Technique: Backtracking

Level: Medium

Maze Problem: You are given a 2D array that represents a maze. It can have 2 values - 0 and 1. 1 represents a wall and 0 represents a path.

The objective is to reach the bottom right corner, i.e, A[A.length-1][A.length-1]. You start from A[0][0]. You can move in 4 directions - up, down, left and right. Find if a path exists to the bottom right of the maze.

For example, a path exists in the following maze:

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

Questions to Clarify:
Q. How do you want the output?
A. Return *true* if a path exists.

Q. Does it matter if the end is a path or a wall?
A. It doesn't matter, the last element (*A[A.length-1][A.length-1]*) can be anything. You just have to get there.

Q. What if the array is empty or null?
A. Return *false* (no path exists).

Q. What if the array has just one element, e.g, *{0}* or *{1}*.
A. Return *true*, because we're already at the last element.

Solution:
We use Recursion with Backtracking. In Backtracking, we try one path, and if it doesn't lead to the result, we try another path.

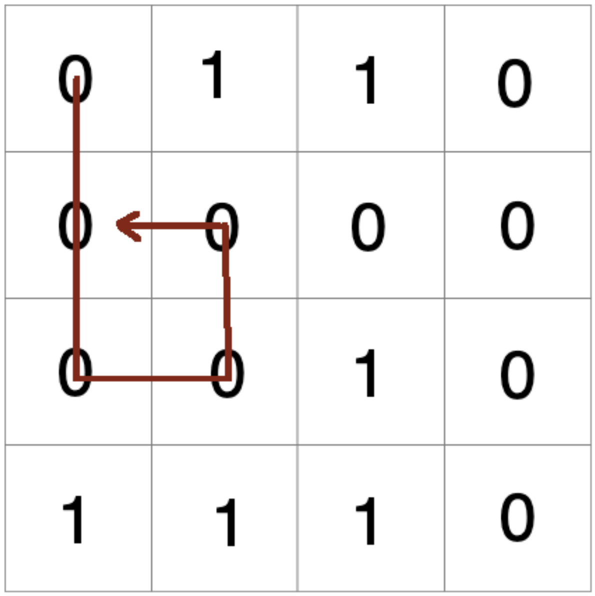
For each element *a[i][j]*, we try all 4 directions.

For example, if we go down, we go to the element below and see if there is a path from there (to the end).

For each path, we keep looking until we encounter:
1. a wall
2. the array boundary
3. the end of the maze (A[A.length-1][A.length-1]).

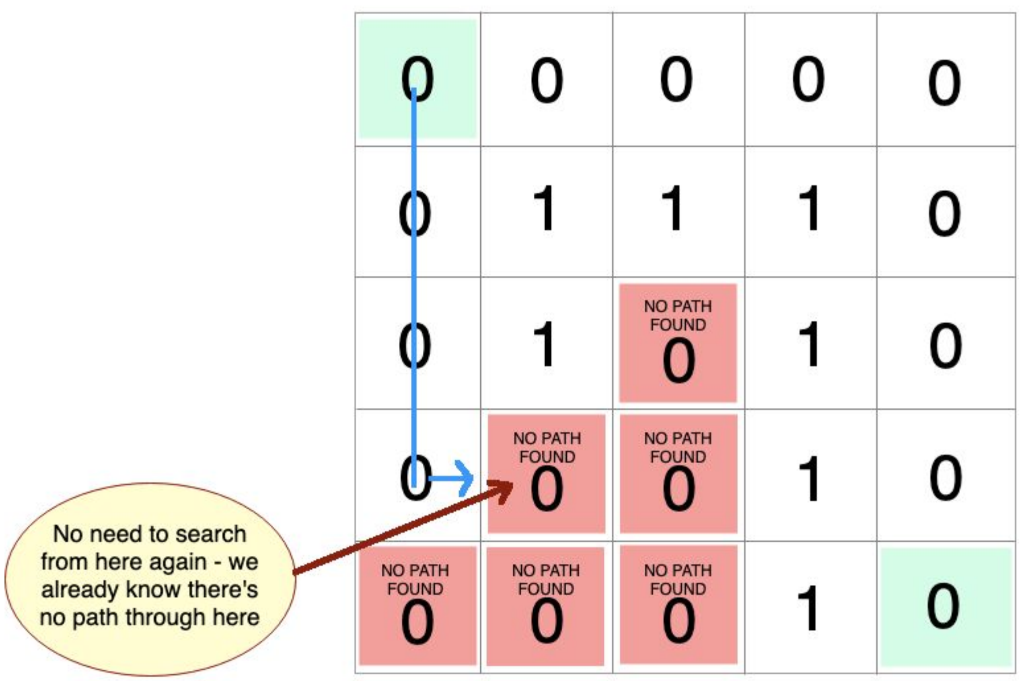
If we encounter 1 & 2, we return false, because there is no path from there. If we encounter the end of the maze, we return true.

There's one problem we haven't addressed - avoiding cycles.



We need some way to tell that *a[i][j]* is already on our current path. Otherwise, we will go around in circles forever.

Additionally, we can add memoization here. If we know that *a[i][j]* doesn't lead to a path, we don't need to search again. We can save that info.



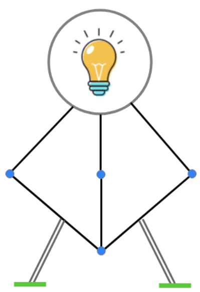
To do this, we can assign one of 3 possible states to each element: UNVISITED, VISITING or NO_PATH_FOUND.

When we are searching from *a[i][j]*, we can set it to VISITING. This will prevent cycles. If you're familiar with graph search, we use a similar concept there.

After processing *a[i][j]* and finding no path to the end, we update the state to NO_PATH_FOUND. In the future, if we come back to *a[i][j]*, we won't have to do this recursion again.

Note: Technically, you can add another state - FOUND_PATH, if we successfully found a path through *a[i][j]*. However, it's not needed here, because we end the program as soon as we find the path. It's up to you to add it for completeness or readability.

Backtracking Spaceship:



Core Idea	From every *a[i][j]*, check if there is a path to the end.
Steps/Splits	Check from left element, right element, up and down.
Converge/Collect	If any of the checks returns *true*, return *true*.
Memoization	Can we memoize? Yes, for every element, we can save three states: UNVISITED, VISITING, NO_PATH_FOUND.
Base Cases	*a[i][j]* is Out of Bounds, Wall,or the last element

Pseudocode:

```
(Note: Never write pseudocode in an actual interview. Unless you're writing a
few lines quickly to plan out your solution. Your actual solution should be in
a real language and use good syntax.)

boolean pathExists(a, i, j, memo)
     if i,j is out of bounds or a wall
```

```
        return false
    if i,j is end of maze
        return true
    if memo[i][j] is VISITING or NO_PATH_FOUND
        return false

    set memo[i][j] to VISITING

    Find 4 points around (i,j)
    for each point
        check if path exists. Return true if it does

    (If we got here, path doesn't exist)
    set memo[i][j] to NO_PATH_FOUND
    return false
```

<u>Test Cases:</u>
Edge Cases: matrix is empty or null, single element (1 & 0)
Base Cases: matrix with 1 row/column
Regular Cases: matrix with all walls, matrix with no walls, matrix with/without path to end, square matrix, rectangular matrix

<u>Time Complexity:</u> $O(4^n)$ without memoization, $O(n)$ with memoization, where *n* is the number of elements in the matrix.

Without memoization, we do 4 splits every time, and we do that at most *n* times. So the time complexity can be $O(4^n)$

With memoization, we process each node only once, so the time complexity is $O(n)$.

<u>Space Complexity:</u> $O(n^2)$ on both the memo and the recursion stack

Why does the recursion stack take $O(n^2)$ space?

Take the case where the path spirals through the maze: we will have roughly $n^2/2$ function calls from start to end.

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

```
public static boolean pathExists(int[][] a) {
    if (a == null || a.length == 0)
        return false;

    State[][] memo = new State[a.length][a[0].length];

    for (State[] states: memo)
        Arrays.fill(states, State.UNVISITED);

    return pathExists(a, 0, 0, memo);
}

public static boolean pathExists(int[][] a, int i, int j, State[][] memo) {
    if (oob(a, i, j) || a[i][j] == 1)
        return false;

    if (i == a.length - 1 && j == a[0].length - 1) // end of maze
        return true;

    if (memo[i][j] == State.NO_PATH_FOUND || memo[i][j] == State.VISITING)
        return false;
    memo[i][j] = State.VISITING;

    Pair[] points = {
            new Pair(i+1,j),
            new Pair(i-1,j),
            new Pair(i,j+1),
            new Pair(i,j-1)
        };

    for (Pair point : points) {
        if (pathExists(a, point.getFirst(), point.getSecond(), memo)) {
            return true;
        }
    }

    memo[i][j] = State.NO_PATH_FOUND;
    return false;
}

/*
 * Helper code. Ask if your interviewer if they want you to implement this.
 */

public enum State {
    UNVISITED,
    VISITING,
```

```
    NO_PATH_FOUND;
}

private static boolean oob(int[][] a, int i, int j) {
    return i < 0 || i >= a.length || j < 0 || j >= a[0].length;
}
```