Average Rating: 4.35 (106 votes) Premium

Solution

Approach #1 (Brute Force) [Time Limit Exceeded]

The brute force approach is simple, we just implement a breadth-first search from each empty room to its nearest gate.

While we are doing the search, we use a 2D array called `distance` to keep track of the distance from the starting point. It also implicitly tell us whether a position had been visited so it won't be inserted into the queue again.

```
private static final int EMPTY = Integer.MAX_VALUE;
private static final int GATE = 0;
private static final int WALL = -1;
private static final List<int[]> DIRECTIONS = Arrays.asList(
        new int[] { 1,  0},
        new int[] {-1,  0},
        new int[] { 0,  1},
        new int[] { 0, -1}
);

public void wallsAndGates(int[][] rooms) {
    if (rooms.length == 0) return;
    for (int row = 0; row < rooms.length; row++) {
        for (int col = 0; col < rooms[0].length; col++) {
            if (rooms[row][col] == EMPTY) {
                rooms[row][col] = distanceToNearestGate(rooms, row, col);
            }
        }
    }
}

private int distanceToNearestGate(int[][] rooms, int startRow, int startCol) {
    int m = rooms.length;
    int n = rooms[0].length;
    int[][] distance = new int[m][n];
    Queue<int[]> q = new LinkedList<>();
    q.add(new int[] { startRow, startCol });
    while (!q.isEmpty()) {
        int[] point = q.poll();
        int row = point[0];
        int col = point[1];
        for (int[] direction : DIRECTIONS) {
            int r = row + direction[0];
            int c = col + direction[1];
            if (r < 0 || c < 0 || r >= m || c >= n || rooms[r][c] == WALL
                    || distance[r][c] != 0) {
                continue;
            }
            distance[r][c] = distance[row][col] + 1;
            if (rooms[r][c] == GATE) {
                return distance[r][c];
            }
            q.add(new int[] { r, c });
        }
    }
    return Integer.MAX_VALUE;
}
```

Complexity analysis

- Time complexity : $O(m^2n^2)$. For each point in the $m \times n$ size grid, the gate could be at most $m \times n$ steps away.
- Space complexity : $O(mn)$. The space complexity depends on the queue's size. Since we won't insert points that have been visited before into the queue, we insert at most $m \times n$ points into the queue.

Approach #2 (Breadth-first Search) [Accepted]

Instead of searching from an empty room to the gates, how about searching the other way round? In other words, we initiate breadth-first search (BFS) from all gates at the same time. Since BFS guarantees that we search all rooms of distance *d* before searching rooms of distance *d* + 1, the distance to an empty room must be the shortest.

```
private static final int EMPTY = Integer.MAX_VALUE;
private static final int GATE = 0;
private static final List<int[]> DIRECTIONS = Arrays.asList(
        new int[] { 1,  0},
        new int[] {-1,  0},
        new int[] { 0,  1},
        new int[] { 0, -1}
);

public void wallsAndGates(int[][] rooms) {
    int m = rooms.length;
    if (m == 0) return;
    int n = rooms[0].length;
    Queue<int[]> q = new LinkedList<>();
```

```
    for (int row = 0; row < m; row++) {
        for (int col = 0; col < n; col++) {
            if (rooms[row][col] == GATE) {
                q.add(new int[] { row, col });
            }
        }
    }
    while (!q.isEmpty()) {
        int[] point = q.poll();
        int row = point[0];
        int col = point[1];
        for (int[] direction : DIRECTIONS) {
            int r = row + direction[0];
            int c = col + direction[1];
            if (r < 0 || c < 0 || r >= m || c >= n || rooms[r][c] != EMPTY) {
                continue;
            }
            rooms[r][c] = rooms[row][col] + 1;
            q.add(new int[] { r, c });
        }
    }
}
```

Complexity analysis

- Time complexity : $O(mn)$.

  If you are having difficulty to derive the time complexity, start simple.

  Let us start with the case with only one gate. The breadth-first search takes at most $m \times n$ steps to reach all rooms, therefore the time complexity is $O(mn)$. But what if you are doing breadth-first search from k gates?

  Once we set a room's distance, we are basically marking it as visited, which means each room is visited at most once. Therefore, the time complexity does not depend on the number of gates and is $O(mn)$.

- Space complexity : $O(mn)$. The space complexity depends on the queue's size. We insert at most $m \times n$ points into the queue.

Report Article Issue

gregpen ★ 276 Last Edit: September 14, 2018 3:25 PM

I think what some folks are missing in this second solution is that each gate is not fully searched before moving on to a new gate. Each gate only looks at the areas within 1 space before we check the next gate. So each area within one space of the gates are checked for rooms and these rooms are marked, then added to the queue. Once all gates are checked, each new space is checked, and so forth. So, once a room gets hit, it has to be from the closest gate.

275 Show 6 replies Reply

leetcodefan ★ 3216 February 9, 2019 3:35 PM

Approach 2 is eye-opening.

157 Show 3 replies Reply

btjd ★ 1170 February 23, 2019 11:23 AM

The phrase "we initiate breadth-first search (BFS) from all gates at the same time" in approach 2 threw me off initially until I started working through the example step by step and then it made sense. I still don't think "at the same time" is an accurate description (because at least for me it just makes me think parallelism/concurrency) and might cause some confusion but to be fair I can't come up with a better way to explain it.

76 Show 4 replies Reply

leetbunny ★ 105 April 21, 2019 10:37 PM

It's smart to enqueue all the 0s at the same time, rather than empty the queue of one 0 and enqueue the next. So whenever an empty room is reached, it must be from the closest gate.

35 Reply

calvinchankf ★ 6218 November 22, 2018 7:43 AM

Acutally i think the DFS approach is a lot easier to understand, e.g. in golang

```
func wallsAndGates(rooms [][]int) {
    for i := 0; i < len(rooms); i++ {
        for j := 0; j < len(rooms[0]); j++ {
            if rooms[i][j] == 0 {
                // to speed up the calculation
                // instead of calculating from the INF, calculating from 0 can avoid redundant calculation
                dfs(rooms, i, j, 0)
            }
```

Read More

39 Show 10 replies Reply

haoyangfan ★ 1717 August 11, 2019 9:38 AM

well, this way of defining directions is too verbose ...

```
private static final List<int[]> DIRECTIONS = Arrays.asList(
        new int[] { 1,  0},
        new int[] {-1,  0},
        new int[] { 0,  1},
        new int[] { 0, -1}
```

}

we can change it to one line

Read More

12 Show 2 replies Reply

FLAGbigoffer ★ 3418 September 11, 2017 4:45 PM

Good job! The second solution is beautiful and definitely right. Because we use BFS in each GATE, each time we move one step and mark the shortest in one cell. As long as this cell is marked, we don't have to visit and re-mark it again. That's it. Scan the whole matrix one time. Time complexity definitely O(m * n).

7 Reply

lcgt ★ 206 Last Edit: August 10, 2019 8:32 PM

Damnit... I started doing DFS from every empty room thinking its very similar to the other Leetcode problem "Word Search" (with the difference that if you land on a room that already has some distance value, you just return currentDist + alreadyComputedValue). So glad I did not try to make my solution work and instead just looked at the solution to realize my approach was fundamentally wrong....

4 Show 3 replies Reply

liu971 ★ 19 Last Edit: October 24, 2018 7:47 PM

Hi, isn't the time complexity O(m * n * k) where k is the number of gates.

8 Show 4 replies Reply

jdrogin ★ 1201 October 25, 2016 2:37 PM

I think it is still O(m * n) because
step 1: you iterate over all rows and columns giving m*n operations each of O(1).*
*step 2: then you start at each gate and do 4 directions, adding in other cells until you have looked at every cell. The key here is that you will only look at each cell once during this process, so again you have O(m*n) it doesn't matter how many initial gates you have you will still iterate over every cell once during this process.*

The result of these 2 steps is O(m*n). Number of gates doesn't matter because you are marking the cells completed as your process them to ensure you process each cell once regardless of what was in your initial queue of starting points.

3 Reply

‹ 1 2 3 4 5 6 7 ›