**994. Rotting Oranges**

<https://leetcode.com/problems/rotting-oranges/>

1. **Listen**

**Problem Statement:**

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.

**Input:**

**m** x **n** 2D matrix.

The matrix is a 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.

**Goal:**

Find the minimum number of minutes that must elapse until no cell has a fresh orange.

Some oranges that are not 4-directionlly adjacent to other oranges may not become rotten.

**Return:**

return the number of minutes it takes until no cell has a fresh orange.

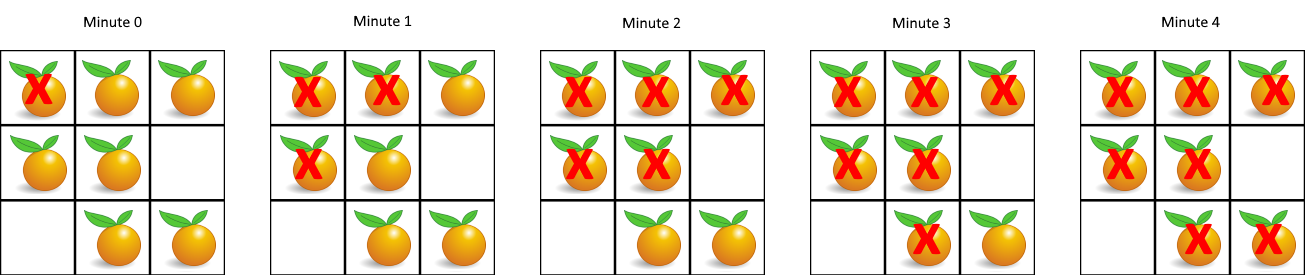
return -1 if this is impossible.

1. **Example**

**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

**Test Cases:**

* No oranges (all ‘0s’)
* All fresh oranges (all ‘1s’)
* No initial rotten oranges (all ‘0’s and ‘1’s)
* No initial fresh oranges (all ‘0’s and ‘2’s)
* All oranges become rotten (mix of 0’s, 1’s, and 2’s)
* 1x1 grid
* m and n are different
* 10x10 grid

1. **Brute Force / Optimize**

**Solution 1: Time = O(MN), Space = O(MN)**

* This problem is a perfect example of performing a **BFS** traversal.
* Each time we traverse, we look at all adjacent neighbors, and then traverse to adjacent nodes.
* The current node represents a rotten cell in the matrix, and we look at all adjacent cells looking for fresh orange cells. We make those fresh orange cells rotten and push them to the queue.
* This is in stark contrast to a DFS solution, in which we would go as deep as possible before backtracking.

**BFS in a Binary Tree is the same in a Graph**

* Performing a BFS on a Graph is just like performing a BFS on a binary tree.
* In a graph, BFS is performed by adding all nodes that are unvisited and adjacent to the current node we are on to the queue.
* In a binary tree, the ‘unvisited adjacent nodes’ are essentially just the non-null children of the node. For each node, we add their children (adjacent nodes) to the queue.

**Initializing the Queue**

* In a binary tree, we start by adding the root to the queue.
* This is synonymous by adding a starting node from a graph to the queue and marking it as visited.
* However, in this problem, we are dealing with indices.
* In addition, we have multiple possible starting points (because there can be multiple initial cells with ‘2’).
* Therefore, we need to traverse over the entire grid, and add any of the cells that have an initial value of ‘2’ to the queue.
* We don’t need to mark them as visited, because here, ‘2’ means a node has been visited, meaning they start off in a visited state.
* As an optimization (to avoid traversing more cells than needed), we also keep track of the number of fresh fruit left to process.
* We can do this by incrementing a fresh counter each time we encounter a 1 when we are iterating over the queue finding rotten oranges to add to the queue.

**BFS Traversal**

* We keep a counter for the number of iterations it will take to rotten the board, which represents the number of minutes that must elapse until no cell has a fresh orange.
* Our while loop condition can terminate on two conditions:
  + when we run out of fresh fruit
  + there are no more rotten fruit (there may or may not still be fresh fruit on the board, but they are not 4-directionally accessible to any rotten fruit)
* In a single minute, all fresh oranges that are adjacent to a rotten orange will go rotten. Therefore, we need to have every rotten orange currently in the queue attempt to rotten fresh neighbors in a single minute.
* Therefore, we take the current size of the queue (currently filled with rotten oranges that were just made rotten), and traverse over adjacent cells from the current cell.
* We pop each node from the queue (special handling must be done for the fact that we are dealing with an array of two indexes instead of just a node).
* We then look in all 4-direcitonally adjacent cells to see if there are any fresh fruit.
* If there are fresh fruit (and we are within the bounds of the matrix), we make it rotten, and add it to the queue. We also decrement the rotten counter.
* After we are done, we return the number of iterations it took to rotten the board.
* If there are still fresh fruit on the board that weren’t accessible to rotten, we return -1.

1. **Walkthrough**

initialize queue

number of fresh oranges variable

traverse over matrix

add any rotten oranges to queue

number of minutes variable

while queue is not empty and there are still fresh oranges

increment minute

for every rotten orange in the queue (that was just turned into a rotten orange)

get rotten orange indices from top of queue

for every unvisited (fresh orange) cell in 4 directions

make orange rotten

add to queue

decrement number of fresh oranges

return the number of minutes it took to rotten the board

return -1 if there are still fresh oranges on the board

We have to empty out the queue (add the extra for loop to iterate over all rotten orange) because the problem constraint shows that during each minute, all rotten oranges creep to adjacent oranges.

1. **Implement**

public int orangesRotting(int[][] grid) {

int m = grid.length, n = grid[0].length;

Queue<int[]> queue = new LinkedList<>();

int fresh = 0;

for (int i = 0; i < m; i += 1) {

for (int j = 0; j < n; j += 1) {

if (grid[i][j] == 2) queue.offer(new int[] { i, j });

else if (grid[i][j] == 1) fresh += 1;

}

}

int count = 0;

int[][] dirs = { { 1, 0 }, { -1, 0 }, { 0, 1 }, { 0, -1 } };

while (!queue.isEmpty() && fresh != 0) {

count += 1;

int sz = queue.size();

for (int i = 0; i < sz; i += 1) {

int[] rotten = queue.poll();

int r = rotten[0], c = rotten[1];

for (int[] dir : dirs) {

int x = r + dir[0], y = c + dir[1];

if (0 <= x && x < m && 0 <= y && y < n && grid[x][y] == 1) {

grid[x][y] = 2;

queue.offer(new int[] { x, y });

fresh -= 1;

}

}

}

}

return fresh == 0 ? count : -1;

}

1. **Test**