DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

CHANDIGARH UNIVERSITY

Discover. Learn. Empower.

DOMAIN WINTER WINNING CAMP 2024

Student Name: Shreya Kandpal **UID:** 22BCS10079
Branch: CSE **Section/Group:** 22BCS_FL_IOT-603/A

VERY EASY

Flood Fill - link

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

CODE:

```
def floodFill(image, sr, sc, color):
    rows, cols = len(image), len(image[0])
    original_color = image[sr][sc]
    if original_color == color:
        return image

    def dfs(r, c):
        if r < 0 or r >= rows or c < 0 or c >= cols or image[r][c] != original_color:
            return
        image[r][c] = color
        dfs(r + 1, c)
        dfs(r - 1, c)
        dfs(r, c + 1)
        dfs(r, c - 1)

    dfs(sr, sc)
    return image

image1 = [[1, 1, 1], [1, 1, 0], [1, 0, 1]]
sr1, sc1, color1 = 1, 1, 2
print(floodFill(image1, sr1, sc1, color1))
```

Output

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

=== Code Execution Successful ===
```

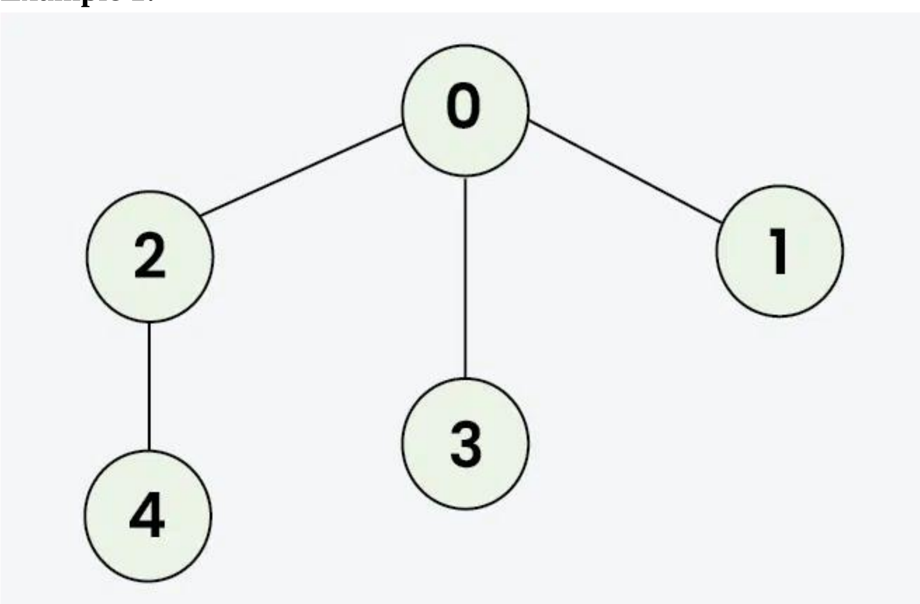
EASY

DFS of Graph

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.

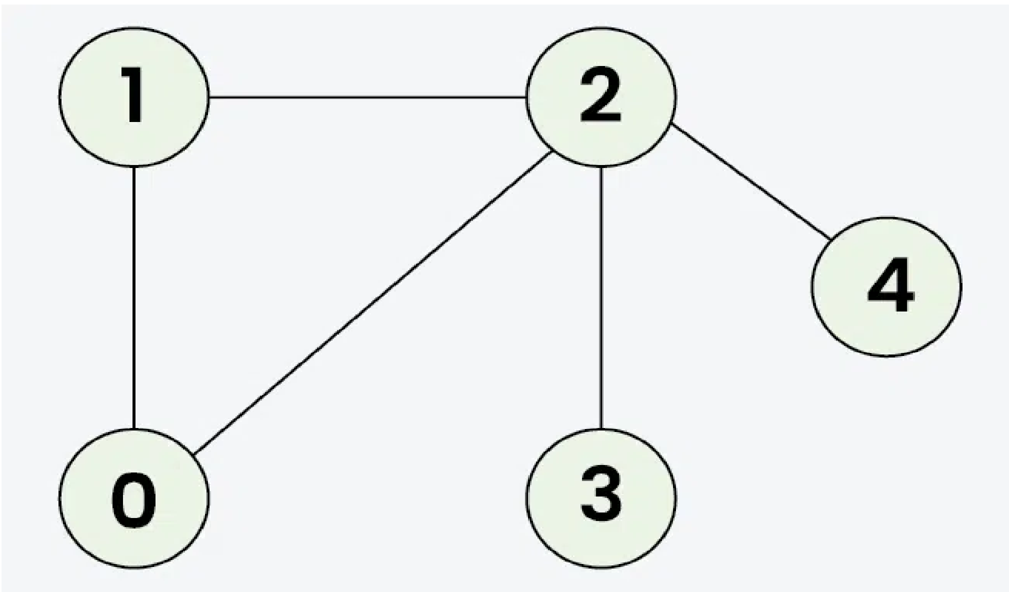
Example 1:



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

CODE:

```
def dfs_traversal(adj):
    def dfs(node, visited, result):

        visited[node] = True
        result.append(node)

        for neighbor in adj[node]:
            if not visited[neighbor]:
                dfs(neighbor, visited, result)

    visited = [False] * len(adj)
    result = []
```

```
    dfs(0, visited, result)

    return result
adj = [[2, 3, 1], [0], [0, 4], [0], [2]]
output = dfs_traversal(adj)
print(output)
```

Output

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

=== Code Execution Successful ===
```

Medium

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:

A	B	C	E
S	F	C	S
A	D	E	E

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

Output: true

Example 2:

A	B	C	E
S	F	C	S
A	D	E	E

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

Example 3:

A	B	C	E
S	F	C	S
A	D	E	E

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?

CODE:

```
def exist(board, word):
    rows, cols = len(board), len(board[0])

    def dfs(r, c, index):
        if index == len(word):
            return True
        if r < 0 or r >= rows or c < 0 or c >= cols or board[r][c] != word[index]:
            return False
        temp, board[r][c] = board[r][c], '#'

        found = (dfs(r + 1, c, index + 1) or
                 dfs(r - 1, c, index + 1) or
                 dfs(r, c + 1, index + 1) or
                 dfs(r, c - 1, index + 1))
        board[r][c] = temp
        return found
    for row in range(rows):
        for col in range(cols):
            if board[row][col] == word[0] and dfs(row, col, 0):
                return True

    return False

board1 = [["A", "B", "C", "E"],
          ["S", "F", "C", "S"],
          ["A", "D", "E", "E"]]
word1 = "ABCCED"
print(exist(board1, word1))  # Output: True
```

Output

```
True

=== Code Execution Successful ===
```

Hard

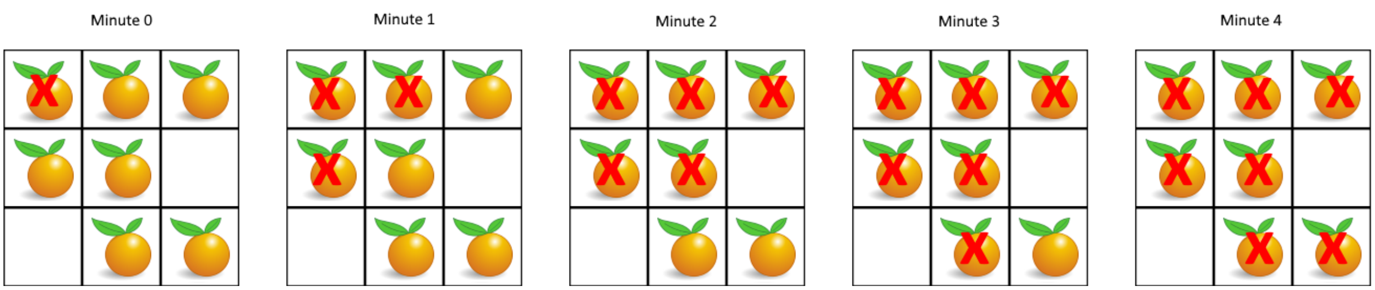
Rotting Oranges

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.

CODE:

```
from collections import deque

def orangesRotting(grid):
    rows, cols = len(grid), len(grid[0])
    queue = deque()
    fresh_count = 0
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 2:
                queue.append((r, c))
            elif grid[r][c] == 1:
                fresh_count += 1
    if fresh_count == 0:
        return 0
    minutes_passed = 0
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    while queue:
        for _ in range(len(queue)):
            x, y = queue.popleft()
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] == 1:
                    grid[nx][ny] = 2
                    queue.append((nx, ny))
                    fresh_count -= 1
        minutes_passed += 1
    return minutes_passed - 1 if fresh_count == 0 else -1

grid1 = [[2, 1, 1], [1, 1, 0], [0, 1, 1]]
print(orangesRotting(grid1))  # Output: 4
```

Output

```
4

=== Code Execution Successful ===
```

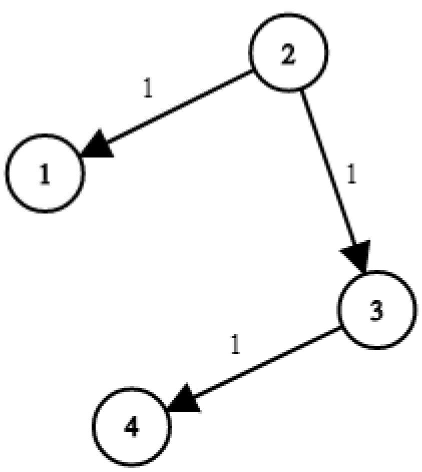
Very Hard

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] = (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.)

CODE:

```
import heapq
from collections import defaultdict
def networkDelayTime(times, n, k):
    graph = defaultdict(list)
    for u, v, w in times:
        graph[u].append((v, w))
    min_heap = [(0, k)]
    shortest_times = {}
    while min_heap:
        current_time, node = heapq.heappop(min_heap)
        if node in shortest_times:
            continue
        shortest_times[node] = current_time
        for neighbor, weight in graph[node]:
            if neighbor not in shortest_times:
                heapq.heappush(min_heap, (current_time + weight, neighbor))
    return max(shortest_times.values()) if len(shortest_times) == n else -1
times1 = [[2, 1, 1], [2, 3, 1], [3, 4, 1]]
n1, k1 = 4, 2
print(networkDelayTime(times1, n1, k1))  # Output: 2
```

Output

```
2

=== Code Execution Successful ===
```