = (min_cost, min_stops)
        best = dict()

        while heap:
            cost, node, stops = heapq.heappop(heap)

            if node == dst:
                return cost

            if stops > k:
                continue

            # Avoid exploring worse paths
            if (node in best and best[node] <= stops):
                continue

            best[node] = stops

            for nei, price in adj[node]:
                heapq.heappush(heap, (cost + price, nei, stops + 1))

        return -1
```

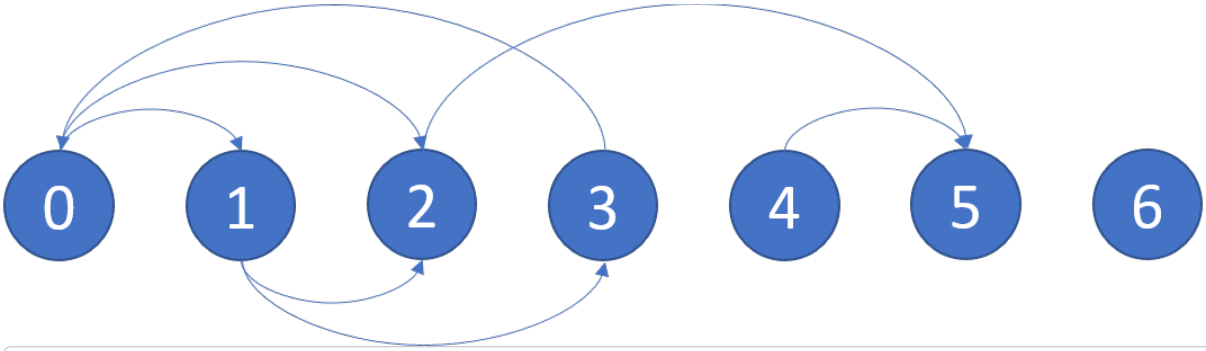
802. Find Eventual Safe States

There is a directed graph of `n` nodes with each node labeled from `0` to `n - 1`. The graph is represented by a **0-indexed** 2D integer array `graph` where `graph[i]` is an integer array of nodes adjacent to node `i`, meaning there is an edge from node `i` to each node in `graph[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** (or another safe node).

Return *an array containing all the **safe nodes** of the graph*. The answer should be sorted in **ascending** order.

Example 1:




```
Input: graph = [[1,2],[2,3],[5],[0],[5],[],[]]
Output: [2,4,5,6]
Explanation: The given graph is shown above.
Nodes 5 and 6 are terminal nodes as there are no outgoing edges from either of them.
Every path starting at nodes 2, 4, 5, and 6 all lead to either node 5 or 6.
```

Example 2:

```
Input: graph = [[1,2,3,4],[1,2],[3,4],[0,4],[]]
Output: [4]
Explanation:
Only node 4 is a terminal node, and every path starting at node 4 leads to node 4.
```

Constraints:

- `n == graph.length`
- `1 <= n <= 10^4`
- `0 <= graph[i].length <= n`
- `0 <= graph[i][j] <= n - 1`
- `graph[i]` is sorted in a strictly increasing order.
- The graph may contain self-loops.
- The number of edges in the graph will be in the range `[1, 4 * 10^4]`.

```
from collections import defaultdict,deque
class Solution:
    def eventualSafeNodes(self, graph: List[List[int]]) -> List[int]:
        adj = defaultdict(list)
        indegree = [0]*len(graph)
        for j in range(len(graph)):
            for i in range(len(graph[j])):
                adj[graph[j][i]].append(j)
                indegree[j] +=1
        que = deque()
        for i in range(len(indegree)):
            if indegree[i] == 0:
                que.append(i)

        ans = []
        while que:
            node = que.popleft()
            ans.append(node)
            for neigh in adj[node]:
                indegree[neigh]-=1
                if indegree[neigh] == 0:
                    que.append(neigh)
        return sorted(ans)
```

827. Making A Large Island

You are given an `n x n` binary matrix `grid` . You are allowed to change **at most one** `0` to be `1` .

Return *the size of the largest **island** in* `grid` *after applying this operation.*

An **island** is a 4-directionally connected group of `1` s.

Example 1:

```
Input: grid = [[1,0],[0,1]]
Output: 3
Explanation: Change one 0 to 1 and connect two 1s, then we get an island with area =
```

Example 2:

```
Input: grid = [[1,1],[1,0]]
Output: 4
Explanation: Change the 0 to 1 and make the island bigger, only one island with area
```

Example 3:

```
Input: grid = [[1,1],[1,1]]
Output: 4
Explanation: Can't change any 0 to 1, only one island with area = 4.
```

Constraints:

- `n == grid.length`
- `n == grid[i].length`
- `1 <= n <= 500`
- `grid[i][j]` is either `0` or `1`.

```
class DSU:
    def __init__(self, size):
        self.parent = list(range(size))
        self.size = [1] * size

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        x_root = self.find(x)
        y_root = self.find(y)
        if x_root == y_root:
            return
        if self.size[x_root] < self.size[y_root]:
            x_root, y_root = y_root, x_root
        self.parent[y_root] = x_root
        self.size[x_root] += self.size[y_root]

class Solution:
    def largestIsland(self, grid: List[List[int]]) -> int:
        n = len(grid)
        dsu = DSU(n * n)

        # Directions for up, down, left, right
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

        # Step 1: Union all connected 1s and record their sizes
        for i in range(n):
            for j in range(n):
                if grid[i][j] == 1:
                    for di, dj in directions:
                        ni, nj = i + di, j + dj
                        if 0 <= ni < n and 0 <= nj < n and grid[ni][nj] == 1:
                            dsu.union(i * n + j, ni * n + nj)

        # Step 2: Find the maximum island size after converting each 0 to 1
        max_size = 0
        has_zero = False

        for i in range(n):
            for j in range(n):
                if grid[i][j] == 0:
                    has_zero = True
                    neighbors = set()
                    for di, dj in directions:
```

```
                    ni, nj = i + di, j + dj
                    if 0 <= ni < n and 0 <= nj < n and grid[ni][nj] == 1:
                        root = dsu.find(ni * n + nj)
                        neighbors.add(root)
                current_size = 1  # the current 0 turned to 1
                for root in neighbors:
                    current_size += dsu.size[root]
                max_size = max(max_size, current_size)

        # If there are no zeros, the entire grid is the largest island
        if not has_zero:
            return n * n

        return max_size
```

947. Most Stones Removed with Same Row or Column

On a 2D plane, we place `n` stones at some integer coordinate points. Each coordinate point may have at most one stone.

A stone can be removed if it shares either **the same row or the same column** as another stone that has not been removed.

Given an array `stones` of length `n` where `stones[i] = [x_i, y_i]` represents the location of the `i^th` stone, return *the largest possible number of stones that can be removed*.

Example 1:

```
Input: stones = [[0,0],[0,1],[1,0],[1,2],[2,1],[2,2]]
Output: 5
Explanation: One way to remove 5 stones is as follows:
1. Remove stone [2,2] because it shares the same row as [2,1].
2. Remove stone [2,1] because it shares the same column as [0,1].
3. Remove stone [1,2] because it shares the same row as [1,0].
4. Remove stone [1,0] because it shares the same column as [0,0].
5. Remove stone [0,1] because it shares the same row as [0,0].
Stone [0,0] cannot be removed since it does not share a row/column with another stone
```

Example 2:

```
Input: stones = [[0,0],[0,2],[1,1],[2,0],[2,2]]
Output: 3
Explanation: One way to make 3 moves is as follows:
1. Remove stone [2,2] because it shares the same row as [2,0].
2. Remove stone [2,0] because it shares the same column as [0,0].
3. Remove stone [0,2] because it shares the same row as [0,0].
Stones [0,0] and [1,1] cannot be removed since they do not share a row/column with ar
```

Example 3:

```
Input: stones = [[0,0]]
Output: 0
Explanation: [0,0] is the only stone on the plane, so you cannot remove it.
```

Constraints:

- `1 <= stones.length <= 1000`
- `0 <= x_i, y_i <= 10^4`
- No two stones are at the same coordinate point.

```
from typing import List

class UnionFind:
    def __init__(self, size):
        self.par = list(range(size))

    def find(self, n):
        while n != self.par[n]:
            self.par[n] = self.par[self.par[n]]
            n = self.par[n]
        return n

    def union(self, a, b):
        root_a = self.find(a)
        root_b = self.find(b)
        if root_a == root_b:
            return False
        self.par[root_a] = root_b
        return True

class Solution:
    def removeStones(self, stones: List[List[int]]) -> int:
        max_coord = 10001  # problem constraints: 0 <= x, y <= 10000
        uf = UnionFind(2 * max_coord)
        seen = set()

        for x, y in stones:
            uf.union(x, y + max_coord)  # offset y to avoid collision with x
            seen.add(x)
            seen.add(y + max_coord)

        # count number of unique roots
        roots = set()
        for node in seen:
            roots.add(uf.find(node))

        return len(stones) - len(roots)
```

994. Rotting Oranges

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

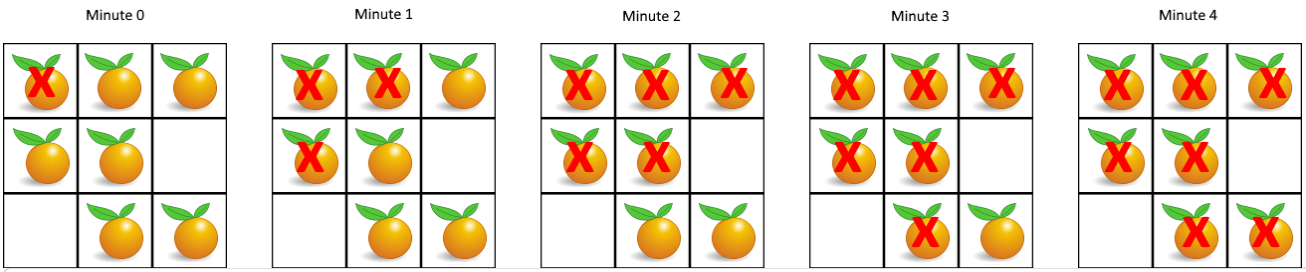
- `0` representing an empty cell,
- `1` representing a fresh orange, or

- 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,
```

Example 3:

```
Input: grid = [[0,2]]
Output: 0
Explanation: Since there are already no fresh oranges at minute 0, the answer is just
```

Constraints:

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

```
from collections import deque
class Solution:
    def orangesRotting(self, grid: List[List[int]]) -> int:
        que = deque()
        m = len(grid)
        n = len(grid[0])
        fresh = 0
        for i in range(m):
            for j in range(n):
                if grid[i][j] == 2:
                    que.append([i,j])
                elif grid[i][j] == 1:
                    fresh+=1
        if fresh == 0:
            return 0
        time = 0
        dirs = [(-1,0),(0,-1),(1,0),(0,1)]
        while que:
            for _ in range(len(que)):
                x,y = que.popleft()
                for dx,dy in dirs:
                    nx,ny = x+dx,y+dy
                    if  0<=nx<m and 0<=ny<n and grid[nx][ny] == 1:
                        grid[nx][ny] = 2
                        fresh-=1
                        que.append([nx,ny])
            if que:
                time +=1
        return time if fresh == 0 else -1
```

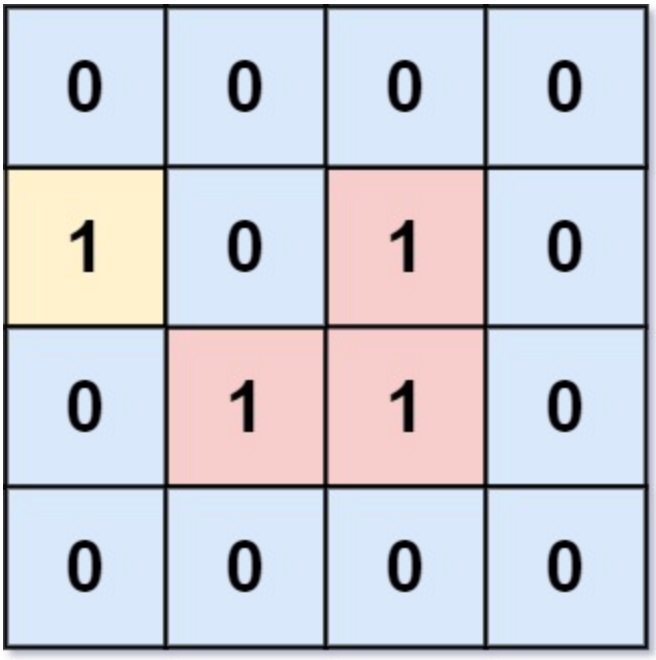
1020. Number of Enclaves

You are given an `m x n` 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`.

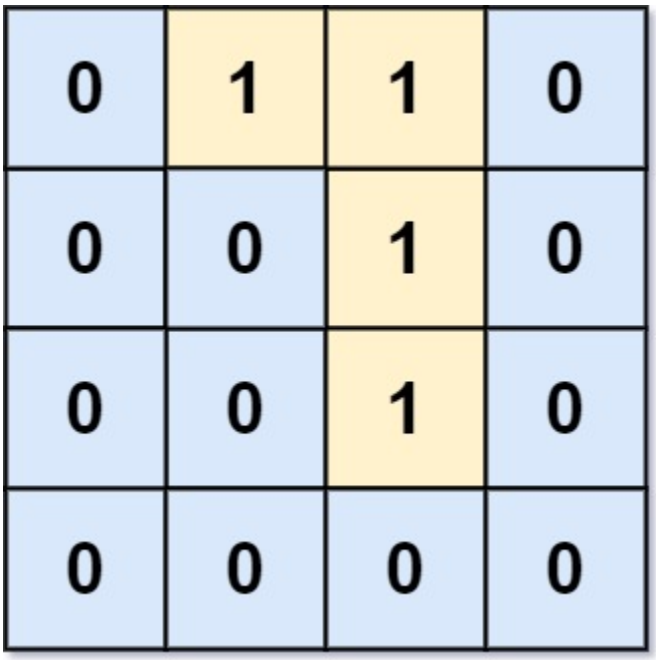
Return *the number of land cells in* `grid` *for which we cannot walk off the boundary of the grid in any number of* ***moves***.

Example 1:




```
Input: grid = [[0,0,0,0],[1,0,1,0],[0,1,1,0],[0,0,0,0]]
Output: 3
Explanation: There are three 1s that are enclosed by 0s, and one 1 that is not enclos
```

Example 2:




```
Input: grid = [[0,1,1,0],[0,0,1,0],[0,0,1,0],[0,0,0,0]]
Output: 0
Explanation: All 1s are either on the boundary or can reach the boundary.
```

Constraints:

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 500`
- `grid[i][j]` is either `0` or `1`.

```
class Solution:
    def numEnclaves(self, grid: List[List[int]]) -> int:
        m = len(grid)
        n = len(grid[0])
        dirs = [(0,1),(1,0),(-1,0),(0,-1)]
        def dfs(i,j):
            grid[i][j] = "E"
            for nx,ny in dirs:
                nx,ny = i+nx,j+ny
                if 0<=nx<m and 0<=ny<n and grid[nx][ny] == 1:
                    dfs(nx,ny)


        for i in range(m):
            if grid[i][0] == 1:
                dfs(i,0)
            if grid[i][n-1] == 1:
                dfs(i,n-1)

        for i in range(n):
            if grid[0][i] == 1:
                dfs(0,i)
            if grid[m-1][i] == 1:
                dfs(m-1,i)

        c  = 0
        for i in range(m):
            for j in range(n):
                if grid[i][j] == 1:
                    c+=1
        return c
```

1091. Shortest Path in Binary Matrix

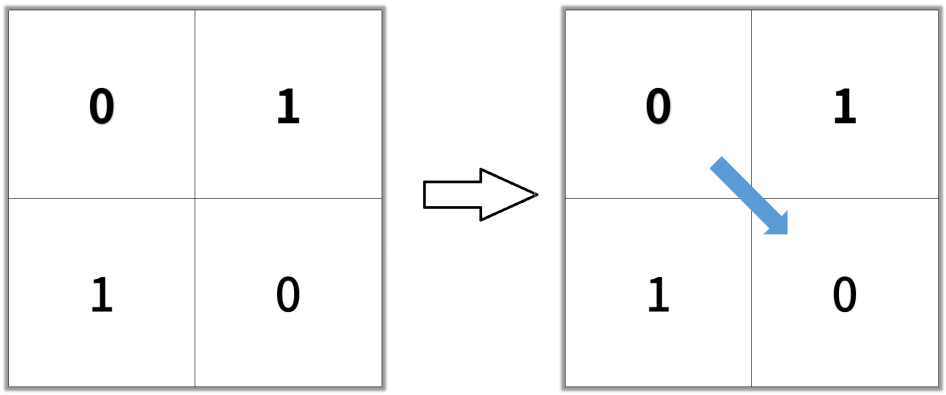
Given an `n x n` binary matrix `grid`, return *the length of the shortest **clear path** in the matrix*. If there is no clear path, return `-1`.

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).

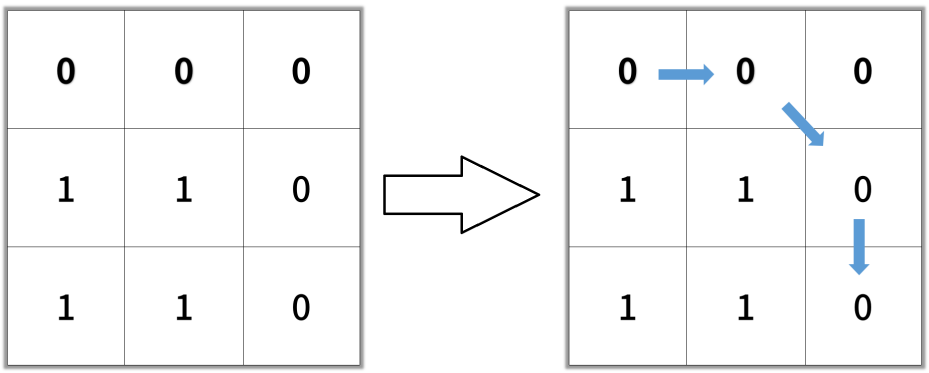
The **length of a clear path** is the number of visited cells of this path.

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

```
import heapq
from collections import deque
class Solution:
    def shortestPathBinaryMatrix(self, grid: List[List[int]]) -> int:
        pq = deque([[0,0]])
        n = len(grid)
        if grid[0][0] == 1 or grid[n-1][n-1] == 1:return -1
        dist = [[float("inf") for _ in range(n)] for _ in range(n)]
        dist[0][0] = 1
        dirs = [(0,1),(1,0),(-1,0),(0,-1),(1,1),(-1,-1),(1,-1),(-1,1)]
        while pq:
            x,y = pq.popleft()
            d = dist[x][y]
            for nx,ny in dirs:
                nx,ny = x+nx,y+ny
                if 0<=nx < n and 0<=ny<n and grid[nx][ny] == 0 and d+1 <dist[nx][n
y]:
                    dist[nx][ny] = d+1
                    pq.append((nx,ny))
        return dist[n-1][n-1] if dist[n-1][n-1] != float("inf") else -1
```

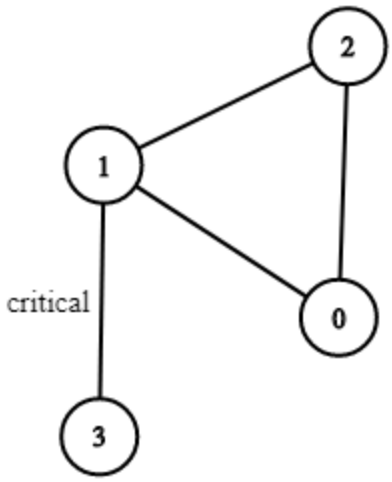
1192. Critical Connections in a Network

There are n servers numbered from 0 to $n - 1$ connected by undirected server-to-server $connections$ forming a network where $connections[i] = [a_i, b_i]$ represents a connection between servers a_i and b_i. Any server can reach other servers directly or indirectly through the network.

A *critical connection* is a connection that, if removed, will make some servers unable to reach some other server.

Return all critical connections in the network in any order.

Example 1:




```
Input: n = 4, connections = [[0,1],[1,2],[2,0],[1,3]]
Output: [[1,3]]
Explanation: [[3,1]] is also accepted.
```

Example 2:

```
Input: n = 2, connections = [[0,1]]
Output: [[0,1]]
```

Constraints:

- $2 <= n <= 10^5$
- $n - 1 <= connections.length <= 10^5$
- $0 <= a_i, b_i <= n - 1$
- $a_i != b_i$
- There are no repeated connections.

```
from typing import List

class Solution:
    def criticalConnections(self, n: int, connections: List[List[int]]) -> List[Lis
t[int]]:
        graph = [[] for _ in range(n)]
        for u, v in connections:
            graph[u].append(v)
            graph[v].append(u)

        disc = [-1] * n      # Discovery times
        low = [-1] * n       # Lowest discovery times
        time = [0]           # Global timer
        result = []

        def dfs(u, parent):
            disc[u] = low[u] = time[0]
            time[0] += 1

            for v in graph[u]:
                if v == parent:
                    continue
                if disc[v] == -1:
                    dfs(v, u)
                    low[u] = min(low[u], low[v])
                    if low[v] > disc[u]:
                        result.append([u, v])
                else:
                    low[u] = min(low[u], disc[v])

        dfs(0, -1)
        return result
```

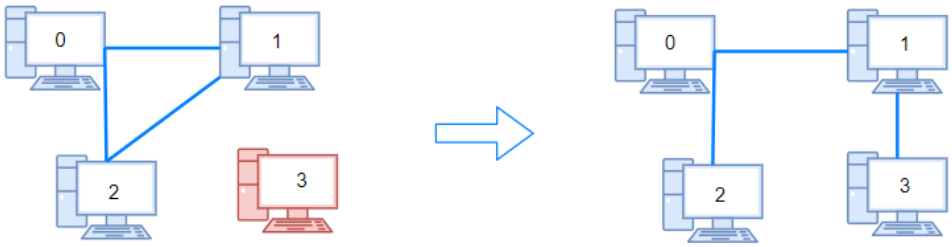
1319. Number of Operations to Make Network Connected

There are `n` computers numbered from `0` to `n - 1` connected by ethernet cables `connections` forming a network where `connections[i] = [a_i, b_i]` represents a connection between computers a_i and b_i. Any computer can reach any other computer directly or indirectly through the network.

You are given an initial computer network `connections`. You can extract certain cables between two directly connected computers, and place them between any pair of disconnected computers to make them directly connected.

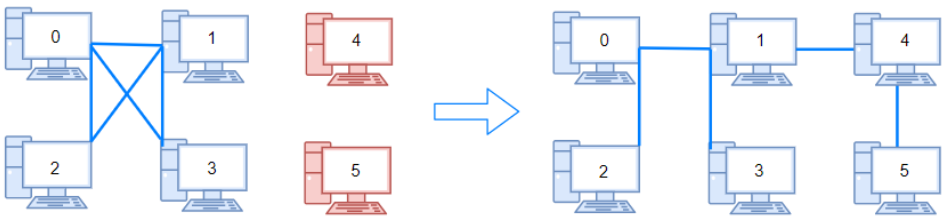
Return *the minimum number of times you need to do this in order to make all the computers connected*. If it is not possible, return `-1` .

Example 1:




```
Input: n = 4, connections = [[0,1],[0,2],[1,2]]
Output: 1
Explanation: Remove cable between computer 1 and 2 and place between computers 1 and
```

Example 2:




```
Input: n = 6, connections = [[0,1],[0,2],[0,3],[1,2],[1,3]]
Output: 2
```

Example 3:

```
Input: n = 6, connections = [[0,1],[0,2],[0,3],[1,2]]
Output: -1
Explanation: There are not enough cables.
```

Constraints:

- `1 <= n <= 10^5`
- `1 <= connections.length <= min(n * (n - 1) / 2, 10^5)`
- `connections[i].length == 2`
- `0 <= a_i, b_i < n`
- `a_i != b_i`
- There are no repeated connections.

- No two computers are connected by more than one cable.

```
class UnionFind:
    def __init__(self,V):
        self.par = list(range(V))
        self.rank = [0]*V
    def get_parent(self):
        return self.par
    def find(self,n):
        while n != self.par[n]:
            self.par[n] = self.par[self.par[n]]
            n = self.par[n]
        return n

    def union(self,a,b):
        root_a = self.find(a)
        root_b = self.find(b)
        if root_a == root_b:
            return False
        self.par[root_a] = root_b
        return True
class Solution:
    def makeConnected(self, n: int, connections: List[List[int]]) -> int:
        if len(connections) < n-1:
            return -1
        cables = len(connections)
        uf = UnionFind(n)
        same_par = 0
        for u,v in connections:
            if not uf.union(u,v):
                same_par+=1
        par = uf.get_parent()
        c = 0
        for i in range(n):
            if par[i] == i:
                c+=1
        c-=1
        if same_par >= c:
            return c

        return same_par if same_par else -1
```

1334. Find the City With the Smallest Number of

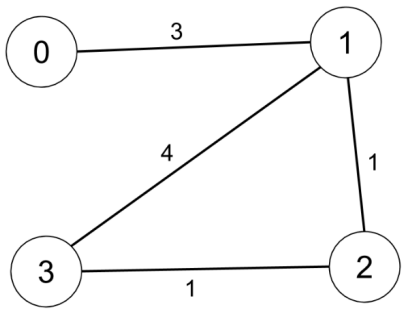
Neighbors at a Threshold Distance

There are `n` cities numbered from `0` to `n-1`. Given the array `edges` where `edges[i] = [from`$_i$`, to`$_i$`, weight`$_i$`]` represents a bidirectional and weighted edge between cities `from`$_i$ and `to`$_i$, and given the integer `distanceThreshold`.

Return the city with the smallest number of cities that are reachable through some path and whose distance is **at most** `distanceThreshold`, If there are multiple such cities, return the city with the greatest number.

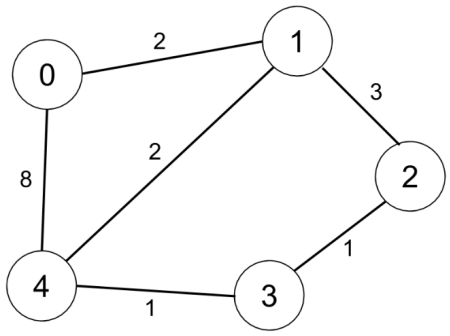
Notice that the distance of a path connecting cities ***i*** and ***j*** is equal to the sum of the edges' weights along that path.

Example 1:




```
Input: n = 4, edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], distanceThreshold = 4
Output: 3
Explanation: The figure above describes the graph.
The neighboring cities at a distanceThreshold = 4 for each city are:
City 0 -> [City 1, City 2]
City 1 -> [City 0, City 2, City 3]
City 2 -> [City 0, City 1, City 3]
City 3 -> [City 1, City 2]
Cities 0 and 3 have 2 neighboring cities at a distanceThreshold = 4, but we have to r
```

Example 2:




```
Input: n = 5, edges = [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]], distanceThre
Output: 0
Explanation: The figure above describes the graph.
The neighboring cities at a distanceThreshold = 2 for each city are:
City 0 -> [City 1]
City 1 -> [City 0, City 4]
City 2 -> [City 3, City 4]
City 3 -> [City 2, City 4]
City 4 -> [City 1, City 2, City 3]
The city 0 has 1 neighboring city at a distanceThreshold = 2.
```

Constraints:

- `2 <= n <= 100`
- `1 <= edges.length <= n * (n - 1) / 2`
- `edges[i].length == 3`
- $0 \le from_i < to_i < n$
- $1 \le weight_i$, `distanceThreshold <= 10^4`
- All pairs $(from_i, to_i)$ are distinct.

```
class Solution:
    def findTheCity(self, n: int, edges: List[List[int]], distanceThreshold: int) -
> int:
        INF = float('inf')
        # Step 1: Initialize distance matrix
        dist = [[INF] * n for _ in range(n)]
        for i in range(n):
            dist[i][i] = 0

        # Step 2: Set direct edge distances
        for u, v, w in edges:
            dist[u][v] = w
            dist[v][u] = w

        # Step 3: Floyd-Warshall to compute all-pairs shortest path
        for k in range(n):
            for i in range(n):
                for j in range(n):
                    if dist[i][k] + dist[k][j] < dist[i][j]:
                        dist[i][j] = dist[i][k] + dist[k][j]

        # Step 4: Find city with minimum reachable cities
        minCount = n + 1
        resultCity = -1

        for i in range(n):
            count = 0
            for j in range(n):
                if i != j and dist[i][j] <= distanceThreshold:
                    count += 1
            # Break ties by choosing the larger index
            if count <= minCount:
                minCount = count
                resultCity = i

        return resultCity
```

1631. Path With Minimum Effort

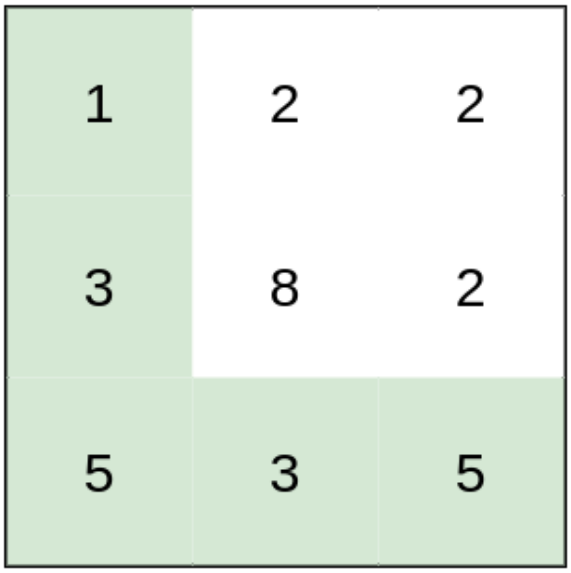
You are a hiker preparing for an upcoming hike. You are given `heights`, a 2D array of size `rows x columns`, where `heights[row][col]` represents the height of cell `(row, col)`. You are situated in the top-left cell, `(0, 0)`, and you hope to travel to the bottom-right cell, `(rows-1, columns-1)` (i.e., **0-indexed**). You can move **up**, **down**, **left**, or **right**, and you wish to find a route that requires the minimum

effort.

A route's **effort** is the **maximum absolute difference** in heights between two consecutive cells of the route.

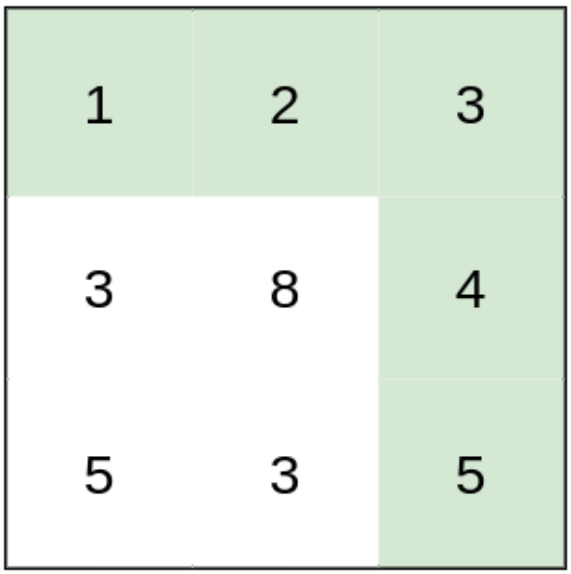
Return *the minimum **effort** required to travel from the top-left cell to the bottom-right cell.*

Example 1:



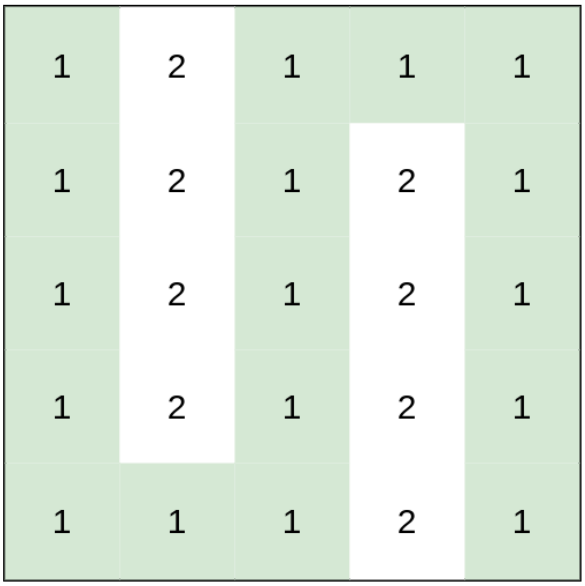

```
Input: heights = [[1,2,2],[3,8,2],[5,3,5]]
Output: 2
Explanation: The route of [1,3,5,3,5] has a maximum absolute difference of 2 in conse
This is better than the route of [1,2,2,2,5], where the maximum absolute difference i
```

Example 2:



```
Input: heights = [[1,2,3],[3,8,4],[5,3,5]]
Output: 1
Explanation: The route of [1,2,3,4,5] has a maximum absolute difference of 1 in conse
```

Example 3:




```
Input: heights = [[1,2,1,1,1],[1,2,1,2,1],[1,2,1,2,1],[1,2,1,2,1],[1,1,1,2,1]]
Output: 0
Explanation: This route does not require any effort.
```

Constraints:

- rows == heights.length
- columns == heights[i].length
- 1 <= rows, columns <= 100
- 1 <= heights[i][j] <= 10^6

```
from collections import deque
import heapq
class Solution:
    def minimumEffortPath(self, heights: List[List[int]]) -> int:
        row = len(heights)
        col = len(heights[0])
        pq = [(0,0,0)]
        dirs = [(0,1),(1,0),(0,-1),(-1,0)]
        max_diff = 0
        vis = [ [False for _ in range(col)] for _ in range(row) ]
        while pq:
            e,x,y = heapq.heappop(pq)
            max_diff = max(e,max_diff)
            if row-1 == x and y == col-1 :
                return max_diff
            if vis[x][y] == True:
                continue
            vis[x][y] = True
            for nx,ny in dirs:
                nx,ny = x+nx,y+ny
                if 0<=nx<row and 0<=ny<col:
                    effort = abs(heights[nx][ny] - heights[x][y])
                    heapq.heappush(pq,(effort,nx,ny))
        return max_diff
```

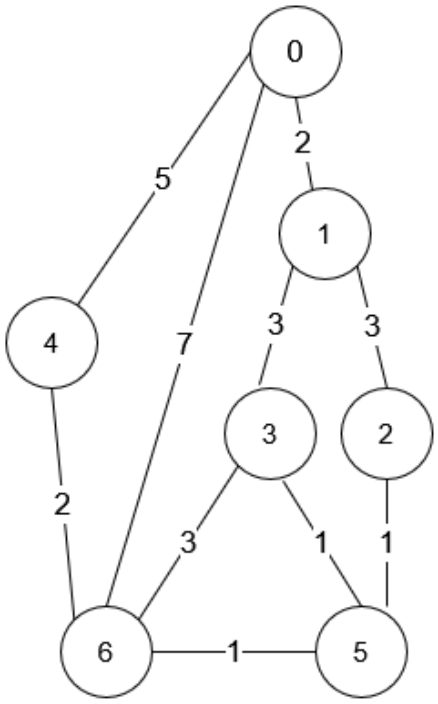
1976. Number of Ways to Arrive at Destination

You are in a city that consists of `n` intersections numbered from `0` to `n - 1` with **bi-directional** roads between some intersections. The inputs are generated such that you can reach any intersection from any other intersection and that there is at most one road between any two intersections.

You are given an integer `n` and a 2D integer array `roads` where `roads[i] = [u_i, v_i, time_i]` means that there is a road between intersections `u_i` and `v_i` that takes `time_i` minutes to travel. You want to know in how many ways you can travel from intersection `0` to intersection `n - 1` in the **shortest amount of time**.

Return *the **number of ways** you can arrive at your destination in the **shortest amount of time***. Since the answer may be large, return it **modulo** $10^9 + 7$.

Example 1:




```
Input: n = 7, 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
Output: 4
Explanation: The shortest amount of time it takes to go from intersection 0 to inters
The four ways to get there in 7 minutes are:
- 0 → 6
- 0 → 4 → 6
- 0 → 1 → 2 → 5 → 6
- 0 → 1 → 3 → 5 → 6
```

Example 2:

```
Input: n = 2, roads = [[1,0,10]]
Output: 1
Explanation: There is only one way to go from intersection 0 to intersection 1, and i
```

Constraints:

- `1 <= n <= 200`
- `n - 1 <= roads.length <= n * (n - 1) / 2`
- `roads[i].length == 3`
- $0 <= u_i, v_i <= n - 1$
- $1 <= time_i <= 10^9$
- $u_i != v_i$
- There is at most one road connecting any two intersections.

- You can reach any intersection from any other intersection.

```
from heapq import heappush,heappop
from collections import deque
class Solution:
    def countPaths(self, n: int, roads: List[List[int]]) -> int:
        MOD = 10**9 + 7
        heap = [[0,0]]
        adj = {i:[] for i in range(n)}
        dist = [float("inf") ]*n
        dist[0] = 0
        for x,y,z in roads:
            adj[x].append([y,z])
            adj[y].append([x,z])
        ways = [0]*n
        ways[0] = 1
        while heap:
            d,node = heappop(heap)
            if d > dist[node]:
                continue
            for neigh,weight in adj[node]:
                if weight+dist[node] < dist[neigh]:
                    dist[neigh]  = weight+dist[node]
                    heappush(heap,[dist[neigh],neigh])
                    ways[neigh] = ways[node]
                elif weight+dist[node] == dist[neigh]:
                    ways[neigh] = (ways[neigh]+ways[node])%MOD
        return ways[n-1]
```