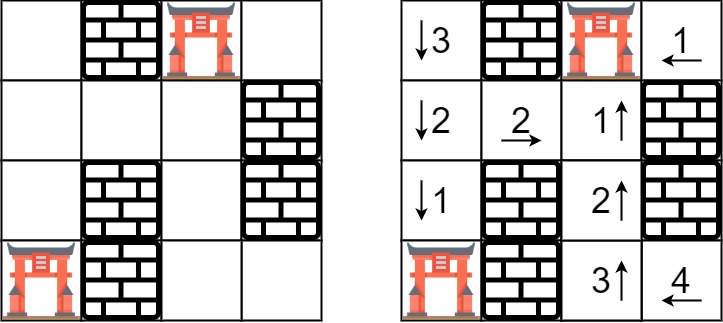
# Question

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 231 - 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]]

**Example 3:**

**Input:** rooms = [[2147483647]]

**Output:** [[2147483647]]

**Example 4:**

**Input:** rooms = [[0]]

**Output:** [[0]]

**Constraints:**

* m == rooms.length
* n == rooms[i].length
* 1 <= m, n <= 250
* rooms[i][j] is -1, 0, or 231 - 1.

# 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)*O*(*m*2*n*2). For each point in the m \times n*m*×*n* size grid, the gate could be at most m \times n*m*×*n* steps away.
* Space complexity : O(mn)*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*m*×*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)*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*m*×*n* steps to reach all rooms, therefore the time complexity is O(mn)*O*(*mn*). But what if you are doing breadth-first search from k*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)*O*(*mn*).

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