**Robot Room Cleaner**

You are controlling a robot that is located somewhere in a room. The room is modeled as an m x n binary grid where 0 represents a wall and 1 represents an empty slot.

The robot starts at an unknown location in the room that is guaranteed to be empty, and you do not have access to the grid, but you can move the robot using the given API Robot.

You are tasked to use the robot to clean the entire room (i.e., clean every empty cell in the room). The robot with the four given APIs can move forward, turn left, or turn right. Each turn is 90 degrees.

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

Design an algorithm to clean the entire room using the following APIs:

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();

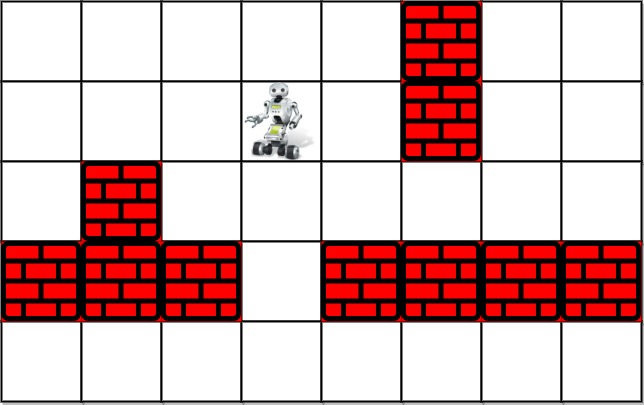
}

**Note** that the initial direction of the robot will be facing up. You can assume all four edges of the grid are all surrounded by a wall.

**Custom testing:**

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 four mentioned APIs without knowing the room layout and the initial robot's position.

**Example 1:**



**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

**Output:** Robot cleaned all rooms.

**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.

**Example 2:**

**Input:** room = [[1]], row = 0, col = 0

**Output:** Robot cleaned all rooms.

**Constraints:**

* m == room.length
* n == room[i].length
* 1 <= m <= 100
* 1 <= n <= 200
* room[i][j] is either 0 or 1.
* 0 <= row < m
* 0 <= col < n
* room[row][col] == 1
* All the empty cells can be visited from the starting position.

/\*\*

\* // This is the robot's control interface.

\* // You should not implement it, or speculate about its implementation

\* function Robot() {

\* // Returns true if the cell in front is open and robot moves into the cell.

\* // Returns false if the cell in front is blocked and robot stays in the current cell.

\* @return {boolean}

\* this.move = function() {

\* ...

\* };

\*

\* // Robot will stay in the same cell after calling turnLeft/turnRight.

\* // Each turn will be 90 degrees.

\* @return {void}

\* this.turnLeft = function() {

\* ...

\* };

\*

\* // Robot will stay in the same cell after calling turnLeft/turnRight.

\* // Each turn will be 90 degrees.

\* @return {void}

\* this.turnRight = function() {

\* ...

\* };

\*

\* // Clean the current cell.

\* @return {void}

\* this.clean = function() {

\* ...

\* };

\* };

\*/

/\*\*

\* @param {Robot} robot

\* @return {void}

\*/

var cleanRoom = function(robot) {

};

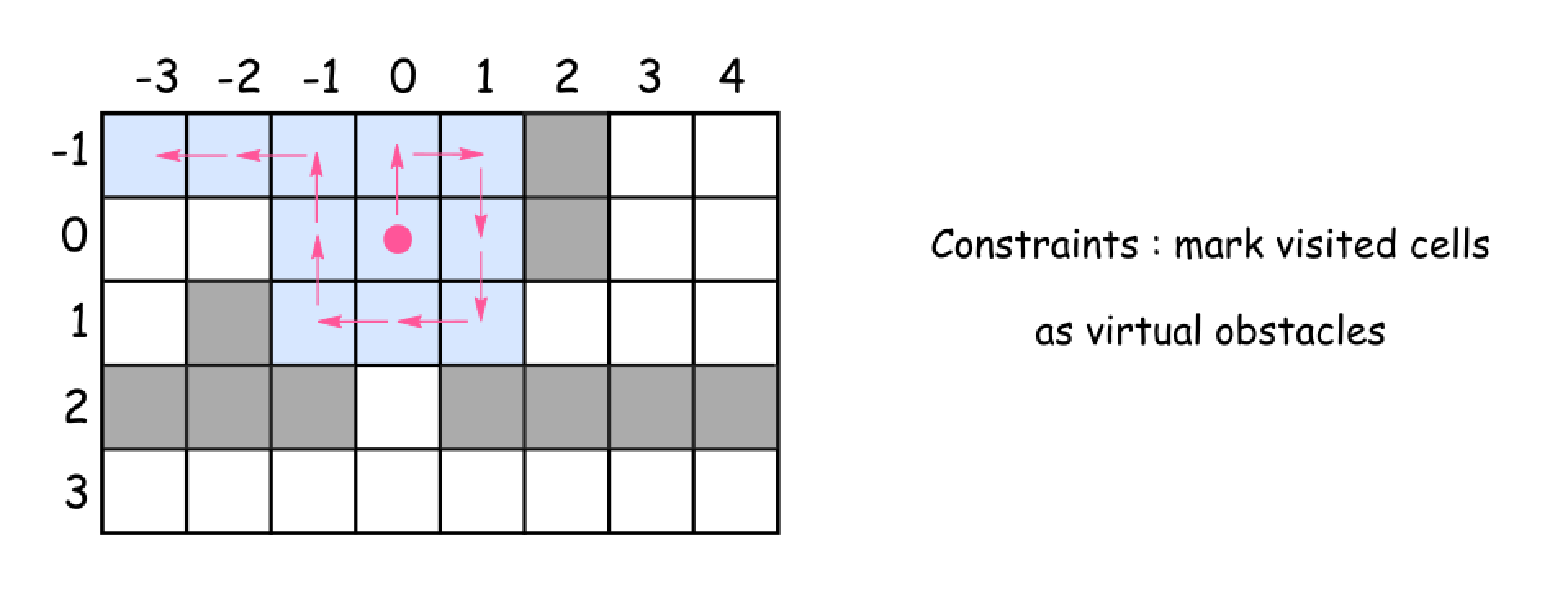
Approach 1: Spiral Backtracking

**Concepts to use**

Let's use here two programming concepts.

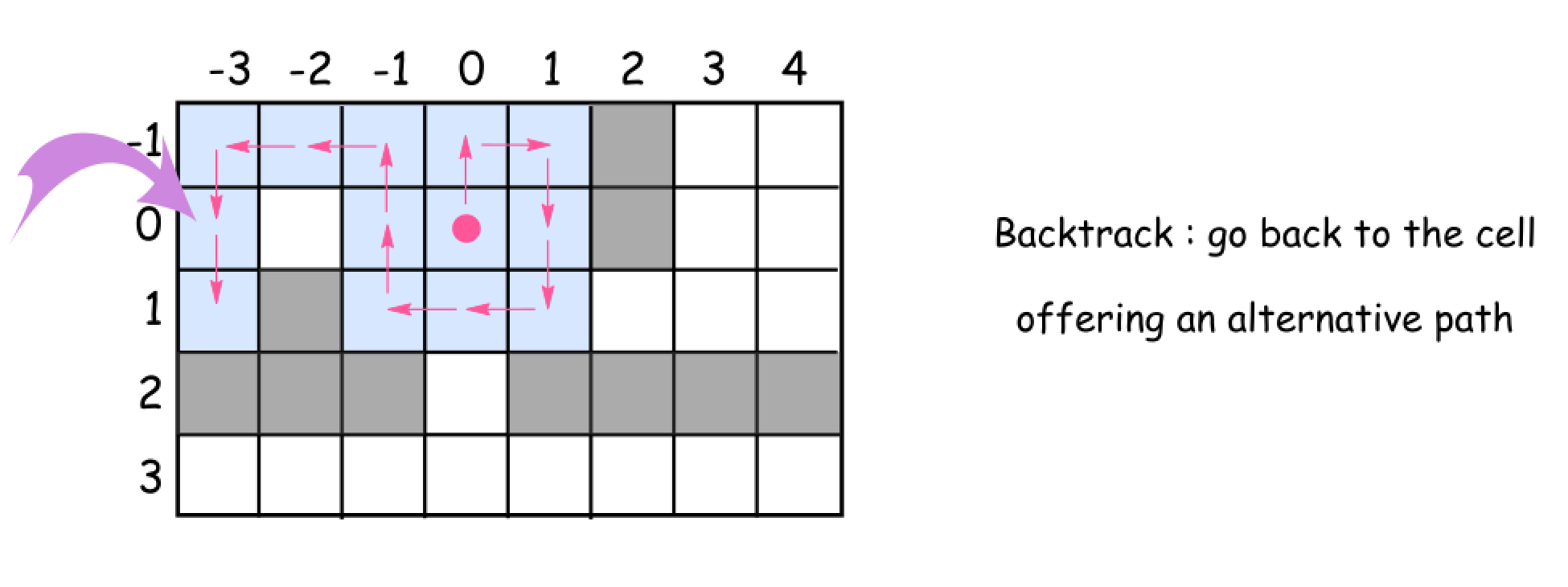
The first one is called *constrained programming*.

That basically means to put restrictions after each robot move. Robot moves, and the cell is marked as visited. That propagates *constraints* and helps to reduce the number of combinations to consider.



The second one called *backtracking*.

Let's imagine that after several moves the robot is surrounded by the visited cells. But several steps before there was a cell which proposed an alternative path to go. That path wasn't used and hence the room is not yet cleaned up. What to do? *To backtrack*. That means to come back to that cell, and to explore the alternative path.



**Intuition**

This solution is based on the same idea as maze solving algorithm called [right-hand rule](https://en.wikipedia.org/wiki/Maze_solving_algorithm#Wall_follower). Go forward, cleaning and marking all the cells on the way as visited. At the obstacle *turn right*, again go forward, *etc*. Always *turn right* at the obstacles and then go forward. Consider already visited cells as virtual obstacles.

What to do if after the right turn there is an obstacle just in front ?

*Turn right* again.

How to explore the alternative paths from the cell ?

Go back to that cell and then *turn right* from your last explored direction.

When to stop ?

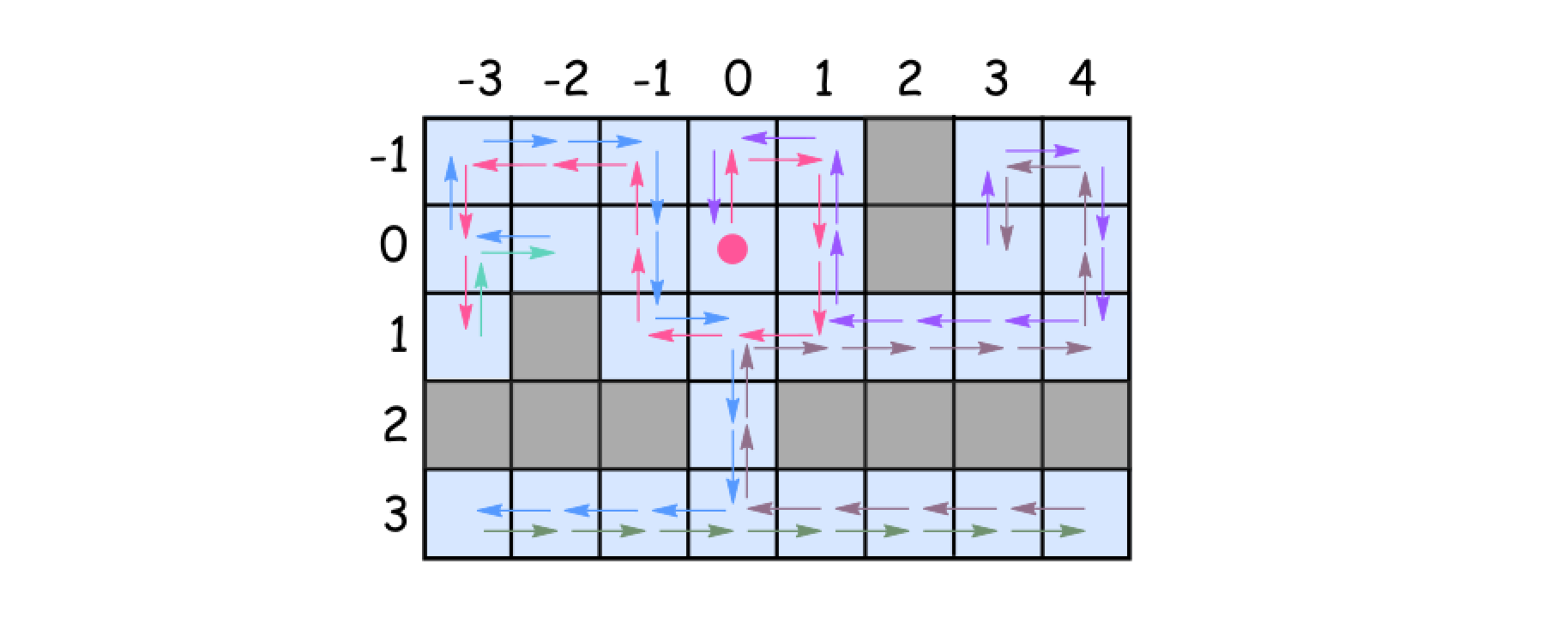
Stop when you explored all possible paths, *i.e.* all 4 directions (up, right, down, and left) for each visited cell.

**Algorithm**

Time to write down the algorithm for the backtrack function backtrack(cell = (0, 0), direction = 0).

* Mark the cell as visited and clean it up.
* Explore 4 directions : up, right, down, and left (the order is important since the idea is always to turn right) :
  + Check the next cell in the chosen direction :
    - If it's not visited yet and there is no obtacles :
      * Move forward.
      * Explore next cells backtrack(new\_cell, new\_direction).
      * Backtrack, *i.e.* go back to the previous cell.
    - Turn right because now there is an obstacle (or a virtual obstacle) just in front.

**Implementation**



class Solution {

// going clockwise : 0: 'up', 1: 'right', 2: 'down', 3: 'left'

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

Set<Pair<Integer, Integer>> visited = new HashSet();

Robot robot;

public void goBack() {

robot.turnRight();

robot.turnRight();

robot.move();

robot.turnRight();

robot.turnRight();

}

public void backtrack(int row, int col, int d) {

visited.add(new Pair(row, col));

robot.clean();

// going clockwise : 0: 'up', 1: 'right', 2: 'down', 3: 'left'

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

int newD = (d + i) % 4;

int newRow = row + directions[newD][0];

int newCol = col + directions[newD][1];

if (!visited.contains(new Pair(newRow, newCol)) && robot.move()) {

backtrack(newRow, newCol, newD);

goBack();

}

// turn the robot following chosen direction : clockwise

robot.turnRight();

}

}

public void cleanRoom(Robot robot) {

this.robot = robot;

backtrack(0, 0, 0);

}

}

C**omplexity Analysis**

* Time complexity : O(N - M)*O*(*N*−*M*), where N*N* is a number of cells in the room and M*M* is a number of obstacles.
  + We visit each non-obstacle cell once and only once.
  + At each visit, we will check 4 directions around the cell. Therefore, the total number of operations would be 4 \cdot (N-M)4⋅(*N*−*M*).
* Space complexity : O(N - M)*O*(*N*−*M*), where N*N* is a number of cells in the room and M*M* is a number of obstacles.
  + We employed a hashtable to keep track of whether a non-obstacle cell is visited or not.