Rat in a Maze Problem

https://www.geeksforgeeks.org/problems/rat-in-a-maze-problem/1

```
import java.util.ArrayList;
import java.util.Collections;

class Solution {
    private static final int[] row = {1, 0, 0, -1}; // D, L, R, U
    private static final int[] col = {0, -1, 1, 0}; // D, L, R, U
    private static final char[] move = {'D', 'L', 'R', 'U'};

    public static ArrayList<String> findPath(int[][] m, int n) {
        ArrayList<String> result = new ArrayList<>();
        if (m[0][0] == 0 || m[n-1][n-1] == 0) return result;
        backtrack(m, n, 0, 0, new boolean[n][n], new StringBuilder(), result);
        Collections.sort(result);
        return result;
    }

    private static void backtrack(int[][] m, int n, int x, int y, boolean[][] visited, StringBuilder
path, ArrayList<String> result) {
        if (x == n-1 && y == n-1) {
            result.add(path.toString());
            return;
        }
```

```
        visited[x][y] = true;
        for (int i = 0; i < 4; i++) {
            int nx = x + row[i], ny = y + col[i];
            if (isSafe(m, n, nx, ny, visited)) {
                path.append(move[i]);
                backtrack(m, n, nx, ny, visited, path, result);
                path.deleteCharAt(path.length() - 1);
            }
        }
        visited[x][y] = false;
    }

    private static boolean isSafe(int[][] m, int n, int x, int y, boolean[][] visited) {
        return x >= 0 && x < n && y >= 0 && y < n && m[x][y] == 1 && !visited[x][y];
    }
}
```

BEST EXPLANATION : ----------

Import Statements

java
Copy code
```
import java.util.ArrayList;
import java.util.Collections;
```

- **ArrayList** is used for dynamic arrays, which can grow as needed. Here, it is used to store all valid paths.

- **Collections** provides utility methods for operating on collections, like sorting the list of paths lexicographically.

Solution Class

java
Copy code
```
class Solution {
```

- This is the class definition for **Solution** which contains all the methods for solving the problem.

Direction Arrays

java
Copy code
```
  private static final int[] row = {1, 0, 0, -1}; // D, L, R, U
   private static final int[] col = {0, -1, 1, 0}; // D, L, R, U
   private static final char[] move = {'D', 'L', 'R', 'U'};
```

- **row** and **col** arrays define the direction of movement in terms of changes in row and column indices. The order of directions corresponds to:
  - Down (**D**) with **(1, 0)**
  - Left (**L**) with **(0, -1)**
  - Right (**R**) with **(0, 1)**
  - Up (**U**) with **(-1, 0)**
- **move** array holds the corresponding characters for each direction.

findPath Method

java
Copy code
```
  public static ArrayList<String> findPath(int[][] m, int n) {
      ArrayList<String> result = new ArrayList<>();
      if (m[0][0] == 0 || m[n-1][n-1] == 0) return result;
      backtrack(m, n, 0, 0, new boolean[n][n], new StringBuilder(), result);
      Collections.sort(result);
      return result;
   }
```

- **Function Signature**: This method takes a 2D matrix **m** and an integer **n** which is the size of the matrix.
- **Result Initialization**: **ArrayList<String> result** is created to store all the paths.
- **Base Case Check**: It checks if the starting cell **(0, 0)** or the destination cell **(n-1, n-1)** is blocked (i.e., **0**). If either is blocked, it returns an empty list because no path is possible.
- **Backtracking Call**: It calls the **backtrack** method starting from **(0, 0)** with a boolean **visited** matrix to keep track of visited cells and a **StringBuilder** to build the path string.
- **Sorting**: After finding all possible paths, it sorts them lexicographically.
- **Return Result**: Finally, it returns the sorted list of paths.

backtrack Method

java
Copy code

```
  private static void backtrack(int[][] m, int n, int x, int y, boolean[][] visited, StringBuilder
path, ArrayList<String> result) {
      if (x == n-1 && y == n-1) {
         result.add(path.toString());
         return;
      }
      visited[x][y] = true;
      for (int i = 0; i < 4; i++) {
         int nx = x + row[i], ny = y + col[i];
         if (isSafe(m, n, nx, ny, visited)) {
            path.append(move[i]);
            backtrack(m, n, nx, ny, visited, path, result);
            path.deleteCharAt(path.length() - 1);
         }
      }
      visited[x][y] = false;
   }
```

- **Function Signature**: This method is a recursive function for exploring paths. It takes the current cell coordinates **(x, y)**, the **visited** matrix, the current **path**, and the **result** list as arguments.
- **Base Case**: If the current cell **(x, y)** is the destination **(n-1, n-1)**, it adds the current path to the result list and returns.
- **Mark Visited**: It marks the current cell **(x, y)** as visited.

- **Explore Moves**: It iterates over all four possible moves (down, left, right, up). For each move:
    - It calculates the new cell coordinates **(nx, ny)**.
    - It checks if the new cell is safe to move to using the **isSafe** method.
    - If safe, it appends the move character to the path and recursively calls **backtrack** for the new cell.
    - After returning from recursion, it removes the last character from the path (backtracking step).
- **Unmark Visited**: After exploring all moves from the current cell, it marks the current cell as not visited (backtracking step).

isSafe Method

java
Copy code
```
  private static boolean isSafe(int[][] m, int n, int x, int y, boolean[][] visited) {
      return x >= 0 && x < n && y >= 0 && y < n && m[x][y] == 1 && !visited[x][y];
   }
}
```

- **Function Signature**: This method checks if a cell **(x, y)** is within the matrix bounds, not blocked, and not already visited.
- **Boundary Check**: It ensures that **x** and **y** are within the range **[0, n-1]**.
- **Block Check**: It checks if the cell value **m[x][y]** is **1** (open).
- **Visited Check**: It checks if the cell has not been visited yet.
- **Return**: It returns **true** if all conditions are met, otherwise returns **false**.

Summary

- The **Solution** class solves the "Rat in a Maze" problem using backtracking.
- It explores all paths from the source to the destination.
- It uses helper methods and direction arrays to manage movements and ensure valid paths.
- The paths are collected in a list, sorted lexicographically, and returned.

This code efficiently finds all possible paths in the maze while ensuring no cell is visited more than once in a single path. The use of backtracking allows it to explore all potential paths and backtrack when it reaches dead-ends.

Time Complexity (TC)

The time complexity of the backtracking solution is determined by how many potential paths it explores and how much work it does for each path.

1. **Exploration of Paths**:
    - The rat can move in four possible directions (down, left, right, up) from each cell.
    - In the worst case, it could explore all possible paths in the matrix.
2. **Recursive Calls**:
    - Each cell is visited at most once in a single path, but the function can backtrack and revisit the cell in different paths.
    - For a matrix of size $N \times N$N×N, the worst-case number of recursive calls can be up to 4^{N}4N, where NN is the length of the path in the worst case (though this is highly theoretical and not achievable practically due to the constraints on valid movements).
3. **Check and Update**:
    - Each recursive call performs constant-time operations to check boundaries, update the path, and mark cells as visited.

Given that the problem's constraints specify $N \leq 5$N≤5, the number of paths will be relatively small. However, for large NN, the theoretical upper bound is: $O(3^{N^2})$O(3N2) This is because each cell has at most 3 other directions to explore after excluding the direction it came from.

Space Complexity (SC)

The space complexity includes both the call stack used by recursion and the space used for storing results.

4. **Call Stack**:
    - The maximum depth of the recursion is N^2N2 in the worst case (visiting all cells in a single path).
    - Therefore, the maximum call stack depth is $O(N^2)$O(N2).
5. **Visited Matrix**:
    - The **visited** boolean matrix takes $O(N^2)$O(N2) space.
6. **Path Storage**:
    - The space used to store all the paths in the **result** list.
    - In the worst case, if there are many paths, the total space used by the result list could be significant, but it is typically $O(N^2 \cdot P)$O(N2·P), where PP is the number of paths.

7. **Path StringBuilder**:
    - The **StringBuilder** used in recursion takes $O(N^2)$ space in the worst case (if the path length is maximum).

Summary

Time Complexity: $O(3N^2)$

- The time complexity can be seen as exponential in the worst case due to the recursive exploration of all potential paths.

Space Complexity: $O(N^2)$

- The space complexity is primarily due to the call stack and the **visited** matrix, both of which are $O(N^2)$.