Description

490. The Maze

There is a ball in a maze with empty spaces and walls. The ball can go through 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.

Given the ball's start position, the destination and the maze, determine whether the ball could stop at the destination.

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 start and destination coordinates are represented by row and column indexes.

```
Example 1

Input 1: a maze represented by a 2D array

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

Input 2: start coordinate (rowStart, colStart) = (0, 4)
Input 3: destination coordinate (rowDest, colDest) = (4, 4)

Output: true
Explanation: One possible way is :
left -> down -> left -> down -> right -> down -> right.
```

```
Example 2

Input 1: a maze represented by a 2D array

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

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

Input 2: start coordinate (rowStart, colStart) = (0, 4)
Input 3: destination coordinate (rowDest, colDest) = (3, 2)

Output: false
Explanation: There is no way for the ball to stop at the destination.
```

Idea Report

This can be represented as a search in a directed graph. Each position the ball can stop will be vertex and the moving path from one location to another is the edge. Because the ball must hit a wall to stop so ball can go from one vertex A to another vertex B not necessarily means ball can go from B to A. Take example 1, [1,1] can go [0,1], but [0,1] cannot go [1,1], so this is a directed graph. The ball has a start and destination location, so this becomes a search problem in the directed graph. We can firstly using BFS, while searching, if the current location is the destination we return true. If we search all the possible locations and still cannot reach to the destination then we return false.

Code

```java
class Solution {
    // BFS AC
    public boolean hasPath(int[][] maze, int[] start, int[] destination) {
        int rows = maze.length;
        int cols = maze[0].length;
        int[][] visited = new int[rows][cols];
        Deque<int[]> q = new ArrayDeque<>();
        q.offer(start);
        visited[start[0]][start[1]] = 1;
        int[] dx = {0, 0, 1, -1};
        int[] dy = {1, -1, 0, 0};

        while (!q.isEmpty()) {
            int[] cur = q.poll();
            int r = cur[0];
            int c = cur[1];
            if (r == destination[0] && c == destination[1]) {
                return true;
            }
            for (int i = 0; i < dx.length; i++) {
```

```
                int x = r;
                int y = c;
                while (x >= 0 && x < rows && y >= 0 && y < cols
                       && maze[x][y] == 0) {
                    x += dx[i];
                    y += dy[i];
                }
                x -= dx[i];
                y -= dy[i];
                if (visited[x][y] == 0) {
                    q.offer(new int[]{x, y});
                    visited[x][y] = 1;
                }
            }
        }

        return false;
    }
}
```

We can also traverse using DFS
Code

```
class Solution {
    // DFS AC
    public boolean hasPath(int[][] maze, int[] start, int[] destination) {
        int[][] visited = new int[maze.length][maze[0].length];
        return helper(maze, start, destination, visited);
    }

    private boolean helper(int[][] maze, int[] start, int[] destination,
                           int[][] visited) {
        if (start[0] == destination[0] && start[1] == destination[1]) {
            return true;
        }

        if (visited[start[0]][start[1]] != 0) {
            return false;
        }

        visited[start[0]][start[1]] = 1;
        int[] dx = {0, 0, 1, -1};
        int[] dy = {1, -1, 0, 0};

        for (int i = 0; i < dx.length; i++) {
            int x = start[0];
            int y = start[1];
            while (0 <= x && x < maze.length && 0 <= y && y < maze[0].length
```

```
                    && maze[x][y] == 0) {
                x += dx[i];
                y += dy[i];
            }
            x -= dx[i];
            y -= dy[i];
            if (visited[x][y] == 0
                && helper(maze, new int[]{x, y}, destination, visited)) {
                return true;
            }
        }
        return false;
    }
}
```

Summary

- Directed graph search problem
- Time O(V+E), space O(E). (If the maze is m x n, it is not possible that there are m x n vertices, so I can only say V + E.)

C++ implementation

```
class Solution {
public:
    // BFS AC
    bool hasPath(vector<vector<int>>& maze, vector<int>& start, vector<int>& destinati
        int rows = maze.size();
        int cols = maze[0].size();
        deque<vector<int>> q;
        q.push_back(start);
        int **visited = new int*[rows];
        for (int i = 0; i < rows; i++) {
            visited[i] = new int[cols]{0};
        }

        visited[start[0]][start[1]] = 1;
        vector<int> dx = {0,0,1,-1};
        vector<int> dy = {1,-1,0,0};

        while (!q.empty()) {
            vector<int> cur = q.front();
            q.pop_front();
```

```
            if (cur[0] == destination[0] && cur[1] == destination[1]) {
                return true;
            }
            for (int i = 0; i < dx.size(); i++) {
                int x = cur[0];
                int y = cur[1];

                while (0 <= x && x < maze.size() && 0 <= y
                       && y < maze[0].size() && maze[x][y] == 0) {
                    x += dx[i];
                    y += dy[i];
                }

                x -= dx[i];
                y -= dy[i];

                if (visited[x][y] == 0) {
                    visited[x][y] = 1;
                    vector<int> next = {x,y};
                    q.push_back(next);
                }
            }
        }

        delete visited;
        return false;
    }
};
```

```
class Solution {
public:
    // DFS AC
    vector<int> dx = {0,0,1,-1};
    vector<int> dy = {1,-1,0,0};

    bool hasPath(vector<vector<int>>& maze, vector<int>& start, vector<int>& destinati
        int rows = maze.size();
        int cols = maze[0].size();
        int **visited = new int*[rows];
        for (int i = 0; i < rows; i++) {
            visited[i] = new int[cols]{0};
        }

        return helper(maze, start, destination, visited);
    }

    bool helper(vector<vector<int>>& maze, vector<int>& start, vector<int>& destinatio
        if (start[0] == destination[0] && start[1] == destination[1]) {
```

```
            return true;
        }

        if (visited[start[0]][start[1]] == 1) {
            return false;
        }

        visited[start[0]][start[1]] = 1;
        int rows = maze.size();
        int cols = maze[0].size();

        for (int i = 0; i < dx.size(); i++) {
            int x = start[0];
            int y = start[1];
            while (0 <= x && x < rows && 0 <= y && y < cols && maze[x][y] == 0) {
                x += dx[i];
                y += dy[i];
            }
            x -= dx[i];
            y -= dy[i];
            if (visited[x][y] == 0) {
                vector<int> next = {x, y};
                if (helper(maze, next, destination, visited)) {
                    return true;
                }
            }
        }

        return false;
    }
}
```