2257. Count Unguarded Cells in the Grid

Simulation

guard's line of sight. The key things to keep in mind are:

Problem Description

Array

Matrix

Medium

This LeetCode problem requires us to calculate the number of unoccupied cells in a 0-indexed m x n grid that are not guarded by any guards and not blocked by walls. We are given two lists, guards and walls, where each element is a pair of coordinates representing the location of a guard or a wall, respectively. A guard can see and thus guard all cells in the four cardinal directions (north, east, south, or west) unless there is a wall or another guard obstructing the view. Cells are considered guarded if at least

one guard can see them. Our goal is to count how many cells are left unguarded, considering that walls and guards themselves occupy some of the cells. Intuition

The solution to this problem is based on simulating the guards' line of sight and marking cells they can "see" as guarded. By

creating a grid, we can simulate the guards' lines of sight in each of the four cardinal directions. We can iterate through each

direction from the guard's position until we either reach the grid's boundary, encounter another guard, or a wall, which blocks the

• Guards and walls occupy some cells, so these should be marked differently to indicate they are not empty and cannot be guarded by other guards. • We need to check the visibility in all directions from each guard until we are obstructed or out of bounds. • After marking all the guarded cells, anything left as unmarked is unguarded, and we just need to count those cells.

- Solution Approach
- The implementation of the solution follows a step-by-step approach which relies on simulation and proper labeling of the cells

As we move through cells, we change their status to 1 to signify that they are guarded.

matrix, where m = n = 3. Imagine our grid looks something like this:

Here's a walkthrough of the algorithm: Initialization: We start by creating a grid g sized m x n, which will serve as a representation of our problem space. The value

guarded.

within the grid.

of each cell in g initially is 0, which stands for unguarded and unoccupied by either a guard or a wall. Marking Guards and Walls: We loop through the guards and walls arrays, marking the corresponding cells in our grid g as 2.

This 2 is a special label indicating cells that are occupied by either a guard or a wall and cannot be further guarded.

are unoccupied and unguarded.

iterative logic and condition checking.

'G' representing the guard and 'W' the wall:

- Guarding Cells: Next, we iterate over all the guard positions. Since guards can see in the four cardinal directions, we simulate this visibility. To do this, we use the dirs tuple which defines the directional steps for north, east, south, and west. For each
- direction: We start from the guard's position and move cell by cell in the current direction. • The movement continues until we reach a cell that is either out of bounds, marked as 2 (a wall or another guard), or has already been
- the grid from the guard's current position. Counting Unguarded Cells: Finally, we go through the entire grid g and count all cells still marked as 0, which denotes they

This simulation uses the pairwise iterator from the dirs tuple to get the direction vectors, which define our movement across

Therefore, the algorithm employs a simulation method and utilizes a matrix to keep track of the state of each cell accurately. This is an efficient way to solve the problem as it does not involve any complex data structures or algorithms, just straightforward

Example Walkthrough Let's walk through a simple example to illustrate the solution approach. Suppose we have a 3×3 grid represented by an m x n

located in the cell at row 1 column 1 (use 0-indexing), and the wall is located at row 0 column 2. The grid now looks like this, with

Now, let's say we have one guard and one wall, with the guards list as [(1, 1)] and the walls list as [(0, 2)]. The guard is

][][W] [][][]

Following the solution approach:

• Initialization: We create an initial grid g of size 3×3 with all cells initialized to 0 to represent unguarded cells.

• Marking Guards and Walls: We mark the guard's and wall's positions on the grid with 2. The grid now looks like this:

```
[0][0][0]
[0][0][0]
```

[0][0][2]

[0][2][0]

[0][0][0]

cells.

grid.

[0][0][0]

wall at (0, 2). • East Direction (to the right): The guard checks the cell (1, 2). It's a 0, so we mark it as 1.

After the guard has scanned all four directions, the grid now looks like:

Initialize the grid with 0 to represent unguarded cells

Mark the positions of guards and walls with 2 on the grid

row, col = row + delta_row, col + delta_col

Count the number of cells that are not guarded (value 0 in the grid)

public int countUnguarded(int m, int n, int[][] guards, int[][] walls) {

// 1 represents a cell guarded by a guard's line of sight,

// 2 represents an occupied cell by either a guard or a wall.

 $grid = [[0] * n for _ in range(m)]$

for guard_row, guard_col in guards:

grid[wall_row] [wall_col] = 2

for guard_row, guard_col in guards:

for delta_row, delta_col in directions:

row, col = guard_row, guard_col

grid[row][col] = 1

// Initialize the grid representation where

// Mark all the guard positions on the grid.

grid[guard[0]][guard[1]] = 2;

// 0 represents an unguarded cell,

int[][] grid = new int[m][n];

for (int[] guard : guards) {

for (auto& guard : guards) {

for (auto& wall : walls) {

for (auto& guard : guards) {

grid[guard[0]][guard[1]] = 2;

// Set wall positions to 2 in the grid

int directions $[5] = \{-1, 0, 1, 0, -1\};$

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

x += dx;

y += dy;

// Direction vectors to move up, right, down, or left

// Visit each guard position and mark guarded areas

// Count unguarded cells (where grid value is still 0)

// Observing in all 4 directions from the guard position

grid[x][y] = 1; // Mark the cell as guarded

let [deltaX, deltaY] = [directions[k], directions[k + 1]];

// Mark the guarded position with a 1

x += deltaX;

y += deltaY;

grid[x][y] = 1;

int x = guard[0], y = guard[1]; // Starting guard's position

int dx = directions[k], dy = directions[k + 1]; // Direction changes

// Move in the current direction until hitting a wall, guard, or boundary

while $(x + dx >= 0 \& x + dx < rows \& y + dy >= 0 \& y + dy < cols & grid[x + dx][y + dy] < 2) {$

grid[wall[0]][wall[1]] = 2;

for wall_row, wall_col in walls:

grid[guard_row][guard_col] = 2

• South Direction (downwards): The guard checks the cell (2, 1). It's a 0, so we mark it as 1.

• West Direction (to the left): The guard checks the cell (1, 0). It's a 0, so we mark it as 1.

[0][1][2] [1][2][1] [0][1][0]

Guarding Cells: Since the guard can look in all four directions, we will simulate this for our single guard at (1, 1):

∘ North Direction (upwards): We check the cell (0, 1). Since it's a 0, we mark it as 1. The cell (0, 2) won't be checked because there's a

Solution Implementation **Python**

def countUnguarded(self, m: int, n: int, guards: List[List[int]], walls: List[List[int]]) -> int:

Continue marking cells in current direction until a wall or guard is reached

while 0 <= row + delta_row < m and 0 <= col + delta_col < n and grid[row + delta_row][col + delta_col] < 2:</pre>

• Counting Unguarded Cells: We count all 0s in the grid to find the unguarded cells. There are three 0s, which means there are three unguarded

This simple example demonstrates how the solution approach accurately simulates guards' visibility to find unguarded cells in the

Define the directions in which guards look (up, right, down, left) directions = [(-1, 0), (0, 1), (1, 0), (0, -1)]# Scan the grid for guards and set the cells they see to 1

from typing import List

class Solution:

```
return sum(cell == 0 for row in grid for cell in row)
Java
```

class Solution {

```
// Mark all the wall positions on the grid.
        for (int[] wall : walls) {
            grid[wall[0]][wall[1]] = 2;
       // This array represents the directional increments: up, right, down, and left.
        int[] directions = \{-1, 0, 1, 0, -1\};
       // Iterate over each guard and mark their line of sight until they hit a wall or the grid's edge.
        for (int[] guard : guards) {
            // Four directions — up, right, down, left
            for (int k = 0; k < 4; ++k) {
                // Starting position for the current guard.
                int x = guard[0], y = guard[1];
                // Directional increments for the current direction.
                int deltaX = directions[k], deltaY = directions[k + 1];
                // Keep marking the grid in the current direction till
                // you hit a wall, guard or boundary of the grid.
                while (x + deltaX) >= 0 && x + deltaX < m && y + deltaY >= 0 && y + deltaY < n && grid[x + deltaX][y + deltaY] < 2
                    x += deltaX;
                    y += deltaY;
                    grid[x][y] = 1; // Marking the cell as guarded.
       // Count all the unguarded spaces, where the grid value is 0.
        int unguardedCount = 0;
        for (int[] row : grid) {
            for (int cellValue : row) {
                if (cellValue == 0) {
                    unguardedCount++;
       // Return the total number of unguarded cells.
       return unguardedCount;
C++
#include <vector>
#include <cstring> // for memset
using namespace std;
class Solution {
public:
    // Function to count unguarded cells in a matrix representing a room
    int countUnguarded(int rows, int cols, vector<vector<int>>& guards, vector<vector<int>>& walls) {
        int grid[rows][cols];
       memset(grid, 0, sizeof(grid)); // Initialize the entire grid to 0
       // Set guard positions to 2 in the grid
```

```
int unquardedCount = 0;
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
               if (grid[i][j] == 0) {
                    unguardedCount++;
        return unguardedCount;
};
TypeScript
function countUnguarded(maxRows: number, maxCols: number, guards: number[][], walls: number[][]): number {
   // Create a grid to represent the museum with default values set to 0
   const grid: number[][] = Array.from({ length: maxRows }, () => Array.from({ length: maxCols }, () => 0));
   // Mark the positions of guards with 2 on the grid
    for (const [row, col] of guards) {
       grid[row][col] = 2;
   // Mark the positions of walls with 2 on the grid
   for (const [row, col] of walls) {
       grid[row][col] = 2;
   // Directions array to facilitate exploration in the 4 cardinal directions
   const directions: number[] = [-1, 0, 1, 0, -1];
   // For each guard, mark the positions they can guard based on the grid constraints
   for (const [guardRow, guardCol] of guards) {
       // Check all directions: up, right, down, and left
        for (let k = 0; k < 4; ++k) {
           // Initialize guard's position to start casting
            let [x, y] = [guardRow, guardCol];
```

// Move in the current direction as long as it's within bounds and not blocked by walls or other guards

while (x + deltaX) >= 0 && x + deltaX < maxRows && y + deltaY >= 0 && y + deltaY < maxCols && grid[x + deltaX][y + deltaY]

```
// Count the number of unguarded positions in the grid
      let unguardedCount = 0;
      for (const row of grid) {
          for (const cell of row) {
              if (cell === 0) {
                  unguardedCount++;
      // Return the count of unguarded positions
      return unguardedCount;
from typing import List
class Solution:
   def countUnguarded(self, m: int, n: int, guards: List[List[int]], walls: List[List[int]]) -> int:
       # Initialize the grid with 0 to represent unguarded cells
        grid = [[0] * n for _ in range(m)]
       # Mark the positions of guards and walls with 2 on the grid
        for guard_row, guard_col in guards:
            grid[guard_row][guard_col] = 2
        for wall_row, wall_col in walls:
           grid[wall_row][wall_col] = 2
       # Define the directions in which guards look (up, right, down, left)
        directions = [(-1, 0), (0, 1), (1, 0), (0, -1)]
       # Scan the grid for guards and set the cells they see to 1
        for guard_row, guard_col in guards:
            for delta_row, delta_col in directions:
                row, col = guard_row, guard_col
                # Continue marking cells in current direction until a wall or guard is reached
               while 0 <= row + delta_row < m and 0 <= col + delta_col < n and grid[row + delta_row][col + delta_col] < 2:</pre>
                    row, col = row + delta_row, col + delta_col
                   grid[row][col] = 1
       # Count the number of cells that are not guarded (value 0 in the grid)
```

The time complexity of the code is 0(m * n + g * (m + n)), where m and n correspond to the number of rows and columns of the grid, and g is the number of guards. The reasoning behind this is as follows:

Time and Space Complexity

Time Complexity

return sum(cell == 0 for row in grid for cell in row)

 The initial setup of the grid g with size m * n takes 0(m * n) time. • The loops for placing guards and walls each run at most 0(g + w), where w is the number of walls, however, these are negligible compared to other terms when g and w are much smaller than m * n.

- taken), hence 4*(m + n) for each guard, aggregating to g * (m + n) for all guards.
- **Space Complexity** The space complexity of the algorithm is 0(m * n). This is because only a single m * n grid is used as extra space to store the

• The nested loops iterate through each guard, and for each guard, scan across the grid in four directions up to m or n times (whichever direction

state of each cell (whether it is free, guarded, or a wall).