994. Rotting Oranges Medium Breadth-First Search Array Matrix **Leetcode Link**

Problem Description

(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

This problem presents a scenario where we have a grid that represents a box of oranges. The oranges can either be fresh

assert that such a state is impossible to achieve.

Intuition

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,

The intuition behind the solution is to use a Breadth-First Search (BFS) approach, which is well-suited for problems involving levels

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

- 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.
- 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
- 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.

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

processing in the next minute.

cnt to keep a count of the fresh oranges.

oranges, and the function returns -1.

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:

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

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

 \circ 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).

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

Example Walkthrough Let's consider a 3×3 grid of oranges to illustrate the solution approach: 1 2 1 0

First, we initialize our variables:

• m is 3 (number of rows).

2 represents a rotten orange.

• 1 represents a fresh orange.

Ø represents an empty cell.

oranges being processed.

3 0 0 1

In this example:

• n is 3 (number of columns). q is an empty queue that will contain the positions of rotten oranges.

• ans is initialized to -1, so if we do not find any fresh oranges, we will return -1.

• cnt is a count of fresh oranges, which we initialize to 0 and will increment as we find fresh oranges.

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.

adjacent to it.

The grid now looks like this:

We enter the second minute:

Increment ans to 2.

1 2 2 0

2 2 2 0

3 0 0 1

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

Now the queue q and cnt are as follows:

As we enter the loop, we increment ans to represent the first minute passing.

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

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

• 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).

• 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

3 0 0 1

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

 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:

From (0, 1), we look at its neighbors, but none are fresh oranges.

We have two positions in q to process: (0, 1) and (1, 0).

• 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

class Solution:

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

31

32

33

34

35

36

37

44

45

46

47

cnt is 0 (no more fresh oranges)

potentially rotting oranges, so it is not counted.

queue = deque()

fresh_count = 0

for i in range(rows):

minutes_passed = 0

• q contains [(1, 1)]

Python Solution from collections import deque

rows, cols = len(grid), len(grid[0])

Go through each cell in the grid.

if grid[i][j] == 2:

elif grid[i][j] == 1:

Track the number of minutes passed.

for _ in range(len(queue)):

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

i, j = queue.popleft()

fresh_count += 1

queue.append((i, j))

for j in range(cols):

def orangesRotting(self, grid: List[List[int]]) -> int:

Get the number of rows and columns of the grid.

Initialize a queue for BFS and a counter for fresh oranges.

If we find a rotten orange, add its position to the queue.

If it's a fresh orange, increment the fresh_count.

Check the adjacent cells in all four directions.

If there are no fresh oranges left, return the minutes passed.

// Function to calculate the number of minutes until no fresh orange left.

// Initialize the queue with all rotten oranges and count fresh ones.

// Number of rows in the grid

// Counter for fresh oranges

queue<pair<int, int>> rottenQueue; // Queue to maintain positions of rotten oranges

// Directions array that represents the 4 adjacent positions (up, right, down, left).

// Processing the grid until the queue is empty or all fresh oranges are rotten.

++minutesElapsed; // Increase time since we're moving to next minute

// Fetch the coordinates of the current rotten orange.

// Traverse 4 directions from the current rotten orange.

int newCol = position.second + directions[j + 1];

int newRow = position.first + directions[j];

// Number of columns in the grid

int orangesRotting(vector<vector<int>>& grid) {

for (int j = 0; j < cols; ++j) {

rottenQueue.emplace(i, j);

} else if (grid[i][j] == 1) {

if (grid[i][j] == 2) {

++freshCount;

// Time in minutes for rotting process.

rottenQueue.pop();

vector<int> directions = $\{-1, 0, 1, 0, -1\}$;

while (!rottenQueue.empty() && freshCount > 0) {

// Process all rotten oranges at the current time.

for (int i = rottenQueue.size(); i > 0; --i) {

auto position = rottenQueue.front();

for (int j = 0; j < 4; ++j) {

int cols = grid[0].size();

for (int i = 0; i < rows; ++i) {

int rows = grid.size();

int minutesElapsed = 0;

int freshCount = 0;

Otherwise, return -1 because some oranges will never rot.

x, y = i + delta_row, j + delta_col

fresh count -= 1

return minutes_passed if fresh_count == 0 else -1

int rows = grid.length, cols = grid[0].length;

int freshOranges = 0; // Counter for fresh oranges

queue.append((x, y))

grid[x][y] = 2

for delta_row, delta_col in [[0, 1], [0, -1], [1, 0], [-1, 0]]:

if 0 <= x < rows and 0 <= y < cols and grid[x][y] == 1:</pre>

If the adjacent cell has a fresh orange, rot it.

24 25 # Perform BFS until the queue is empty or there are no fresh oranges left. 26 while queue and fresh_count > 0: 27 # Increment minutes each time we start a new round of BFS. 28 minutes_passed += 1 29 30 # Loop over all the rotten oranges at the current minute.

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

38 39 40 41 42 43

Java Solution

1 class Solution {

Deque<int[]> queue = new LinkedList<>(); // Queue for rotten oranges' positions 6 // Preprocess the grid, enqueue all rotten oranges and count the fresh ones for (int i = 0; i < rows; ++i) { 8 for (int j = 0; j < cols; ++j) { 9 if (grid[i][j] == 2) { // If orange is rotten 10 11 queue.offer(new int[] {i, j}); } else if (grid[i][j] == 1) { // If orange is fresh 12 13 fresh0ranges++; 14 15 16 17 18 int minutesElapsed = 0; // Time counter for rotting process 19 int[] directions = $\{1, 0, -1, 0, 1\}$; // Directions for adjacent cells: right, down, left, up 20 21 // Start BFS from initially rotten oranges 22 while (!queue.isEmpty() && freshOranges > 0) { minutesElapsed++; // Increase time since each level of BFS represents 1 minute 23 24 for (int i = queue.size(); i > 0; --i) { // Iterate over rotten oranges at current time 25 int[] position = queue.poll(); 26 for (int j = 0; j < 4; ++j) { // Check all adjacent cells 27 int x = position[0] + directions[j]; 28 int y = position[1] + directions[j + 1]; // If adjacent cell is within bounds and has a fresh orange 29 30 if $(x >= 0 \&\& x < rows \&\& y >= 0 \&\& y < cols \&\& grid[x][y] == 1) {$ 31 grid[x][y] = 2; // Rot the fresh orange 32 freshOranges--; // Decrement the fresh orange count 33 queue.offer(new int[] {x, y}); // Enqueue the new rotten orange's position 34 35 36 37 38 39 // If there are still fresh oranges left, return -1, otherwise return elapsed time return freshOranges > 0 ? -1 : minutesElapsed; 40 41 42 43 C++ Solution

37 38 39 40 41 42

#include <vector>

using namespace std;

2 #include <queue>

5 class Solution {

public:

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

```
43
                         // Check boundary conditions and whether the position contains a fresh orange.
                         if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && grid[newRow][newCol] == 1) {</pre>
 44
 45
                             --freshCount;
                                                         // Fresh orange becomes rotten
                             grid[newRow][newCol] = 2; // Update grid to reflect rotten orange
 46
 47
                             rottenQueue.emplace(newRow, newCol); // Add the new rotten orange to the queue
 48
 49
 50
 51
 52
 53
             // Check if there are any fresh oranges left.
 54
             if (freshCount > 0) {
 55
                 return -1; // Not all oranges can be rotten
 56
             } else {
                 return minutesElapsed; // All oranges are rotten, return total time taken
 57
 58
 59
 60 };
 61
Typescript Solution
    function orangesRotting(grid: number[][]): number {
         const rows = grid.length;
         const cols = grid[0].length;
  4
         let freshOranges = 0; // Counter for fresh oranges
  5
         const queue: [number, number][] = []; // Queue to store positions of rotten oranges
  6
  8
         // Populate the queue with the initial positions of rotten oranges and count fresh oranges
         for (let i = 0; i < rows; i++) {</pre>
  9
             for (let j = 0; j < cols; j++) {</pre>
 10
                 if (grid[i][j] === 1) {
 11
 12
                     freshOranges++;
                 } else if (grid[i][j] === 2) {
 13
 14
                     queue.push([i, j]);
 15
 16
 17
 18
 19
         let minutes = 0; // Counter for elapsed minutes
 20
        // Directions for the adjacent cells (up, right, down, left)
 21
         const directions = [1, 0, -1, 0, 1];
 22
 23
        // Loop until there are no more fresh oranges or the queue is empty
 24
        while (freshOranges !== 0 && queue.length !== 0) {
 25
             const currentLevelSize = queue.length; // Number of rotten oranges at the current minute
 26
 27
             // Convert adjacent fresh oranges to rotten for each rotten orange in the queue
             for (let i = 0; i < currentLevelSize; i++) {</pre>
 28
 29
                 const [x, y] = queue.shift(); // Get the next rotten orange position
 30
 31
                 // Check all four directions around the rotten orange
 32
                 for (let j = 0; j < 4; j++) {
 33
                     const newX = x + directions[j];
 34
                     const newY = y + directions[j + 1];
 35
 36
                     // If the adjacent cell has a fresh orange, convert it to rotten
 37
                     if (newX >= 0 && newX < rows && newY >= 0 && newY < cols && grid[newX][newY] === 1) {</pre>
 38
                         grid[newX][newY] = 2;
                         queue.push([newX, newY]); // Add the new rotten orange position to the queue
 39
                         freshOranges--; // Decrease the count of fresh oranges
 40
 41
 43
 44
 45
             minutes++; // Increment the minutes after each level of adjacency check
 46
 47
 48
        // If there are any remaining fresh oranges, return -1, otherwise return the elapsed minutes
```

50 51

Time and Space Complexity

return freshOranges !== 0 ? -1 : minutes;

49

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.

The space complexity of the code is also 0(M * N). This is due to the queue which in the worst case might need to store all the cells

Space Complexity

Time Complexity

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.