994. Rotting Oranges

Medium | Breadth-First Search | Array | Matrix

Problem Description

This problem presents a scenario where we have a grid that represents a box of oranges. The oranges can either be fresh (represented by `1`), rotten (`2`), or the cell can be empty (`0`). The challenge is to find out how long it will take for all fresh oranges to become rotten given that each minute, every fresh orange that is adjacent (up, down, left, or right) to a rotten orange will become rotten itself. If it's impossible for all oranges to become rotten (e.g., if some oranges are unreachable by the rotten ones), we are to return `-1`.

The problem expects us to determine the minimum number of minutes that must pass until there are no fresh oranges left or to assert that such a state is impossible to achieve.

Intuition

The intuition behind the solution is to use a Breadth-First Search (BFS) approach, which is well-suited for problems involving levels or minutes passing by as in this case. We can think of each minute as a level in a BFS traversal.

The process begins by scanning the grid to do two things:

1. Identify and enqueue all the initially rotten oranges (cells with a value `2`), as these will be the starting points for the spread of rot, and
2. Count the amount of fresh oranges (cells with a value `1`) because we need to keep track of when no fresh oranges are left.

The solution proceeds in rounds, where each round represents one minute. In each round, the solution:

- Increments the time counter once for all rotten oranges that will affect fresh ones in that minute.
- Dequeues a location of a rotten orange and checks its 4-directional neighbors.
- If a neighbor is a fresh orange, it becomes rotten. We decrement the count of fresh oranges and enqueue the new rotten orange's location for the next round of processing.

If there are no more fresh oranges (`cnt = 0`), the BFS is complete, and the time counter (`ans`) reflects the minimum number of minutes elapsed. If some fresh oranges were never reached (and therefore `cnt` is not zero after the BFS), it's impossible to rot all oranges, and the function returns `-1`.

Solution Approach

The implementation of the solution starts by initializing some basic variables:

- `m`, `n` to store the dimensions of the grid.
- `q`, a double-ended queue (`deque`), which is used to perform efficient pop and append operations from both ends, crucial for BFS.
- `cnt` to keep a count of the fresh oranges.

Next, we iterate through the grid to fill our queue `q` with the initial positions of the rotten oranges and count the number of fresh oranges using `cnt`. This setup phase is necessary to kickstart the BFS process.

Now, we define a variable `ans` to keep track of the time taken in minutes. We execute a loop that continues as long as there are fresh oranges (`cnt`) and there are rotten oranges in the queue to process (`q`). Each iteration of the loop represents a passing minute where potential rotting of adjacent oranges occurs.

Inside the loop, we increment `ans` for each round of minute and process all rotten oranges in the queue for that minute:

- Pop each position of a rotten orange from the front of the queue.
- Look at each of the four adjacent cells using the provided direction vectors `[[0, 1], [0, -1], [1, 0], [-1, 0]]`.
- For each adjacent cell:
  - Calculate the new coordinates `x` and `y` based on the current position `i`, `j` and the direction vector `a`, `b`.
  - Check if the adjacent cell is within the grid bounds and if it contains a fresh orange (`grid[x][y] == 1`).
  - If the adjacent cell is a fresh orange, decrement `cnt`, set that cell to `2` (rotten), and append its position to the queue `q` for processing in the next minute.

Once the BFS loop ends, we check whether there are any fresh oranges left untouched (`cnt` > 0). If there are, we return `-1`, indicating that it's impossible to rot all oranges. If `cnt` is `0`, meaning all fresh oranges have been turned rotten, we return `ans`, the total minutes elapsed.

The use of BFS ensures that we process all oranges that could potentially become rotten in a particular minute before moving on to the next minute. This closely emulates the passage of time and the spread of rot from orange to orange. The `deque` data structure allows us to efficiently enqueue new rotten oranges for subsequent processing while still keeping track of the current round of oranges being processed.

Example Walkthrough

Let's consider a 3×3 grid of oranges to illustrate the solution approach:

```
2 1 0
1 1 0
0 0 1
```

In this example:

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

First, we initialize our variables:

- `m` is 3 (number of rows).
- `n` is 3 (number of columns).
- `q` is an empty queue that will contain the positions of rotten oranges.
- `cnt` is a count of fresh oranges, which we initialize to 0 and will increment as we find fresh oranges.
- `ans` is initialized to -1, so if we do not find any fresh oranges, we will return -1.

We scan the grid and find that there is one rotten orange at position (0,0), and there are three fresh oranges. We add the position of the rotten orange to `q` and increment `cnt` as we count fresh oranges.

Now the queue `q` and `cnt` are as follows:

- `q` contains [(0, 0)]
- `cnt` is 3 (as there are 3 fresh oranges)

Next, we set `ans` to 0 as we prepare to simulate time passing.

Now, we begin our BFS loop. Since `q` is not empty and `cnt` is not zero, we proceed:

- As we enter the loop, we increment `ans` to represent the first minute passing.
- We dequeue (0, 0) from `q` and examine its neighbors (0, 1), (1, 0), as only these two are within the bounds of the grid and adjacent to it.
- Neighbor (0, 1) is a fresh orange, so we set it to rotten (`grid[0][1] = 2`), decrement `cnt` to 2, and enqueue (0, 1) for the next round.
- Neighbor (1, 0) is fresh as well, we perform the same operation: set it to rotten, decrement `cnt` to 1, and enqueue (1, 0).

The grid now looks like this:

```
2 2 0
2 1 0
0 0 1
```

And `q` has [(0, 1), (1, 0)] with `ans` being 1.

We enter the second minute:

- Increment `ans` to 2.
- We have two positions in `q` to process: (0, 1) and (1, 0).
- From (0, 1), we look at its neighbors, but none are fresh oranges.
- From (1, 0), we look at neighbor (1, 1) which is fresh and make it rotten, decrement `cnt` to 0, and enqueue (1, 1).

Now the grid and variables are:

```
2 2 0
2 2 0
0 0 1
```

- `q` contains [(1, 1)]
- `cnt` is 0 (no more fresh oranges)
- `ans` is 2.

Now we have processed all fresh oranges, so we exit the BFS loop and return `ans`, which is 2. This is the minimum number of minutes until there are no fresh oranges left. The last orange at (2, 2) could never become rotten as it's not adjacent to any rotten or potentially rotting oranges, so it is not counted.

Python Solution

```python
from collections import deque

class Solution:
    def orangesRotting(self, grid: List[List[int]]) -> int:
        # Get the number of rows and columns of the grid.
        rows, cols = len(grid), len(grid[0])

        # Initialize a queue for BFS and a counter for fresh oranges.
        queue = deque()
        fresh_count = 0

        # Go through each cell in the grid.
        for i in range(rows):
            for j in range(cols):
                # If we find a rotten orange, add its position to the queue.
                if grid[i][j] == 2:
                    queue.append((i, j))
                # If it's a fresh orange, increment the fresh_count.
                elif grid[i][j] == 1:
                    fresh_count += 1

        # Track the number of minutes passed.
        minutes_passed = 0

        # Perform BFS until the queue is empty or there are no fresh oranges left.
        while queue and fresh_count > 0:
            # Increment minutes each time we start a new round of BFS.
            minutes_passed += 1

            # Loop over all the rotten oranges at the current minute.
            for _ in range(len(queue)):
                i, j = queue.popleft()

                # Check the adjacent cells in all four directions.
                for delta_row, delta_col in [[0, 1], [0, -1], [1, 0], [-1, 0]]:
                    x, y = i + delta_row, j + delta_col

                    # If the adjacent cell has a fresh orange, rot it.
                    if 0 <= x < rows and 0 <= y < cols and grid[x][y] == 1:
                        fresh_count -= 1
                        grid[x][y] = 2
                        queue.append((x, y))

        # If there are no fresh oranges left, return the minutes passed.
        # Otherwise, return -1 because some oranges will never rot.
        return minutes_passed if fresh_count == 0 else -1
```

Java Solution

```java
class Solution {
    public int orangesRotting(int[][] grid) {
        int rows = grid.length, cols = grid[0].length;
        int freshOranges = 0; // Counter for fresh oranges
        Deque<int[]> queue = new LinkedList<>(); // Queue for rotten oranges' positions

        // Preprocess the grid, enqueue all rotten oranges and count the fresh ones
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                if (grid[i][j] == 2) { // If orange is rotten
                    queue.offer(new int[] {i, j});
                } else if (grid[i][j] == 1) { // If orange is fresh
                    freshOranges++;
                }
            }
        }

        int minutesElapsed = 0; // Time counter for rotting process
        int[] directions = {1, 0, -1, 0, 1}; // Directions for adjacent cells: right, down, left, up

        // Start BFS from initially rotten oranges
        while (!queue.isEmpty() && freshOranges > 0) {
            minutesElapsed++; // Increase time since each level of BFS represents 1 minute
            for (int i = queue.size(); i > 0; --i) { // Iterate over rotten oranges at current time
                int[] position = queue.poll();
                for (int j = 0; j < 4; ++j) { // Check all adjacent cells
                    int x = position[0] + directions[j];
                    int y = position[1] + directions[j + 1];
                    // If adjacent cell is within bounds and has a fresh orange
                    if (x >= 0 && x < rows && y >= 0 && y < cols && grid[x][y] == 1) {
                        grid[x][y] = 2; // Rot the fresh orange
                        freshOranges--; // Decrement the fresh orange count
                        queue.offer(new int[] {x, y}); // Enqueue the new rotten orange's position
                    }
                }
            }
        }

        // If there are still fresh oranges left, return -1, otherwise return elapsed time
        return freshOranges > 0 ? -1 : minutesElapsed;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <queue>
using namespace std;

class Solution {
public:
    // Function to calculate the number of minutes until no fresh orange left.
    int orangesRotting(vector<vector<int>>& grid) {
        int rows = grid.size();          // Number of rows in the grid
        int cols = grid[0].size();       // Number of columns in the grid
        int freshCount = 0;              // Counter for fresh oranges
        queue<pair<int, int>> rottenQueue; // Queue to maintain positions of rotten oranges

        // Initialize the queue with all rotten oranges and count fresh ones.
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                if (grid[i][j] == 2) {
                    rottenQueue.emplace(i, j);
                } else if (grid[i][j] == 1) {
                    ++freshCount;
                }
            }
        }

        // Time in minutes for rotting process.
        int minutesElapsed = 0;
        // Directions array that represents the 4 adjacent positions (up, right, down, left).
        vector<int> directions = {-1, 0, 1, 0, -1};

        // Processing the grid until the queue is empty or all fresh oranges are rotten.
        while (!rottenQueue.empty() && freshCount > 0) {
            ++minutesElapsed; // Increase time since we're moving to next minute

            // Process all rotten oranges at the current time.
            for (int i = rottenQueue.size(); i > 0; --i) {
                // Fetch the coordinates of the current rotten orange.
                auto position = rottenQueue.front();
                rottenQueue.pop();
                // Traverse 4 directions from the current rotten orange.
                for (int j = 0; j < 4; ++j) {
                    int newRow = position.first + directions[j];
                    int newCol = position.second + directions[j + 1];
                    // Check boundary conditions and whether the position contains a fresh orange.
                    if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && grid[newRow][newCol] == 1) {
                        --freshCount;                // Fresh orange becomes rotten
                        grid[newRow][newCol] = 2;  // Update grid to reflect rotten orange
                        rottenQueue.emplace(newRow, newCol); // Add the new rotten orange to the queue
                    }
                }
            }
        }

        // Check if there are any fresh oranges left.
        if (freshCount > 0) {
            return -1; // Not all oranges can be rotten
        } else {
            return minutesElapsed; // All oranges are rotten, return total time taken
        }
    }
};
```

Typescript Solution

```typescript
function orangesRotting(grid: number[][]): number {
    const rows = grid.length;
    const cols = grid[0].length;

    let freshOranges = 0; // Counter for fresh oranges
    const queue: [number, number][] = []; // Queue to store positions of rotten oranges

    // Populate the queue with the initial positions of rotten oranges and count fresh oranges
    for (let i = 0; i < rows; i++) {
        for (let j = 0; j < cols; j++) {
            if (grid[i][j] === 1) {
                freshOranges++;
            } else if (grid[i][j] === 2) {
                queue.push([i, j]);
            }
        }
    }

    let minutes = 0; // Counter for elapsed minutes
    // Directions for the adjacent cells (up, right, down, left)
    const directions = [1, 0, -1, 0, 1];

    // Loop until there are no more fresh oranges or the queue is empty
    while (freshOranges !== 0 && queue.length !== 0) {
        const currentLevelSize = queue.length; // Number of rotten oranges at the current minute

        // Convert adjacent fresh oranges to rotten for each rotten orange in the queue
        for (let i = 0; i < currentLevelSize; i++) {
            const [x, y] = queue.shift(); // Get the next rotten orange position

            // Check all four directions around the rotten orange
            for (let j = 0; j < 4; j++) {
                const newX = x + directions[j];
                const newY = y + directions[j + 1];

                // If the adjacent cell has a fresh orange, convert it to rotten
                if (newX >= 0 && newX < rows && newY >= 0 && newY < cols && grid[newX][newY] === 1) {
                    grid[newX][newY] = 2;
                    queue.push([newX, newY]); // Add the new rotten orange position to the queue
                    freshOranges--; // Decrease the count of fresh oranges
                }
            }
        }

        minutes++; // Increment the minutes after each level of adjacency check
    }

    // If there are any remaining fresh oranges, return -1, otherwise return the elapsed minutes
    return freshOranges !== 0 ? -1 : minutes;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(M * N)`, where `M` is the number of rows and `N` is the number of columns in the grid. This is because in the worst case, we may have to look at each cell in the grid. The while loop runs until all the rotten oranges have been processed. In the worst case, processing one rotten orange could affect 4 adjacent cells, but each cell is enqueued and dequeued only once. This gives us the worst-case scenario of processing `M * N` cells.

Space Complexity

The space complexity of the code is also `O(M * N)`. This is due to the queue which in the worst case might need to store all the cells if they are all rotten oranges. In addition to the queue, we have constant space for the variables `m`, `n`, `cnt`, and `ans`, but they do not scale with the size of the input, so the dominating factor is the space needed for the queue.