

Problem 499: The Maze III

Problem Information

Difficulty: **Hard**

Acceptance Rate: 0.00%

Paid Only: No

Problem Description

There is a ball in a

maze

with empty spaces (represented as

0

) and walls (represented as

1

). The ball can go through the empty spaces by rolling

up, down, left or right

, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction (must be different from last chosen direction). There is also a hole in this maze. The ball will drop into the hole if it rolls onto the hole.

Given the

$m \times n$

maze

, the ball's position

ball

and the hole's position

hole

, where

ball = [ball

row

, ball

col

]

and

hole = [hole

row

, hole

col

]

, return

a string

instructions

of all the instructions that the ball should follow to drop in the hole with the

shortest distance

possible

. If there are multiple valid instructions, return the

lexicographically minimum

one. If the ball can't drop in the hole, return

"impossible"

.

If there is a way for the ball to drop in the hole, the answer

instructions

should contain the characters

'u'

(i.e., up),

'd'

(i.e., down),

'l'

(i.e., left), and

'r'

(i.e., right).

The

distance

is the number of

empty spaces

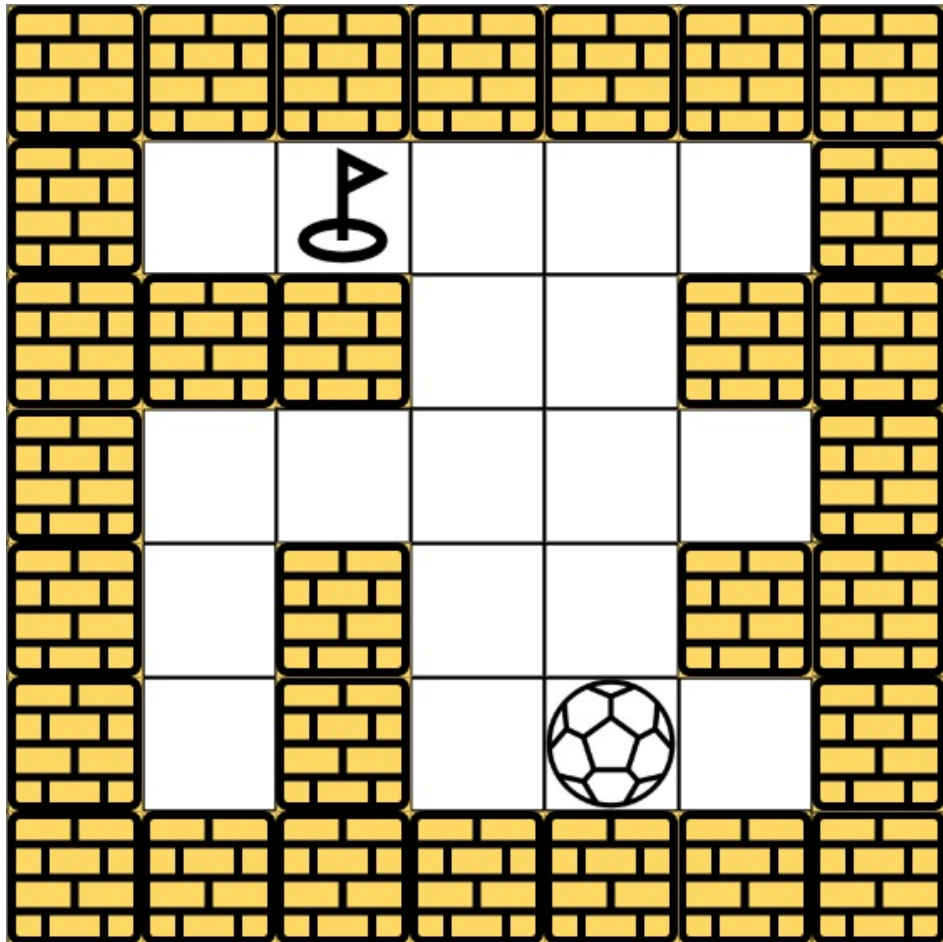
traveled by the ball from the start position (excluded) to the destination (included).

You may assume that

the borders of the maze are all walls

(see examples).

Example 1:



Input:

```
maze = [[0,0,0,0,0,0],[1,1,0,0,1],[0,0,0,0,0],[0,1,0,0,1],[0,1,0,0,0]], ball = [4,3], hole = [0,1]
```

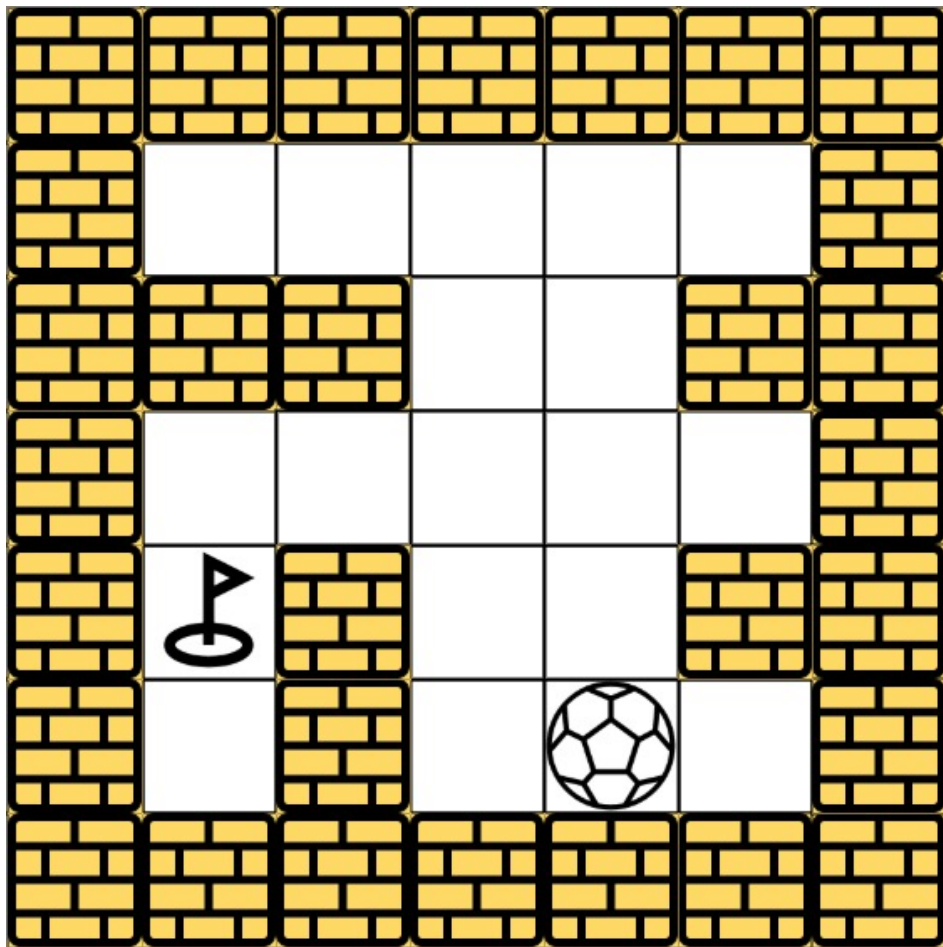
Output:

"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' < 'u'. So the output is "lul".

Example 2:



Input:

maze = [[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]], ball = [4,3], hole = [3,0]

Output:

"impossible"

Explanation:

The ball cannot reach the hole.

Example 3:

Input:

maze = [[0,0,0,0,0,0,0],[0,0,1,0,0,1,0],[0,0,0,0,1,0,0],[0,0,0,0,0,0,1]], ball = [0,4], hole = [3,5]

Output:

"dldr"

Constraints:

m == maze.length

n == maze[i].length

1 <= m, n <= 100

maze[i][j]

is

0

or

1

.

ball.length == 2

hole.length == 2

0 <= ball

row

, hole

row

<= m

0 <= ball

col

, hole

col

<= n

Both the ball and the hole exist in an empty space, and they will not be in the same position initially.

The maze contains

at least 2 empty spaces

.

Code Snippets

C++:

```

class Solution {
public:
    string findShortestWay(vector<vector<int>>& maze, vector<int>& ball,
vector<int>& hole) {

    }
};

```

Java:

```

class Solution {
    public String findShortestWay(int[][] maze, int[] ball, int[] hole) {

    }
}

```

Python3:

```

class Solution:
    def findShortestWay(self, maze: List[List[int]], ball: List[int], hole:
List[int]) -> str:

```

Python:

```

class Solution(object):
    def findShortestWay(self, maze, ball, hole):
        """
        :type maze: List[List[int]]
        :type ball: List[int]
        :type hole: List[int]
        :rtype: str
        """

```

JavaScript:

```

/**
 * @param {number[][]} maze
 * @param {number[]} ball
 * @param {number[]} hole
 * @return {string}
 */
var findShortestWay = function(maze, ball, hole) {

```

```
};
```

TypeScript:

```
function findShortestWay(maze: number[][], ball: number[], hole: number[]):  
string {  
  
};
```

C#:

```
public class Solution {  
    public string FindShortestWay(int[][] maze, int[] ball, int[] hole) {  
  
    }  
}
```

C:

```
char* findShortestWay(int** maze, int mazeSize, int* mazeColSize, int* ball,  
int ballSize, int* hole, int holeSize) {  
  
}
```

Go:

```
func findShortestWay(maze [][]int, ball []int, hole []int) string {  
  
}
```

Kotlin:

```
class Solution {  
    fun findShortestWay(maze: Array<IntArray>, ball: IntArray, hole: IntArray):  
    String {  
  
    }  
}
```

Swift:

```

class Solution {
    func findShortestWay(_ maze: [[Int]], _ ball: [Int], _ hole: [Int]) -> String
    {

    }

}

```

Rust:

```

impl Solution {
    pub fn find_shortest_way(maze: Vec<Vec<i32>>, ball: Vec<i32>, hole: Vec<i32>)
    -> String {

    }

}

```

Ruby:

```

# @param {Integer[][]} maze
# @param {Integer[]} ball
# @param {Integer[]} hole
# @return {String}
def find_shortest_way(maze, ball, hole)

end

```

PHP:

```

class Solution {

    /**
     * @param Integer[][] $maze
     * @param Integer[] $ball
     * @param Integer[] $hole
     * @return String
     */
    function findShortestWay($maze, $ball, $hole) {

    }

}

```

Dart:

```

class Solution {
  String findShortestWay(List<List<int>> maze, List<int> ball, List<int> hole)
  {

  }
}

```

Scala:

```

object Solution {
  def findShortestWay(maze: Array[Array[Int]], ball: Array[Int], hole:
  Array[Int]): String = {

  }
}

```

Elixir:

```

defmodule Solution do
  @spec find_shortest_way(maze :: [[integer]], ball :: [integer], hole ::
  [integer]) :: String.t
  def find_shortest_way(maze, ball, hole) do

  end
end

```

Erlang:

```

-spec find_shortest_way(Maze :: [[integer()]], Ball :: [integer()], Hole ::
[integer()]) -> unicode:unicode_binary().
find_shortest_way(Maze, Ball, Hole) ->
.

```

Racket:

```

(define/contract (find-shortest-way maze ball hole)
  (-> (listof (listof exact-integer?)) (listof exact-integer?) (listof
exact-integer?) string?)
  )

```

Solutions

C++ Solution:

```
/*
 * Problem: The Maze III
 * Difficulty: Hard
 * Tags: array, string, graph, search, queue, heap
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

class Solution {
public:
    string findShortestWay(vector<vector<int>>& maze, vector<int>& ball,
        vector<int>& hole) {

    }
};
```

Java Solution:

```
/**
 * Problem: The Maze III
 * Difficulty: Hard
 * Tags: array, string, graph, search, queue, heap
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

class Solution {
    public String findShortestWay(int[][] maze, int[] ball, int[] hole) {

    }
}
```

Python3 Solution:

```
"""
Problem: The Maze III
Difficulty: Hard
```

```
Tags: array, string, graph, search, queue, heap
```

```
Approach: Use two pointers or sliding window technique
```

```
Time Complexity:  $O(n)$  or  $O(n \log n)$ 
```

```
Space Complexity:  $O(1)$  to  $O(n)$  depending on approach
```

```
"""
```

```
class Solution:
```

```
def findShortestWay(self, maze: List[List[int]], ball: List[int], hole:  
List[int]) -> str:
```

```
# TODO: Implement optimized solution
```

```
pass
```

Python Solution:

```
class Solution(object):
```

```
def findShortestWay(self, maze, ball, hole):
```

```
"""
```

```
:type maze: List[List[int]]
```

```
:type ball: List[int]
```

```
:type hole: List[int]
```

```
:rtype: str
```

```
"""
```

JavaScript Solution:

```
/**
```

```
 * Problem: The Maze III
```

```
 * Difficulty: Hard
```

```
 * Tags: array, string, graph, search, queue, heap
```

```
 *
```

```
 * Approach: Use two pointers or sliding window technique
```

```
 * Time Complexity:  $O(n)$  or  $O(n \log n)$ 
```

```
 * Space Complexity:  $O(1)$  to  $O(n)$  depending on approach
```

```
 */
```

```
/**
```

```
 * @param {number[][]} maze
```

```
 * @param {number[]} ball
```

```
 * @param {number[]} hole
```

```
 * @return {string}
```

```

*/
var findShortestWay = function(maze, ball, hole) {

};

```

TypeScript Solution:

```

/**
 * Problem: The Maze III
 * Difficulty: Hard
 * Tags: array, string, graph, search, queue, heap
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

function findShortestWay(maze: number[][], ball: number[], hole: number[]):
string {

};

```

C# Solution:

```

/*
 * Problem: The Maze III
 * Difficulty: Hard
 * Tags: array, string, graph, search, queue, heap
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

public class Solution {
    public string FindShortestWay(int[][] maze, int[] ball, int[] hole) {

    }
}

```

C Solution:

```

/*
 * Problem: The Maze III
 * Difficulty: Hard
 * Tags: array, string, graph, search, queue, heap
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

char* findShortestWay(int** maze, int mazeSize, int* mazeColSize, int* ball,
int ballSize, int* hole, int holeSize) {

}

```

Go Solution:

```

// Problem: The Maze III
// Difficulty: Hard
// Tags: array, string, graph, search, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func findShortestWay(maze [][]int, ball []int, hole []int) string {

}

```

Kotlin Solution:

```

class Solution {
    fun findShortestWay(maze: Array<IntArray>, ball: IntArray, hole: IntArray):
String {

    }
}

```

Swift Solution:

```

class Solution {
    func findShortestWay(_ maze: [[Int]], _ ball: [Int], _ hole: [Int]) -> String

```

```
{  
  
}  
}
```

Rust Solution:

```
// Problem: The Maze III  
// Difficulty: Hard  
// Tags: array, string, graph, search, queue, heap  
//  
// Approach: Use two pointers or sliding window technique  
// Time Complexity: O(n) or O(n log n)  
// Space Complexity: O(1) to O(n) depending on approach  
  
impl Solution {  
    pub fn find_shortest_way(maze: Vec<Vec<i32>>, ball: Vec<i32>, hole: Vec<i32>)  
        -> String {  
  
    }  
}
```

Ruby Solution:

```
# @param {Integer[][]} maze  
# @param {Integer[]} ball  
# @param {Integer[]} hole  
# @return {String}  
def find_shortest_way(maze, ball, hole)  
  
end
```

PHP Solution:

```
class Solution {  
  
    /**  
     * @param Integer[][] $maze  
     * @param Integer[] $ball  
     * @param Integer[] $hole  
     * @return String  
     */  
}
```

```
function findShortestWay($maze, $ball, $hole) {

}

}
```

Dart Solution:

```
class Solution {
String findShortestWay(List<List<int>> maze, List<int> ball, List<int> hole)
{

}

}
```

Scala Solution:

```
object Solution {
def findShortestWay(maze: Array[Array[Int]], ball: Array[Int], hole:
Array[Int]): String = {

}

}
```

Elixir Solution:

```
defmodule Solution do
@spec find_shortest_way(maze :: [[integer]], ball :: [integer], hole ::
[integer]) :: String.t
def find_shortest_way(maze, ball, hole) do

end

end
```

Erlang Solution:

```
-spec find_shortest_way(Maze :: [[integer()]], Ball :: [integer()], Hole ::
[integer()]) -> unicode:unicode_binary().
find_shortest_way(Maze, Ball, Hole) ->
.
```

Racket Solution:

```
(define/contract (find-shortest-way maze ball hole)
  (-> (listof (listof exact-integer?)) (listof exact-integer?) (listof
exact-integer?) string?)
)
```