CS 3201 Algorithm Design 1/2020

6118135 Jan Duldhardt

Term Project

# Rotting Oranges

**Content**

## Introduction

The reason why I have selected this problem is because last year I applied for some internships at big tech companies and I knew that they are having this kind of data structures and algorithm coding interview. At that time I was still taking the course Data Structures and Algorithms so I was very inexperienced with how to solve this kind of problem. At that time I had no idea about something like BFS so solving this problem was almost impossible for me at that time, I could not wrap my head around it.

So after completing Data Structures and Algorithms and learning how to implement BFS in this Algorithm Design course I wanted to give this problem another try and see if what I have learned so far can help me solve the question I struggled one year ago with.

## Question

**994. Rotting Oranges**

Medium 2564 198 Add to List Share

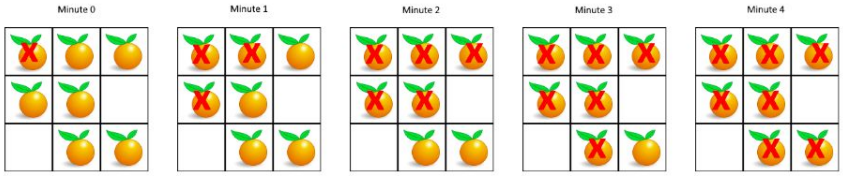
In a given grid, each cell can have one of three values:

- the value `0` representing an empty cell;
- the value `1` representing a fresh orange;
- the value `2` representing a rotten orange.

Every minute, any fresh orange that is adjacent (4-directionally) 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` instead.

**Example 1:**




```
Input: [[2,1,1],[1,1,0],[0,1,1]]
Output: 4
```

**Example 2:**

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

I think the description of the question is very clear.
We need to return the number of minutes it takes for all oranges to be rotten if it is not possible because an orange is not adjacent to any rotten orange. We have to return -1.
Also in case there is no fresh orange to begin with we return 0 (=> 0 minutes).

**The approach**

Even though my code should be easy to understand I will quickly go through it.

First we have the **rottenlist** which contains the points of oranges which are rotten as a tuple (row,col) and a counter **minutes** to count the iterations.

Then we go through the grid once to get all the initial rotten oranges and save them to our **rottenlist**.

After that we have our loop **while rottenlist is not empty** we want to replace our current rottenlist with the new rottenlist which are the adjacent fresh oranges of all the rotten oranges in our rottenlist.

For that I just define the method **getAdj** which gets the grid and the rottenlist as a parameter.

In that method I just check if the adjacent positions are valid and not out of bounds and after that check if the adjacent is a fresh orange, and if so the orange is now rotten and appended to the rottenlist for the next iteration.

Now let's go back to the main while **rottenlist != []** after we get the new rottenlist we need to check if this rottenlist is not empty and if so we increase the counter.

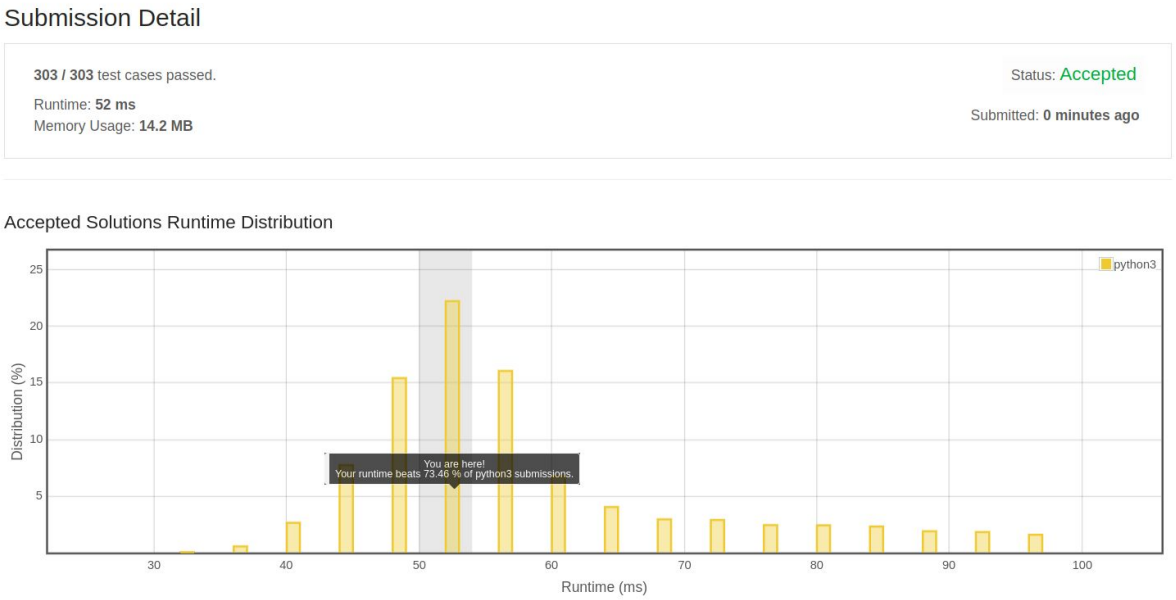
The last thing we have to do before returning **minutes** is to check whether or not there is still a fresh orange in the grid left, this can only happen if there was no connection between a rotten orange and the remaining fresh one as in **Example 2**.

If no fresh one is found we can return **minutes**.

**Code**

```
def getAdj(grid, rotten):
    adj = []
    for point in rotten:
        row = point[0]
        col = point[1]
        directions = [(1,0),(0,1),(-1,0),(0,-1)]
        for direction in directions:
            newrow = row + direction[0]
            newcol = col + direction[1]
            if newrow >= 0 and newrow < len(grid) and newcol >= 0 and newcol <
len(grid[0]) and grid[newrow][newcol] == 1:
                point = newrow, newcol
                grid[newrow][newcol] = 2
                adj.append(point)
    return adj


class Solution:
    def orangesRotting(self, grid: List[List[int]]) -> int:
        rottenlist = []
        minutes = 0


        for i in range(len(grid)):
            for j in range(len(grid[i])):
                if grid[i][j] == 2:
                    rottenlist.append((i,j))

        while rottenlist != []:
            rottenlist = getAdj(grid,rottenlist)

            if rottenlist:
                minutes += 1

        for i in range(len(grid)):
            for j in range(len(grid[i])):
                if grid[i][j] == 1:
                    return -1

        return minutes
```

## Analysis




As we can see in the statistics my running time is better than 73% of other peoples python code and my memory usage is better than 100% of peoples python code.

But if we analyse the asymptotic runtime complexity of my solution we can notice that it is actually the optimal solution.
Let's say our grid length (row * col) is denoted as **n**
Then we have **O(n)** and because every change is saved in the initial grid we get a space complexity of **O(1).**

**Time complexity: O(n) where n is the length of the grid (row * col)**
**Space complexity: O(1)**

## Comments

This question actually comes from a real interview question from **Amazon**. In my opinion the question in general is not too difficult if you are familiar with BFS but in this case there are a lot of edge cases that need to be considered. Also the space complexity because in class we learned to always use a new class for the state but in this case we can just use the grid to save the information. All in all this question was challenging to get the optimal solution and I am happy that the knowledge I have learned the past year about Data Structures and Algorithms and Algorithm Design has bared fruits.

## References

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