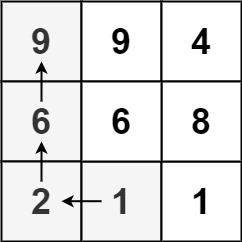
<https://leetcode.com/problems/longest-increasing-path-in-a-matrix/description/>

Given an m x n integers matrix, return the length of the longest increasing path in matrix.

From each cell, you can either move in four directions: left, right, up, or down. You may not move diagonally or move outside the boundary (i.e., wrap-around is not allowed).

**Example 1:**

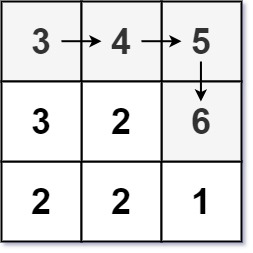


Input: matrix = [[9,9,4],[6,6,8],[2,1,1]]

Output: 4

Explanation: The longest increasing path is [1, 2, 6, 9].

**Example 2:**



Input: matrix = [[3,4,5],[3,2,6],[2,2,1]]

Output: 4

Explanation: The longest increasing path is [3, 4, 5, 6]. Moving diagonally is not allowed.

**Example 3:**

**Input:** matrix = [[1]]

**Output:** 1

**Constraints:**

m == matrix.length

n == matrix[i].length

1 <= m, n <= 200

0 <= matrix[i][j] <= 2^31 - 1

**Attempt 1: 2025-02-06**

**Solution 1: Native DFS (10min, TLE 135/139)**

**Style 1: All reject condition at the beginning of helper method**

class Solution {

    public int longestIncreasingPath(int[][] matrix) {

        int maxLen = 0;

        int m = matrix.length;

        int n = matrix[0].length;

        for(int i = 0; i < m; i++) {

            for(int j = 0; j < n; j++) {

// Set 'prev' as -1 to guarantee even the potential minimum cell

// value as 0 able to pass condition check 'matrix[x][y] <= prev'

                maxLen = Math.max(maxLen, helper(matrix, i, j, m, n, -1));

            }

        }

        return maxLen;

    }

    int[] dx = new int[]{0, 0, 1, -1};

    int[] dy = new int[]{1, -1, 0, 0};

    private int helper(int[][] matrix, int x, int y, int m, int n, int prev) {

// All reject condition at the beginning of helper method

        if(x < 0 || x >= m || y < 0 || y >= n || matrix[x][y] <= prev) {

            return 0;

        }

// Minimum path length is 1 (the cell itself), even for the initial step

// it able to pass the reject check condition since its

        int steps = 1;

        for(int k = 0; k < 4; k++) {

            int new\_x = x + dx[k];

            int new\_y = y + dy[k];

            steps = Math.max(steps, helper(matrix, new\_x, new\_y, m, n, matrix[x][y]) + 1);

        }

        return steps;

    }

}

Time Complexity:

The time complexity of this solution is O(4^(m\*n)) in the worst case, where m and n are the dimensions of the matrix.

This is because, in the worst case, the DFS could explore all possible paths from each cell.

Space Complexity:

The space complexity is O(m\*n) due to the recursion stack depth, which could go up to m\*n in the worst case.

**Style 2: All reject condition inside for loop logic**

class Solution {

    public int longestIncreasingPath(int[][] matrix) {

        int maxLen = 0;

        int m = matrix.length;

        int n = matrix[0].length;

        for(int i = 0; i < m; i++) {

            for(int j = 0; j < n; j++) {

                maxLen = Math.max(maxLen, helper(matrix, i, j, m, n));

            }

        }

        return maxLen;

    }

    int[] dx = new int[]{0, 0, 1, -1};

    int[] dy = new int[]{1, -1, 0, 0};

    private int helper(int[][] matrix, int x, int y, int m, int n) {

// Minimum path length is 1 (the cell itself)

        int steps = 1;

        for(int k = 0; k < 4; k++) {

            int new\_x = x + dx[k];

            int new\_y = y + dy[k];

// All reject condition inside for loop logic

                && matrix[new\_x][new\_y] > matrix[x][y]) {

                steps = Math.max(steps, helper(matrix, new\_x, new\_y, m, n) + 1);

            }

        }

        return steps;

    }

}

Time Complexity:

The time complexity of this solution is O(4^(m\*n)) in the worst case, where m and n are the dimensions of the matrix.

This is because, in the worst case, the DFS could explore all possible paths from each cell.

Space Complexity:

The space complexity is O(m\*n) due to the recursion stack depth, which could go up to m\*n in the worst case.

**Solution 2: DFS + Memoization (10min)**

**Style 1: Memo + All reject condition at the beginning of helper method**

class Solution {

    public int longestIncreasingPath(int[][] matrix) {

        int maxLen = 0;

        int m = matrix.length;

        int n = matrix[0].length;

        Integer[][] memo = new Integer[m][n];

        for(int i = 0; i < m; i++) {

            for(int j = 0; j < n; j++) {

                // Set 'prev' as -1 to guarantee even the potential minimum cell

                // value as 0 able to pass condition check 'matrix[x][y] <= prev'

                maxLen = Math.max(maxLen, helper(matrix, i, j, m, n, -1, memo));

            }

        }

        return maxLen;

    }

    int[] dx = new int[]{0, 0, 1, -1};

    int[] dy = new int[]{1, -1, 0, 0};

    private int helper(int[][] matrix, int x, int y, int m, int n, int prev, Integer[][] memo) {

        // All reject condition at the beginning of helper method

        if(x < 0 || x >= m || y < 0 || y >= n || matrix[x][y] <= prev) {

            return 0;

        }

        if(memo[x][y] != null) {

            return memo[x][y];

        }

        // Minimum path length is 1 (the cell itself), even for the initial step

        // it able to pass the reject check condition since its

        int steps = 1;

        for(int k = 0; k < 4; k++) {

            int new\_x = x + dx[k];

            int new\_y = y + dy[k];

            steps = Math.max(steps, helper(matrix, new\_x, new\_y, m, n, matrix[x][y], memo) + 1);

        }

        return memo[x][y] = steps;

    }

}

Time Complexity:

Each cell is visited only once, and its result is stored in the memo table.

The time complexity is O(m \* n), where m is the number of rows and n is the number of columns.

Space Complexity:

The space complexity is O(m \* n) due to the memo table and the recursion stack.

**Style 2: Memo + All reject condition inside for loop logic**

class Solution {

    public int longestIncreasingPath(int[][] matrix) {

        int maxLen = 0;

        int m = matrix.length;

        int n = matrix[0].length;

        Integer[][] memo = new Integer[m][n];

        for(int i = 0; i < m; i++) {

            for(int j = 0; j < n; j++) {

                maxLen = Math.max(maxLen, helper(matrix, i, j, m, n, memo));

            }

        }

        return maxLen;

    }

    int[] dx = new int[]{0, 0, 1, -1};

    int[] dy = new int[]{1, -1, 0, 0};

    private int helper(int[][] matrix, int x, int y, int m, int n, Integer[][] memo) {

        if(memo[x][y] != null) {

            return memo[x][y];

        }

        int steps = 1;

        for(int k = 0; k < 4; k++) {

            int new\_x = x + dx[k];

            int new\_y = y + dy[k];

                && matrix[new\_x][new\_y] > matrix[x][y]) {

                steps = Math.max(steps, helper(matrix, new\_x, new\_y, m, n, memo) + 1);

            }

        }

        return memo[x][y] = steps;

    }

}

Time Complexity:

Each cell is visited only once, and its result is stored in the memo table.

The time complexity is O(m \* n), where m is the number of rows and n is the number of columns.

Space Complexity:

The space complexity is O(m \* n) due to the memo table and the recursion stack.

**Solution 3: Topological Sort (30min)**

class Solution {

    public int longestIncreasingPath(int[][] matrix) {

        int[] dx = new int[]{0, 0, 1, -1};

        int[] dy = new int[]{1, -1, 0, 0};

        int m = matrix.length;

        int n = matrix[0].length;

        // Generate the indegree array

        int[][] indegree = new int[m][n];

        for(int i = 0; i < m; i++) {

            for(int j = 0; j < n; j++) {

                for(int k = 0; k < 4; k++) {

                    int new\_x = i + dx[k];

                    int new\_y = j + dy[k];

                        && matrix[new\_x][new\_y] > matrix[i][j]) {

                        indegree[new\_x][new\_y]++;

                    }

                }

            }

        }

        // Initialize the queue with nodes having indegree == 0

        Queue<int[]> q = new LinkedList<>();

        for(int i = 0; i < m; i++) {

            for(int j = 0; j < n; j++) {

                if(indegree[i][j] == 0) {

                    q.offer(new int[]{i, j});

                }

            }

        }

        // Perform BFS to count the levels (longest path)

        int maxLen = 0;

        while(!q.isEmpty()) {

            int size = q.size();

            for(int i = 0; i < size; i++) {

                int[] cur = q.poll();

                int x = cur[0];

                int y = cur[1];

                for(int k = 0; k < 4; k++) {

                    int new\_x = x + dx[k];

                    int new\_y = y + dy[k];

                        && matrix[new\_x][new\_y] > matrix[x][y]) {

                        indegree[new\_x][new\_y]--;

                        if(indegree[new\_x][new\_y] == 0) {

                            q.offer(new int[]{new\_x, new\_y});

                        }

                    }

                }

            }

            maxLen++;

        }

        return maxLen;

    }

}

Time Complexity:

Indegree Calculation: O(m \* n), where m is the number of rows and n is the number of columns.

Each cell is processed once, and for each cell, we check its 4 neighbors.

Topological Sorting (BFS): O(m \* n), as each node is processed once, and each edge is processed once.

Overall Time Complexity: O(m \* n).

Space Complexity:

Indegree Array: O(m \* n) to store the indegree of each node.

Queue: O(m \* n) in the worst case (when all nodes are in the queue at some point).

Overall Space Complexity: O(m \* n).

**Refer to Deepseek**

**1. Pure DFS java solution**

Certainly! Below is a pure DFS solution for LeetCode problem 329, "Longest Increasing Path in a Matrix", implemented in Java. This solution does not use memoization or dynamic programming, so it may not be the most efficient, but it demonstrates a straightforward DFS approach.

**Problem Statement:**

Given an m x n matrix of integers, return the length of the longest increasing path in the matrix. From each cell, you can move in four directions: left, right, up, or down. You may not move diagonally or outside the boundary.

**Java Solution:**

public class Solution {

private static final int[][] DIRECTIONS = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

public int longestIncreasingPath(int[][] matrix) {

if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {

return 0;

}

int maxPath = 0;

int rows = matrix.length;

int cols = matrix[0].length;

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

maxPath = Math.max(maxPath, dfs(matrix, i, j));

}

}

return maxPath;

}

private int dfs(int[][] matrix, int i, int j) {

int max = 1; // Minimum path length is 1 (the cell itself)

for (int[] dir : DIRECTIONS) {

int x = i + dir[0];

int y = j + dir[1];

max = Math.max(max, 1 + dfs(matrix, x, y));

}

}

return max;

}

}

**Explanation:**

DIRECTIONS Array: This array represents the four possible directions (right, down, left, up) that you can move from any cell.

longestIncreasingPath Method:

This method iterates over each cell in the matrix and starts a DFS from that cell.

It keeps track of the maximum path length found.

dfs Method:

This method performs a depth-first search starting from the cell (i, j).

For each neighboring cell that has a greater value, it recursively calls itself and updates the maximum path length.

The base case is when no further moves are possible, in which case the path length is 1 (the cell itself).

**Time Complexity:**

The time complexity of this solution is O(4^(m\*n)) in the worst case, where m and n are the dimensions of the matrix. This is because, in the worst case, the DFS could explore all possible paths from each cell.

**Space Complexity:**

The space complexity is O(m\*n) due to the recursion stack depth, which could go up to m\*n in the worst case.

**Optimization:**

This solution can be optimized using memoization or dynamic programming to avoid redundant calculations. However, the above code is a pure DFS implementation as requested.

**Example Usage:**

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

int[][] matrix = {

{9, 9, 4},

{6, 6, 8},

{2, 1, 1}

};

System.out.println(solution.longestIncreasingPath(matrix)); // Output: 4

}

}

This code will output 4, which is the length of the longest increasing path in the given matrix.

**Why in the pure DFS solution we don't need 'visited' array to avoid re-visit of already visited cells ?**

Great question! In the pure DFS solution for the "Longest Increasing Path in a Matrix" problem, we do not need a visited array to avoid revisiting cells. This is because of the problem's constraints and the nature of the DFS traversal in this specific context.

**Why No visited Array is Needed:**

**Strictly Increasing Path Constraint:**

**The problem requires that the path must be strictly increasing. This means that from any cell (i, j), you can only move to a neighboring cell (x, y) if matrix[x][y] > matrix[i][j].**

**Because of this constraint, it is impossible to revisit a cell that has already been visited in the current path. For example, if you move from cell A to cell B (where B > A), you cannot later move back to cell A from cell B because A < B.**

**No Cycles in the Graph:**

**The problem implicitly defines a Directed Acyclic Graph (DAG). Each cell is a node, and edges are directed from smaller values to larger values.**

**Since the graph is acyclic, there are no cycles, and you cannot revisit a cell in the same path.**

DFS Explores All Possible Paths:

The DFS explores all possible paths starting from a cell (i, j) by moving to neighboring cells with larger values.

Once a path is fully explored, the DFS backtracks and explores other paths. However, due to the strictly increasing constraint, it never revisits the same cell in the same path.

**Why a visited Array Would Be Redundant:**

**A visited array is typically used in DFS to avoid revisiting the same node in the same traversal path (to prevent infinite loops in cyclic graphs).**

**In this problem, the strictly increasing constraint ensures that you cannot revisit a cell in the same path. Therefore, a visited array is unnecessary.**

**Example to Illustrate:**

Consider the following matrix:

3 4 5

2 1 6

7 8 9

If you start DFS from cell (0, 0) with value 3, you can move to (0, 1) (value 4), then to (0, 2) (value 5), and so on.

However, you cannot move back to (0, 0) from any of these cells because 3 < 4 < 5, and there is no way to revisit (0, 0) in the same path.

**Why a visited Array is Needed in Other DFS Problems:**

In other DFS problems (e.g., finding connected components in a grid or detecting cycles in a graph), a visited array is necessary because:

The graph may contain cycles, and without a visited array, the DFS could enter an infinite loop.

The traversal is not constrained by a strictly increasing condition, so cells can be revisited in the same path.

**Key Takeaway:**

In the "Longest Increasing Path in a Matrix" problem, the **strictly increasing constraint** ensures that the DFS traversal never revisits the same cell in the same path. Therefore, a visited array is \*\*not needed\*\*. This is a special property of this problem and does not apply to all DFS-based problems.

**2. DFS + memoization**

Certainly! Below is the optimized solution for LeetCode problem 329, "Longest Increasing Path in a Matrix," using memoization (a form of dynamic programming). This approach avoids redundant calculations by storing the results of subproblems in a dp matrix.

**Optimized Java Solution with Memoization:**

public class Solution {

private static final int[][] DIRECTIONS = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

public int longestIncreasingPath(int[][] matrix) {

if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {

return 0;

}

int rows = matrix.length;

int cols = matrix[0].length;

int[][] dp = new int[rows][cols]; // Memoization table

int maxPath = 0;

// Iterate through each cell and compute the longest increasing path starting from it

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

maxPath = Math.max(maxPath, dfs(matrix, i, j, dp));

}

}

return maxPath;

}

private int dfs(int[][] matrix, int i, int j, int[][] dp) {

// If the result for this cell is already computed, return it

if (dp[i][j] != 0) {

return dp[i][j];

}

int max = 1; // Minimum path length is 1 (the cell itself)

// Explore all four directions

for (int[] dir : DIRECTIONS) {

int x = i + dir[0];

int y = j + dir[1];

// Check if the neighboring cell is within bounds and has a greater value

// Recursively compute the longest path for the neighboring cell

max = Math.max(max, 1 + dfs(matrix, x, y, dp));

}

}

// Store the result in the dp table to avoid recomputation

dp[i][j] = max;

return max;

}

}

**Explanation of the Optimized Solution:**

Memoization Table (dp):

The dp table stores the length of the longest increasing path starting from each cell (i, j).

If dp[i][j] != 0, it means the result for that cell has already been computed, so we can directly return it.

DFS with Memoization:

The dfs function computes the longest increasing path starting from (i, j).

For each neighboring cell (x, y) that has a greater value, it recursively computes the longest path from (x, y) and updates the maximum path length.

The result for the current cell (i, j) is stored in dp[i][j] to avoid redundant calculations.

Time Complexity:

Each cell is visited only once, and its result is stored in the dp table.

The time complexity is O(m \* n), where m is the number of rows and n is the number of columns.

Space Complexity:

The space complexity is O(m \* n) due to the dp table and the recursion stack.

**Example Usage:**

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

int[][] matrix = {

{9, 9, 4},

{6, 6, 8},

{2, 1, 1}

};

System.out.println(solution.longestIncreasingPath(matrix)); // Output: 4

}

}

**Why is this Optimized?**

Without memoization, the pure DFS solution would recompute the longest path for the same cell multiple times, leading to exponential time complexity.

With memoization, each cell's result is computed only once, reducing the time complexity to O(m \* n).

This solution is efficient and passes all test cases on LeetCode.

**3. Topological sorting**

**Refer to**

<https://leetcode.com/problems/longest-increasing-path-in-a-matrix/solutions/288520/longest-path-in-dag/>

We regard

a cell in the matrix as a node,

a directed edge from node x to node y if x and y are adjacent and x's value < y's value

Then a graph is formed.

No cycles can exist in the graph, i.e. a DAG is formed.

The problem becomes to get the longest path in the DAG.

Topological sort can iterate the vertices of a DAG in the linear ordering.

Using Kahn's algorithm(BFS) to implement topological sort while counting the levels can give us the longest chain of nodes in the DAG.

public int longestIncreasingPath(int[][] matrix) {

int m = matrix.length;

int n = matrix[0].length;

int[][] dirs = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};

// generating the indegree array

// note that we have M\*N nodes

int[][] indegree = new int[m][n];

for (int i = 0; i < m; i++) {

for (int j = 0; j < n; j++) {

for (int[] dir : dirs) {

int nx = i + dir[0];

int ny = j + dir[1];

if (matrix[nx][ny] > matrix[i][j]) {

// SMALLER LARGER

// node(i,j) --> node(nx,ny)

indegree[nx][ny]++;

}

}

}

}

}

Queue<int[]> queue = new LinkedList<>();

for (int i = 0; i < m; i++) {

for (int j = 0; j < n; j++) {

if (indegree[i][j] == 0) {

queue.add(new int[]{i, j});

}

}

}

int length = 0;

while (!queue.isEmpty()) {

// level by level

int size = queue.size();

for (int i = 0; i < size; i++) {

int[] cur = queue.poll();

int x = cur[0];

int y = cur[1];

for (int[] dir: dirs) {

int nx = x + dir[0];

int ny = y + dir[1];

// note that indegree[x][y] is zero

if (matrix[x][y] < matrix[nx][ny]) {

indegree[nx][ny]--;

if (indegree[nx][ny] == 0) {

queue.offer(new int[]{nx, ny});

}

}

}

}

}

length++;

}

return length;

}

The above efficient approach to solve the problem using topological sorting on a **Directed Acyclic Graph (DAG)**. This approach is based on the idea of treating the matrix as a graph and finding the longest path in the DAG. It is a clever optimization and works well for this problem.

Let’s break down the solution and analyze its correctness and efficiency:

**Key Idea:**

Graph Representation:

Each cell in the matrix is treated as a node.

A directed edge is drawn from node (i, j) to node (x, y) if:

(x, y) is adjacent to (i, j) (up, down, left, or right).

The value of (x, y) is greater than the value of (i, j).

DAG Property:

**Since the edges are directed from smaller values to larger values, the graph is acyclic (no cycles can exist).**

Longest Path in DAG:

**The problem reduces to finding the longest path in this DAG.**

**This can be efficiently solved using topological sorting and BFS (Kahn's algorithm).**

**Explanation of the Code:**

Indegree Calculation:

The indegree array is used to count the number of incoming edges to each node.

For each cell (i, j), we check its neighbors. If a neighbor (nx, ny) has a larger value, we increment indegree[nx][ny] because there is an edge from (i, j) to (nx, ny).

Topological Sorting with BFS:

Nodes with indegree == 0 are added to the queue. These are the starting points of the DAG (nodes with no incoming edges).

We process the queue level by level. For each node, we decrement the indegree of its neighbors and add them to the queue if their indegree becomes 0.

The number of levels processed in the BFS corresponds to the length of the longest path.

Result:

The length variable keeps track of the number of levels processed, which is the length of the longest increasing path.

**Time Complexity:**

Indegree Calculation: O(m \* n), where m is the number of rows and n is the number of columns. Each cell is processed once, and for each cell, we check its 4 neighbors.

Topological Sorting (BFS): O(m \* n), as each node is processed once, and each edge is processed once.

Overall Time Complexity: O(m \* n).

**Space Complexity:**

Indegree Array: O(m \* n) to store the indegree of each node.

Queue: O(m \* n) in the worst case (when all nodes are in the queue at some point).

Overall Space Complexity: O(m \* n).

**Why is this a Good Idea?**

Efficiency:

The time complexity is O(m \* n), which is optimal for this problem.

It avoids the overhead of recursion and memoization used in the DFS approach.

**Intuitive Graph Representation:**

**The problem is naturally modeled as a graph, and topological sorting is a standard technique for finding the longest path in a DAG.**

Level-by-Level Processing:

The BFS processes nodes level by level, making it easy to count the length of the longest path.

**Comparison with DFS + Memoization:**

DFS + Memoization:

Also has a time complexity of O(m \* n).

Uses recursion, which can lead to stack overflow for very large matrices.

Easier to implement for those familiar with DFS.

Topological Sorting (BFS):

Also has a time complexity of O(m \* n).

Avoids recursion and is more suitable for very large matrices.

Requires understanding of graph theory and topological sorting.

**Final Thoughts:**

The topological sorting approach is a great alternative to the DFS + memoization solution. It is efficient, intuitive, and avoids recursion. If you are comfortable with graph algorithms, this is a clean and elegant solution.

Here’s the code you provided, formatted and slightly commented for clarity:

class Solution {

    public int longestIncreasingPath(int[][] matrix) {

        int m = matrix.length;

        int n = matrix[0].length;

        int[][] dirs = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};

        // Generate the indegree array

        int[][] indegree = new int[m][n];

        for (int i = 0; i < m; i++) {

            for (int j = 0; j < n; j++) {

                for (int[] dir : dirs) {

                    int nx = i + dir[0];

                    int ny = j + dir[1];

                        indegree[nx][ny]++;

                    }

                }

            }

        }

        // Initialize the queue with nodes having indegree == 0

        Queue<int[]> queue = new LinkedList<>();

        for (int i = 0; i < m; i++) {

            for (int j = 0; j < n; j++) {

                if (indegree[i][j] == 0) {

                    queue.add(new int[]{i, j});

                }

            }

        }

        // Perform BFS to count the levels (longest path)

        int length = 0;

        while (!queue.isEmpty()) {

            int size = queue.size();

            for (int i = 0; i < size; i++) {

                int[] cur = queue.poll();

                int x = cur[0];

                int y = cur[1];

                for (int[] dir : dirs) {

                    int nx = x + dir[0];

                    int ny = y + dir[1];

                        indegree[nx][ny]--;

                        if (indegree[nx][ny] == 0) {

                            queue.offer(new int[]{nx, ny});

                        }

                    }

                }

            }

            length++;

        }

        return length;

    }

}

This solution is correct and efficient, and it’s a great addition to your toolkit for solving similar problems!

**Refer to**

[L2328.Number of Increasing Paths in a Grid](note://WEB1ab466bd849e83e490085cec0d601f1d)

[Topological Sort DFS, BFS and DAG](note://5DF27CD4E3CA4F61BCB248EF65D2E17F)