The Maze III

There is a **ball** in a maze with empty spaces and walls. The ball can go through empty spaces by rolling **up** (u), **down** (d), **left** (l) or **right** (r), but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction. There is also a **hole** in this maze. The ball will drop into the hole if it rolls on to the hole.

Given the **ball position**, the **hole position** and the **maze**, find out how the ball could drop into the hole by moving the **shortest distance**. The distance is defined by the number of **empty spaces** traveled by the ball from the start position (excluded) to the hole (included). Output the moving **directions** by using 'u', 'd', 'l' and 'r'. Since there could be several different shortest ways, you should output the **lexicographically smallest** way. If the ball cannot reach the hole, output "impossible".

The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of the maze are all walls. The ball and the hole coordinates are represented by row and column indexes.

Example 1

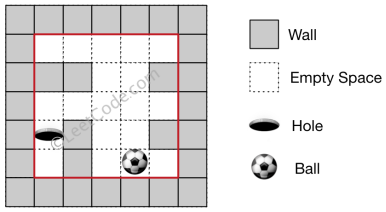
```
Input 1: a maze represented by a 2D array

0 0 0 0 0
1 1 0 0 1
0 0 0 0 0
0 1 0 0 1
0 1 0 0 0

Input 2: ball coordinate (rowBall, colBall) = (4, 3)
Input 3: hole coordinate (rowHole, colHole) = (0, 1)

Output: "lul"
Explanation: There are two shortest ways for the ball to drop into the hole.
The first way is left -> up -> left, represented by "lul".
The second way is up -> left, represented by 'ul'.
Both ways have shortest distance 6, but the first way is lexicographically smaller
because 'l'
```

Example 2

Input 1: a maze represented by a 2D array

```
0 0 0 0 0
1 1 0 0 1
0 0 0 0 0
0 1 0 0 1
0 1 0 0 0
```

Input 2: ball coordinate (rowBall, colBall) = (4, 3)
Input 3: hole coordinate (rowHole, colHole) = (3, 0)
Output: "impossible"
Explanation: The ball cannot reach the hole.



Note:

1. There is only one ball and one hole in the maze.
2. Both the ball and hole exist on an empty space, and they will not be at the same position initially.
3. The given maze does not contain border (like the red rectangle in the example pictures), but you could assume the border of the maze are all walls.
4. The maze contains at least 2 empty spaces, and the width and the height of the maze won't exceed 30.

Solution 1

```
public class Solution {
    int min; //min distance to hole
    String minS; //min distance's path string
    int[] hole;
    int[][] maze;
    int[][] map; //shortest distant traveling from ball to this point
    int[][] dirs = {{0,1},{-1,0},{1,0},{0,-1}}; //r, u, d, l
    public String findShortestWay(int[][] maze, int[] ball, int[] hole) {
        this.min = Integer.MAX_VALUE;
        this.minS = null;
        this.hole = hole;
        this.maze = maze;
        this.map = new int[maze.length][maze[0].length];
        for(int i = 0; i<map.length; i++) Arrays.fill(map[i], Integer.MAX_VALUE);


        move(ball[0], ball[1], 0, "", -1);
        return (minS==null) ? "impossible" : minS;
    }

    private void move(int r, int c, int cnt, String path, int dir){//dir is a inde
x of dirs
        if(cnt > min || cnt > map[r][c]) return; //not a shortest route for sure
        if(dir!=-1){//if not from start point
            //add path
            if(dir==0) path+='r';
            else if(dir==1) path+='u';
            else if(dir==2) path+='d';
            else path+='l';

            //roll along dir
            while(r>=0 && r<maze.length && c>=0 && c<maze[0].length && maze[r][c]
==0){
                map[r][c] = Math.min(map[r][c], cnt);
                if(r==hole[0] && c==hole[1]){//check hole
                    if(cnt==min && path.compareTo(minS)<0){
                        minS=path;
                    }else if(cnt<min){
                        min = cnt;
                        minS = path;
                    }
                    return;
                }
                r += dirs[dir][0];
                c += dirs[dir][1];
                cnt++;
            }
            r -= dirs[dir][0];//[r,c] is wall, need to walk back 1 step
            c -= dirs[dir][1];
            cnt--;
        }

        //hit wall (or start) -> try to turn
        for(int i = 0; i<dirs.length; i++){
```

```
            if(dir == i) continue;//dont keep going
            if(dir == (3-i)) continue;//dont go back
            int newR = r + dirs[i][0];
            int newC = c + dirs[i][1];
            if(newR>=0 && newR<maze.length && newC>=0 && newC<maze[0].length && m
aze[newR][newC]==0){//can go
                move(r, c, cnt, path, i);
            }
        }
    }
}
```

Each time, first add the direction to the path, and then go with that direction, checking for hole along the way. When hit a wall, try to turn, and go with the new direction. For the starting point, don't "go", jump directly to "turn" part.

written by Chidong original link here

Solution 2

We can use Dijkstra's algorithm to find the shortest distance from the ball to the hole. If you are unfamiliar with this algorithm, how it works is that we process events in priority order, where the priority is (distance, path_string). When an event is processed, it adds neighboring nodes with respective distance. To repeatedly find the highest priority node to process, we use a heap (priority queue or 'pq'), where we can add nodes with logarithmic time complexity, and maintains the invariant that pq[0] is always the smallest (highest priority.) When we reach the hole for the first time (if we do), we are guaranteed to have the right answer in terms of having the shortest distance and the lexicographically smallest path-string.

When we look for the neighbors of a location in the matrix, we simulate walking up/left/right/down as long as we are inside the bounds of the matrix and the path is clear. If during this simulation we reach the hole prematurely, we should also stop. If after searching with our algorithm it is the case that we never reached the hole, then the task is impossible.

```
def findShortestWay(self, A, ball, hole):
    ball, hole = tuple(ball), tuple(hole)
    R, C = len(A), len(A[0])

    def neighbors(r, c):
        for dr, dc, di in [(-1, 0, 'u'), (0, 1, 'r'),
                           (0, -1, 'l'), (1, 0, 'd')]:
            cr, cc, dist = r, c, 0
            while (0 <= cr + dr < R and
                    0 <= cc + dc < C and
                    not A[cr+dr][cc+dc]):
                cr += dr
                cc += dc
                dist += 1
                if (cr, cc) == hole:
                    break
            yield (cr, cc), di, dist

    pq = [(0, '', ball)]
    seen = set()
    while pq:
        dist, path, node = heapq.heappop(pq)
        if node in seen: continue
        if node == hole: return path
        seen.add(node)
        for nei, di, nei_dist in neighbors(*node):
            heapq.heappush(pq, (dist+nei_dist, path+di, nei) )

    return "impossible"
```

written by awice original link here

Solution 3

I am rolling the ball from the stopping point recursively in the down-left-right-up order, and keeping track of the path with the shortest distance. That way, the first discovered path will be lexicographically smallest (if distance is the same). Except the very first roll, the ball is rolled only vertically (down and up) or horizontally (left and right), alternating.

When the ball stops, I am using the existing maze vector to store the current distance. If the ball was there before, I only continue the search if the current distance is smaller.

```
string roll(vector<vector<int>>& maze, int rowBall, int colBall, const vector<int>
& hole,
    int d_row, int d_col, int steps, const string& path, pair<string, int>& res)
{
    if (steps < res.second) {
        if (d_row != 0 || d_col != 0) { // both are zero for the initial ball pos
ition.
            while ((rowBall + d_row) >= 0 && (colBall + d_col) >= 0 && (rowBall +
d_row) <  maze.size()
                && (colBall + d_col) < maze[0].size() && maze[rowBall + d_row][co
lBall + d_col] != 1)
            {
                rowBall += d_row;
                colBall += d_col;
                ++steps;
                if (rowBall == hole[0] && colBall == hole[1] && steps < res.secon
d) res = {path, steps};
            }
        }
        if (maze[rowBall][colBall] == 0 || steps + 2 < maze[rowBall][colBall]) {
            maze[rowBall][colBall] = steps + 2; // '1' is for the walls.
            if (d_row == 0) roll(maze, rowBall, colBall, hole, 1, 0, steps, path
+ "d", res);
            if (d_col == 0) roll(maze, rowBall, colBall, hole, 0, -1, steps, path
+ "l", res);
            if (d_col == 0) roll(maze, rowBall, colBall, hole, 0, 1, steps, path
+ "r", res);
            if (d_row == 0) roll(maze, rowBall, colBall, hole, -1, 0, steps, path
+ "u", res);
        }
    }
    return res.first;
}
string findShortestWay(vector<vector<int>>& maze, vector<int>& ball, vector<int>&
hole)
{
    return roll(maze, ball[0], ball[1], hole, 0, 0, 0, "", pair<string, int>() =
{"impossible", INT_MAX});
}
```

written by votrubac original link here

From Bootloader?