<https://leetcode.ca/all/489.html>

Given a robot cleaner in a room modeled as a grid.

Each cell in the grid can be empty or blocked.

The robot cleaner with 4 given APIs can move forward, turn left or turn right. Each turn it made is 90 degrees.

When it tries to move into a blocked cell, its bumper sensor detects the obstacle and it stays on the current cell.

Design an algorithm to clean the entire room using only the 4 given APIs shown below.

interface Robot {

  // returns true if next cell is open and robot moves into the cell.

  // returns false if next cell is obstacle and robot stays on the current cell.

  boolean move();

// Robot will stay on the same cell after calling turnLeft/turnRight.

  // Each turn will be 90 degrees.

  void turnLeft();

  void turnRight();

// Clean the current cell.

void clean();

}

Example:

Input:

room = [

[1,1,1,1,1,0,1,1],

[1,1,1,1,1,0,1,1],

[1,0,1,1,1,1,1,1],

[0,0,0,1,0,0,0,0],

[1,1,1,1,1,1,1,1]

],

row = 1,

col = 3

Explanation:

All grids in the room are marked by either 0 or 1.

0 means the cell is blocked, while 1 means the cell is accessible.

The robot initially starts at the position of row=1, col=3.

From the top left corner, its position is one row below and three columns right.

**Notes:**

The input is only given to initialize the room and the robot's position internally. You must solve this problem "blindfolded". In other words, you must control the robot using only the mentioned 4 APIs, without knowing the room layout and the initial robot's position.

The robot's initial position will always be in an accessible cell.

The initial direction of the robot will be facing up.

All accessible cells are connected, which means the all cells marked as 1 will be accessible by the robot.

Assume all four edges of the grid are all surrounded by wall.

**Attempt 1: 2024-11-29**

**Solution 1: DFS + Special Backtracking (60 min)**

interface Robot {

boolean move(); // Moves the robot one step forward; returns false if blocked.

void turnLeft(); // Rotates the robot 90 degrees counterclockwise.

void turnRight(); // Rotates the robot 90 degrees clockwise.

void clean(); // Cleans the current cell.

}

class Solution {

// Directions: [up, right, down, left]

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

private Set<String> visited = new HashSet<>();

public void cleanRoom(Robot robot) {

helper(robot, 0, 0, 0); // Start at (0, 0), facing up (direction = 0)

}

private void helper(Robot robot, int x, int y, int dir) {

// Mark the current cell as visited and clean it

String position = x + "," + y;

if (visited.contains(position)) {

return;

}

visited.add(position);

robot.clean();

// Explore the 4 directions

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

int newDir = (dir + i) % 4; // Compute the new direction

int new\_x = x + directions[newDir][0];

int new\_y = y + directions[newDir][1];

// Only when next cell not visited and accessible, we will continue discover

if (!visited.contains(new\_x + "," + new\_y) && robot.move()) {

// Move to the new cell and backtrack

helper(robot, new\_x, new\_y, newDir);

// Backtrack to the previous position and restore direction

goBack(robot);

}

// Turn the robot 90 degrees clockwise

// Note: Not like normal DFS problem can naturally(auto) change to other directions

// since the given problem requires use 'turn direction API' to change the

// direction, we have to 'manually' change the direction in for loop each time

robot.turnRight();

}

}

private void goBack(Robot robot) {

// Move back to the previous cell

robot.turnRight();

robot.turnRight();

robot.move();

robot.turnRight();

robot.turnRight();

}

}

**Refer to**

<https://algo.monster/liteproblems/489>

**Problem Description**

This problem presents a scenario where you are in control of a robot placed in an unknown position within a room. The room is set up as a grid with "1" representing an empty space that the robot can move through, and "0" representing a wall that the robot cannot pass. Your task is to create an algorithm that instructs the robot to clean the entire room, covering every "1" cell in the grid.

The robot is equipped with a set of APIs that allow it to:

Move forward into an adjacent cell (move method), unless there is a wall which will cause the robot to stay put.

Turn left or right by 90 degrees without moving from its current cell (turnLeft and turnRight methods).

Clean the cell it is currently on (clean method).

One crucial detail to note is that the initial orientation of the robot is facing "up," and all four edges of the grid are walled off. Furthermore, you must write this algorithm without any knowledge of the room's layout or the robot's starting position.

**Intuition**

The solution to this problem involves a depth-first search (DFS) strategy. **The idea behind DFS is to exhaustively explore a path until hitting an obstacle (in this case, a wall), backtrack, and then try another path. This principle is well-suited for the problem since we need to ensure every navigable cell is visited and cleaned.**

Since the robot's starting point is unknown, and the room's layout is also unknown, we can imagine the robot's initial position as the origin of a coordinate system, (0, 0).

For the DFS to work, we need to:

Keep track of cells that have already been visited to avoid redundant cleaning and to prevent the robot from going in cycles.

Identify a way to represent the robot's orientation since it can face four different directions (up, right, down, and left). We use a direction vector (dirs) for this purpose.

The dfs function in the solution recursively explores the grid by trying to move in the current facing direction, cleaning the cell, and then attempting to move forward in the next direction. **If a move is successful (indicating an empty cell), the robot moves forward, and the search continues from the new cell position. After exploring one direction, the robot turns 90 degrees to the right to check the next direction.**

**Once the robot hits a wall and cannot move forward:**

**It backtracks to the previous cell by performing a 180-degree turn (turnRight twice), moving forward one cell, and then realigning to the original orientation by another 180-degree turn.**

**It then continues the DFS from the previous cell in the next direction.**

This process continues until all reachable cells have been visited and cleaned.

**Solution Approach**

The solution uses depth-first search (DFS) to navigate and clean the room. DFS is a popular algorithm for exploring all nodes in a graph or all vertices in a grid by moving as far as possible until you can no longer proceed, then backtracking.

Here's how the algorithm in the provided solution works:

Data Structure for Visited Cells: A set named vis is used to keep track of visited coordinates. This prevents the robot from cleaning the same cell multiple times and ensures that the algorithm terminates.

Directional Handling: The dirs tuple provides the relative coordinates when moving up, right, down, and left respectively, assuming an orientation of facing up initially. These directions correspond to the changes in the robot's coordinates given its current direction.

Recursive DFS Function: The heart of the solution is the dfs(i, j, d) function, which takes the robot's current x and y coordinates (i and j), and its current direction (d).

It starts by cleaning the current cell and adding the cell to the vis (visited set).

The robot then attempts to move in its current direction and three more directions by turning right sequentially.

After attempting to move in each direction:

If the move is successful, the robot has discovered a new cell, and dfs is called recursively from that new position.

After the recursive call (which represents backtracking to the source cell), the robot performs two 90-degree turns to face the opposite direction (robot.turnRight called twice), moves forward to return to the original cell, and then realigns to the initial direction by turning another 180 degrees in total.

It then turns right to change to the next direction and continue the DFS.

Triggering DFS: The DFS begins by calling dfs(0, 0, 0) from the robot's initial position, which is treated as the origin (0, 0) and its initial direction (up).

Throughout this process, the robot continues to move, turn, and clean until it has reached and cleaned all accessible cells. The vis set ensures that the robot never revisits a cell, efficiently covering the entire grid.

By using these approaches, the robot is able to clean the entire room—represented by the 1 cells in the grid—without any prior knowledge of the room's layout or its own initial position.

**Example Walkthrough**

Imagine a room represented by the following 3x3 grid and the robot starts at the cell marked R (which we consider (0, 0) for our coordinate system), facing upwards. 1 indicates an empty space, and 0 indicates a wall:

Room:

1 1 1

0 1 0

R 1 1

Step-by-step using the DFS approach:

Initialization: The robot starts at (0, 0). The cell is automatically cleaned and is added to the visited set vis.

Exploration:

The robot tries to move forward (up) into (0, 1). Since it's free, the cell is cleaned, and this position is added to vis. The function dfs(0, 1, 0) is called.

From (0,1), the robot continues and moves up to (0,2), which is also free. The robot cleans it and then attempts to go right, but there is a wall, so it turns right and tries to go down but hits another wall. It turns right again, and now it moves left, cleans (0,1), and returns to (0,2) and faces down (original direction).

The sequence repeats until it faces upward again, and since all directions are visited, it returns to (0,1). The robot tries to go right, and it's blocked, so it tries to go down and arrives at (0,0), which is already cleaned and in vis.

Now the robot tries to move right from (0,0). It can't because there is a wall, so it rotates right and moves down to (1,0). The robot cleans the cell, then tries to go left (hit wall), up (cleaned and in vis), right, and finally down. Since down is the only direction it hasn't come from, it moves to (2,0) and cleans it.

At (2,0), the robot repeats the process: trying each direction, cleaning new cells, and moving accordingly. It soon finds it can move to (2,1) and cleans it.

At (2,1), the robot cannot move right or down because of walls; it moves up and finds it's already visited (1,1). After rotating right, it can move left to (2,0) but since it's been visited, the robot rotates right again and finally faces the last unexplored direction at (2,1) which is the cell on its left (2,2).

At (2,2), which is the last cell, the robot can't move up or right due to walls, and the left and down directions lead to already cleaned cells. It has now finished cleaning.

By the end of this process, the robot has covered the entire accessible area, and each 1 has been visited and turned into a 2 (indicating the robot has cleaned it).

Resulting visited cells:

2 2 2

0 2 0

2 2 2

This example shows how the robot effectively uses the DFS method to clean the entire room. Each step is guided by whether the space has been visited or if there's an obstacle, and the visited set vis ensures that no space gets cleaned twice. The right turning mechanism helps the robot to systematically explore all four directions in its current location.

**Solution Implementation**

import java.util.Set;

import java.util.HashSet;

import java.util.List;

// We're assuming that the Robot interface is defined elsewhere according to the initial code block

interface Robot {

public boolean move();

public void turnLeft();

public void turnRight();

public void clean();

}

public class RobotCleaner {

// Store the four possible directions the robot can move: up, right, down, left.

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

// Create a HashSet to keep track of visited cells, represented as (x, y) coordinates.

private final Set<List<Integer>> visited = new HashSet<>();

// The robot interface instance.

private Robot robot;

/\*\*

\* Public method to clean the room. This method will be called initially.

\*

\* @param robot The robot interface through which we control and clean.

\*/

public void cleanRoom(Robot robot) {

this.robot = robot;

// Begin the depth-first search (DFS) at the starting point (0, 0) with an initial direction (0).

dfs(0, 0, 0);

}

/\*\*

\* Perform DFS to clean the room. Will explore all four directions at each point.

\*

\* @param row Current row position of the robot.

\* @param col Current column position of the robot.

\* @param direction Current direction the robot is facing.

\* (0, 1, 2, 3) corresponds to (up, right, down, left).

\*/

private void dfs(int row, int col, int direction) {

// Clean the current cell.

robot.clean();

// Mark the current cell as visited.

visited.add(List.of(row, col));

// Explore all four directions.

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

// Compute the new direction.

int newDirection = (direction + k) % 4;

// Compute the next cell's coordinates based on the new direction.

int nextRow = row + directions[newDirection];

int nextCol = col + directions[newDirection + 1];

// If the next cell has not been visited and is accessible, move and continue DFS.

if (!visited.contains(List.of(nextRow, nextCol)) && robot.move()) {

dfs(nextRow, nextCol, newDirection);

// Move back to the previous cell (backtracking).

robot.turnRight();

robot.turnRight();

robot.move();

// Reorient to the original direction before the next loop iteration.

robot.turnRight();

robot.turnRight();

}

// Turn the robot to the next direction (90 degrees to the right).

// Note: Not like normal DFS problem can naturally(auto) change to other directions

// since the given problem requires use 'turn direction API' to change the

// direction, we have to 'manually' change the direction in for loop each time

robot.turnRight();

}

}

}

**Time and Space Complexity**

The time complexity of the above code is O(4^(N-M)), where N is the total number of cells in the room and M is the number of obstacles. This is because the algorithm has to visit each non-obstacle cell once and at each cell, it makes up to 4 decisions – move in 4 possible directions. The recursion may go up to 4 branches at each level but would not revisit cells that are already visited, summarized by visited set vis.

The space complexity of the DFS is O(N) for the recursive call stack as well as the space to hold the set of visited cells (in the worst case where there are no obstacles and we can move to every cell). However, in a densely packed room with obstructions, the number of visited states will be less than N.

**Refer to chatGPT**

**Problem Analysis**

**Key Challenges:**

The grid dimensions are unknown.

Obstacles may block paths, and the robot can only move in four directions (up, right, down, left).

The robot has no direct access to the grid but can:

Move forward (robot.move()).

Turn left/right (robot.turnLeft() or robot.turnRight()).

Clean the current cell (robot.clean()).

**Solution Approach:**

**Use backtracking to explore all reachable cells.**

Maintain a Set to track visited cells, avoiding re-cleaning or re-processing.

Handle movement and directional changes programmatically.

import java.util.HashSet;

import java.util.Set;

interface Robot {

boolean move(); // Moves the robot one step forward; returns false if blocked.

void turnLeft(); // Rotates the robot 90 degrees counterclockwise.

void turnRight(); // Rotates the robot 90 degrees clockwise.

void clean(); // Cleans the current cell.

}

class Solution {

// Directions: [up, right, down, left]

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

private Set<String> visited = new HashSet<>();

public void cleanRoom(Robot robot) {

backtrack(robot, 0, 0, 0); // Start at (0, 0), facing up (direction = 0)

}

private void backtrack(Robot robot, int x, int y, int dir) {

// Mark the current cell as visited and clean it

String position = x + "," + y;

if (visited.contains(position)) return;

visited.add(position);

robot.clean();

// Explore the 4 directions

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

int newDir = (dir + i) % 4; // Compute the new direction

int nx = x + directions[newDir][0];

int ny = y + directions[newDir][1];

if (!visited.contains(nx + "," + ny) && robot.move()) {

// Move to the new cell and backtrack

backtrack(robot, nx, ny, newDir);

// Backtrack to the previous position and restore direction

goBack(robot);

}

// Turn the robot 90 degrees clockwise

// Note: Not like normal DFS problem can naturally(auto) change to other directions

// since the given problem requires use 'turn direction API' to change the

// direction, we have to 'manually' change the direction in for loop each time

robot.turnRight();

}

}

private void goBack(Robot robot) {

// Move back to the previous cell

robot.turnRight();

robot.turnRight();

robot.move();

robot.turnRight();

robot.turnRight();

}

}

**Normally in backtracking problem, we have below pattern on dealing with 'visited' status:**

visited.add(position);

for (int i = 0; i < 4; i++) { ... dfs work ...}

visited.remove(position);

**but why not seen this pattern in this problem ?**

The typical backtracking pattern you mentioned, where the visited status is added and then removed (e.g., visited.add() and visited.remove()), is not needed in LeetCode 489 (Robot Room Cleaner) because **the problem involves exploring a physical environment with a fixed grid and the robot cannot revisit cells that have already been cleaned.**

Here’s why we don’t see the visited.remove() step in this problem:

**No Revisit Requirement**

In Robot Room Cleaner, each cell is visited only once because:

A cleaned cell does not need to be processed again.

Revisiting a cell would result in redundant work, which we avoid by maintaining the visited set.

The visited set permanently marks cells as visited, unlike problems where we may revisit cells for different paths.

**Search Space is Fixed**

**Unlike traditional backtracking problems (e.g., where paths or permutations are explored), here:**

**The robot operates in a fixed grid environment.**

**Each cell has exactly one state: visited or unvisited.**

**Once marked as visited, a cell remains permanently visited.**

**This means there's no need to "unmark" a cell after exploring it, as doing so would only result in redundant computation.**

**The Robot Physically Moves**

In this problem, the robot's position must be restored after exploring all directions (using the goBack() function).

The backtracking logic ensures the robot correctly navigates the grid and cleans all reachable cells without revisiting or undoing visited status.

**Traditional Pattern is for Non-Physical Backtracking**

Problems like generating permutations or subsets use the visited.remove() step to explore all possible paths, even if they revisit nodes.

Example: In permutations, revisiting elements is necessary to generate new orderings.

Here, revisiting is unnecessary and inefficient because:

The goal is to clean every cell exactly once, not explore multiple paths to the same cell.

**Comparison Example**

**Traditional Backtracking Problem:**

**Generating subsets of a list:**

void backtrack(List<Integer> subset, int[] nums, int start) {

result.add(new ArrayList<>(subset));

for (int i = start; i < nums.length; i++) {

subset.add(nums[i]);

backtrack(subset, nums, i + 1);

subset.remove(subset.size() - 1); // Undo the change

}

}

Here, subset.remove(...) is essential to explore all subsets.

**Robot Cleaner:**

**Permanent marking of visited cells:**

visited.add(position); // Mark as visited

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

if (!visited.contains(nextPosition)) {

backtrack(robot, nx, ny, newDir);

}

}

goBack(robot); // Restore position, but no "visited.remove()"

Here, there’s no need to "undo" visited.add(position) because we only care about reaching and cleaning all cells.

**Conclusion**

In LeetCode 489, the visited.remove() step is unnecessary because:

**Cells are only cleaned once and never revisited.**

**The goal is to clean all reachable cells, not explore all paths to a cell.**

**Permanent marking with visited.add() ensures efficiency by avoiding redundant work.**