<https://leetcode.ca/all/286.html>

You are given an m x n grid rooms initialized with these three possible values.

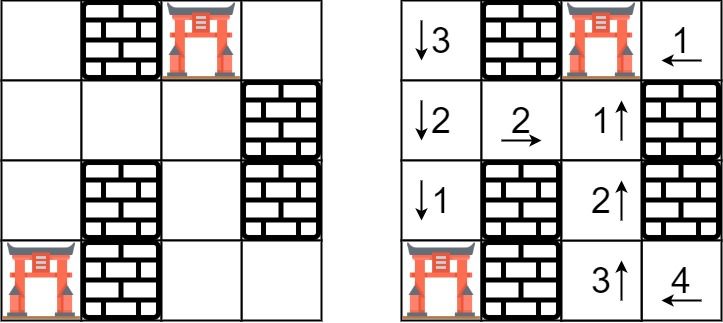
-1 A wall or an obstacle.

0 A gate.

INF Infinity means an empty room. We use the value 2^31 - 1 = 2147483647 to represent INF as you may assume that the distance to a gate is less than 2147483647.

Fill each empty room with the distance to *its nearest gate*. If it is impossible to reach a gate, it should be filled with INF.

**Example 1:**



**Input:**

rooms = [[2147483647,-1,0,2147483647],

[2147483647,2147483647,2147483647,-1],

[2147483647,-1,2147483647,-1],

[0,-1,2147483647,2147483647]]

**Output:**

[[3,-1,0,1],

[2,2,1,-1],

[1,-1,2,-1],

[0,-1,3,4]]

**Example 2:**

**Input:**

rooms = [[-1]]

**Output:**

[[-1]]

**Constraints:**

m == rooms.length

n == rooms[i].length

1 <= m, n <= 250

rooms[i][j] is -1, 0, or 2^31 - 1.

**Attempt 1: 2023-10-8**

**Solution 1: BFS + Level order traversal (10min, no extra space visited 2D array needed, just need + 1 based on current )**

class Solution {

    public void wallsAndGates(int[][] rooms) {

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

        int m = rooms.length;

        int n = rooms[0].length;

        for(int i = 0; i < m; i++) {

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

                if(rooms[i][j] == 0) {

                    q.offer(new int[] {i, j});

                }

            }

        }

        int[] dx = new int[] {0, 0, 1, -1};

        int[] dy = new int[] {1, -1, 0, 0};

        int level = 0;

        while(!q.isEmpty()) {

            int size = q.size();

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

                int[] cur = q.poll();

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

                    int new\_x = cur[0] + dx[k];

                    int new\_y = cur[1] + dy[k];

                        rooms[new\_x][new\_y] = rooms[cur[0]][cur[1]] + 1;

                        q.offer(new int[] {new\_x, new\_y});

                    }

                }

            }

        }

    }

    public static void main(String[] args) {

        Solution so = new Solution();

        int INF = Integer.MAX\_VALUE;

        int[][] grid = new int[][] {{INF, -1, 0, INF}, {INF, INF, INF, -1}, {INF, -1, INF, -1}, {0, -1, INF, INF}};

        so.wallsAndGates(grid);

        System.out.println("done");

    }

}

Time Complexity: O(M\*N)

Space Complexity: O(M\*N)

**Refer to Deepseek**

For LeetCode 286 "Walls and Gates", the recommended approach is to use **Breadth-First Search (BFS)** starting from all gates at once. This method efficiently guarantees the shortest path in an unweighted grid.

Here is a Java solution using the multi-source BFS approach:

import java.util.\*;

public class Solution {

public void wallsAndGates(int[][] rooms) {

// Check for empty grid

if (rooms == null || rooms.length == 0 || rooms[0].length == 0) return;

int m = rooms.length;

int n = rooms[0].length;

// Directions for moving up, down, left, and right

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

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

// Add all gates to the queue

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

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

if (rooms[i][j] == 0) {

queue.offer(new int[]{i, j});

}

}

}

// Perform BFS

while (!queue.isEmpty()) {

int[] cell = queue.poll();

int row = cell[0];

int col = cell[1];

for (int[] dir : dirs) {

int newRow = row + dir[0];

int newCol = col + dir[1];

// Check if the new cell is within bounds and is an empty room

// Update the distance and add the new cell to the queue

rooms[newRow][newCol] = rooms[row][col] + 1;

queue.offer(new int[]{newRow, newCol});

}

}

}

}

}

**🔍 How the BFS Solution Work**s

The key to this problem is to **start from the gates (0s)**, not the empty rooms. The BFS expands in "waves" from all gates simultaneously, ensuring that the first time an empty room is reached, it's via the shortest path.

**Initialization**: The algorithm first finds all gates (cells with value 0) and adds their coordinates to a queue.

**Processing**: For each cell dequeued, it checks all four adjacent cells (up, down, left, right).

**Updating Rooms**: If an adjacent cell is a valid empty room (Integer.MAX\_VALUE), its distance is updated to current cell's value + 1, and it's added to the queue to propagate the distance further.

**💡 Key Points to Not**e

**Time Complexity**: **O(m\*n)**, where m is the number of rows and n is the number of columns. Each cell is processed at most once.

**Space Complexity**: **O(m\*n)** in the worst case, due to the BFS queue storage.

**Why BFS?** For an unweighted grid like this, BFS naturally finds the shortest path. A DFS approach would not guarantee this and could be less efficient.

**📝 Alternative DFS Approac**h

While BFS is generally preferred, a DFS solution is also possible. The idea is to iterate through the grid, and for each gate, perform a DFS that updates adjacent rooms only if the new distance is shorter. However, this approach may not be as intuitive for finding shortest paths and can be less efficient in some cases compared to multi-source BFS.

I hope this explanation helps you understand the solution. Would you like me to walk through a small example to see how the BFS processes the grid step-by-step?

**Refer to Kimi**

Here is the standard multi-source BFS solution for LeetCode 286 “Walls and Gates” (Java).

Time: O(m·n) – each cell is enqueued at most once.

Space: O(m·n) for the queue.

import java.util.\*;

public class Solution {

private static final int EMPTY = Integer.MAX\_VALUE;

private static final int GATE = 0;

private static final int WALL = -1;

private static final int[][] DIRS = {{1,0},{-1,0},{0,1},{0,-1}};

public void wallsAndGates(int[][] rooms) {

if (rooms == null || rooms.length == 0 || rooms[0].length == 0) return;

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

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

// 1. enqueue all gates at once

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

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

if (rooms[i][j] == GATE) q.offer(new int[]{i, j});

}

}

// 2. multi-source BFS

while (!q.isEmpty()) {

int[] cur = q.poll();

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

for (int[] d : DIRS) {

int nr = r + d[0], nc = c + d[1];

if (nr < 0 || nr >= m || nc < 0 || nc >= n || rooms[nr][nc] != EMPTY) continue;

rooms[nr][nc] = rooms[r][c] + 1; // distance from nearest gate

q.offer(new int[]{nr, nc});

}

}

}

}

How it works

Every gate (0) is a BFS source.

We propagate distance layer-by-layer to all reachable EMPTY (Integer.MAX\_VALUE) cells, overwriting them with the shortest distance.

Walls (-1) are simply skipped.

**Refer to**

[L994.Rotting Oranges (Ref.286,2101)](note://2EAD142D528B482F821654DFDE49CC4D)