329. Longest Increasing Path in a Matrix Breadth-First Search

Graph

# **Problem Description** This LeetCode problem asks us to find the length of the longest increasing path in a given 2D matrix of integers. The path can be

Depth-First Search

constructed by moving from each cell in one of four directions - left, right, up, or down - but not diagonally. The path must consist of a sequence of numbers where each number is greater than the one before it. As a path cannot wrap around the border of the matrix, once you reach the edge, you cannot go any further in that direction. The goal is to determine the maximum length of such an increasing path. Intuition

Topological Sort

Memoization

Array

Dynamic Programming

Matrix

## To arrive at the solution, we can use depth-first search (DFS) to explore each possible path from each cell in the matrix. The main idea is to traverse the matrix using DFS to find the longest upward path starting from any given cell. At each cell (i, j), we look up,

Hard

Leetcode Link

represents a potential path, and we use recursion to explore all paths. We need to keep track of the length of the path we've found, and update the maximum length when we discover a longer path. To improve the efficiency of our search, we can use memoization to store the results of the subproblems that have already been computed. In this code, dfs(i, j) represents the length of the longest path starting from cell (i, j). The @cache decorator from Python's functools module ensures that once we compute dfs(i, j) for a particular cell (i, j), we do not recompute it again - the

down, left, and right for any adjacent cells that have a higher value than the current cell, and continue on from there. Each movement

We iterate through every cell in the matrix, invoking dfs(i, j) to get the length of the longest increasing path starting from that cell, keeping track of the maximum value as we go. By the end of this process, we will have explored all the possible paths and found the longest possible path length in the matrix, which will be our final answer.

**Solution Approach** The solution uses depth-first search (DFS) along with memoization to efficiently find the longest increasing path. Here is a step-bystep explanation of how the solution approach is implemented:

• We begin by defining a dfs function that takes the coordinates (1, 1) of a cell as arguments. This function will return the length

## of the longest increasing path starting from this cell. To remember the length of paths we've already found, we decorate this function with the @cache decorator, enabling memoization.

ans and the result of dfs(x, y).

result is stored and reused.

 Within the dfs function, we initialize a variable ans to 0 to keep track of the longest path found starting from the given cell. • We create a loop that iterates over the adjacent cells to the current cell (i, j). This loop uses pairwise to iterate through the

- directional offsets (-1, 0, 1, 0, -1) to access the left, right, up, and down neighboring cells by adding these offsets to the current cell's indices. • For each neighboring cell (x, y), we check if the cell is within the matrix bounds and if its value is greater than the current cell's
- value, which would make it a candidate for the increasing path. • If the adjacent cell (x, y) meets the conditions, we call dfs(x, y) and update ans with the maximum value between the current
- After exploring all possible paths from the current cell, we return ans + 1, with the + 1 accounting for the length of the path including the current cell.

Outside of the dfs function, we determine the dimensions of the matrix m and n. We then iterate through every cell in the matrix,

calling dfs(i, j) for each and finding the maximum value across all cells. This maximum value is the length of the longest

increasing path in the matrix. We finally return this maximum value as the solution to the problem.

The application of DFS, along with memoization via the @cache decorator, allows us to efficiently avoid recomputation of paths that

have already been computed. This makes the algorithm significantly faster, especially for larger matrices, as we are able to reuse

Let's use a small example to illustrate the solution approach. Suppose we have the following 2D matrix:

- previously computed results for the subproblems. Example Walkthrough

We want to find the length of the longest increasing path in this matrix.

Following the solution approach, we apply DFS and memoization as outlined:

1. We define our dfs function with memoization to keep track of the longest path starting from each cell.

4. At (0, 1), with the value 4, the only valid increasing step is to move down to (1, 1) with the value 5. We call dfs(1, 1).

5. At (1, 1), with the value 5, there are no further adjacent cells with larger values, so its longest path is just itself, with a length of

6. Returning to (0, 1) with value 4, we add 1 for the current cell to the maximum path length we can get by moving from it, which

7. Now we go back to the starting cell (0, 0). With the memoized results, we know the longest path from (0, 1) is of length 2. So

2. We start with the cell at (0, 0) which has the value 3. We look at its neighbors (0, 1) with value 4. 3. Since 4 is greater than 3, it's a valid step in the increasing path. We call dfs(0, 1).

1. The value is memoized.

length is already memoized.

Matrix:

is 1 for the (1, 1) cell. The total is 2, which is memoized for (0, 1).

as we avoid recalculating the longest path from each cell more than once.

def longestIncreasingPath(self, matrix: List[List[int]]) -> int:

# Define direction vectors for traversing up, down, left, and right

# Use memoization to cache the results of previous computations

new\_row, new\_col = row + d\_row, col + d\_col

max\_path = max(max\_path, dfs(i, j))

int nextCol = col + directions[k + 1];

return ++dp[row][col];

# Check if the new position is within bounds and increases in value

# Recursively find the longest increasing path from the new position

max\_path\_length = max(max\_path\_length, 1 + dfs(new\_row, new\_col))

# Define the dimensions of the matrix

@lru\_cache(maxsize=None)

rows, cols = len(matrix), len(matrix[0])

for d row, d col in directions:

directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

for (0, 0), the length is 2 plus the current cell: 3.

8. Next, we check the cell (1, 0).

9. The cell (1, 0) has a value of 2 and only one neighbor (1, 1) with a greater value. We call dfs(1, 1), but this time, the path

- 10. For (1, 0), we take the memoized path length of (1, 1), which is 1, and add 1 for the current cell, giving us a total path length of 2.
- 1) does not lead to any other cell, so its path length is 1. 12. We iterate through each cell in the matrix and choose the maximum length from the results of the dfs calls. The final result for

Applying the steps from the solution to this small example matrix demonstrates the increase in efficiency gained from memoization,

the longest increasing path in this example is 3, which starts from cell (0, 0) and goes through (0, 1) to (1, 1).

11. Therefore, the possible path lengths starting from each cell are 3 from (0, 0) and 2 from both (0, 1) and (1, 0). The cell at (1,

from typing import List from functools import lru\_cache class Solution:

def dfs(row: int, col: int) -> int: 15 # Start with the longest path of 1 that includes the current cell  $max_path_length = 1$ 18 # Explore all possible directions from the current cell 19

if 0 <= new\_row < rows and 0 <= new\_col < cols and matrix[new\_row][new\_col] > matrix[row][col]:

```
26
27
28
                return max_path_length
29
           # Iterate over all matrix cells to find the starting point of the longest increasing path
30
```

38 # Example usage

max\_path = 0

return max\_path

for i in range(rows):

for j in range(cols):

Python Solution

9

10

11

12

13

14

20

21

22

23

24

25

31

32

33

34

35

36

37

```
39 # solution = Solution()
 40 # print(solution.longestIncreasingPath([[9,9,4],[6,6,8],[2,1,1]])) # Output: 4
 41
Java Solution
    class Solution {
         private int numRows;
         private int numCols;
         private int[][] grid; // Renamed "matrix" to "grid" for clarity
         private int[][] dp; // Renamed "f" to "dp" for denoting the "dynamic programming" cache
  6
         public int longestIncreasingPath(int[][] matrix) {
             numRows = matrix.length;
             numCols = matrix[0].length;
             dp = new int[numRows][numCols];
 10
 11
             this.grid = matrix;
 12
             int longestPath = 0;
 13
 14
             // Iterate over each cell in the matrix
 15
             for (int i = 0; i < numRows; ++i) {
 16
                 for (int j = 0; j < numCols; ++j) {</pre>
 17
                     // Update the longest path found so far
 18
                     longestPath = Math.max(longestPath, dfs(i, j));
 19
 20
 21
             return longestPath;
 22
 23
 24
 25
         private int dfs(int row, int col) {
 26
             // If the current cell already contains a computed length, return it
 27
             if (dp[row][col] != 0) {
                 return dp[row][col];
 28
 29
 30
 31
             // Direction vectors for the 4 adjacent cells (up, right, down, left)
 32
             int[] directions = \{-1, 0, 1, 0, -1\};
 33
 34
             // Explore all adjacent cells
 35
             for (int k = 0; k < 4; k++) {
 36
                 int nextRow = row + directions[k];
```

// Check if the next cell is within bounds and has a larger value than the current cell

// Update the length of the longest increasing path from the current cell

dp[row][col] = Math.max(dp[row][col], dfs(nextRow, nextCol));

// Increment the stored length by 1 to include the current cell and return it

int cols = matrix[0].size(); // Number of columns in the matrix

vector<int> directions =  $\{-1, 0, 1, 0, -1\}$ ;

return cache[row][col];

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

// Check all four possible directions

if (cache[row][col]) {

int longestPath = 0; // Initialize the length of the longest path

// Depth-first search function to find the longest increasing path

function<int(int, int)> dfs = [&](int row, int col) -> int {

vector<vector<int>> cache(rows, vector<int>(cols, 0)); // Memoization cache

// If we've already computed this position, return the cached value

int newRow = row + directions[k]; // Calculate new row position

// Directions array representing the four possible movements (up, right, down, left)

if (nextRow >= 0 && nextRow < numRows && nextCol >= 0 && nextCol < numCols && grid[nextRow][nextCol] > grid[row][col])

### 1 class Solution { public: int longestIncreasingPath(vector<vector<int>>& matrix) { int rows = matrix.size(); // Number of rows in the matrix

C++ Solution

37

38

39

40

41

42

43

44

45

46

47

48

49 }

9

10

11

12

13

14

15

16

17

18

19

24

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

60

62

63

64

65

67

68

69

70

71

72

73

74

75

82

83

84

86

85 }

}

\*/

if (cache[row][column] > 0) {

let maxLength = 1;

if (

return cache[row][column];

for (const [dx, dy] of directions) {

const newColumn = column + dy;

const newRow = row + dx;

cache[row][column] = maxLength;

\* @param matrix - The input matrix of numbers.

function longestIncreasingPath(matrix: Matrix): number {

// Handle an empty matrix case by returning 0 early.

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

return maxLength;

return 0;

const rows = matrix.length;

let globalMaxLength = 0;

const columns = matrix[0].length;

// Iterate over each cell in the matrix.

// Return the global maximum length found.

// Initialize the maximum length from the current cell to 1 (the cell itself).

// Recursively compute the longest path from the neighboring cell.

// Update the maximum length if the computed path is longer.

// Check if the new cell is within bounds and has a value greater than the current cell.

// Explore all possible directions from the current cell.

newRow >= ∅ && newRow < matrix.length &&

maxLength = Math.max(maxLength, length);

// Cache the computed maximum length for the current cell.

\* Computes the length of the longest increasing path in a matrix.

\* @returns The length of the longest increasing path in the matrix.

// Initialize rows and columns counts, and the cache matrix.

// The overall maximum length of the longest increasing path.

cache = Array.from({length: rows}, () => Array(columns).fill(0));

newColumn >= 0 && newColumn < matrix[0].length &&

const length = 1 + dfs(newRow, newColumn, matrix);

matrix[newRow][newColumn] > matrix[row][column]

```
int newCol = col + directions[k + 1]; // Calculate new column position
 20
 21
                    // If the new position is within bounds and is greater than the current cell
 22
                     if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && matrix[newRow][newCol] > matrix[row][col]) {
 23
                        // Recursively explore the new position and update the cache
                         cache[row][col] = max(cache[row][col], dfs(newRow, newCol));
 24
 25
 26
 27
                // After exploring all directions, increment the path length by 1 and return
 28
                 return ++cache[row][col];
            };
 29
 30
 31
            // Iterate through each cell in the matrix
 32
             for (int i = 0; i < rows; ++i) {
 33
                 for (int j = 0; j < cols; ++j) {
                    // Compute the longest increasing path starting from (i, j)
 35
                     longestPath = max(longestPath, dfs(i, j));
 36
 37
 38
            // Return the length of the longest path after processing all cells
 39
             return longestPath;
 40
 41 };
 42
Typescript Solution
  1 // Type definition for the matrix of numbers.
  2 type Matrix = number[][];
    // Cache for storing the length of the longest increasing path from a particular cell.
    let cache: Matrix;
    // Directions array representing the movement offsets for the four cardinal directions.
    const directions: [number, number][] = [
         [-1, 0], // up
         [0, 1], // right
 10
         [1, 0], // down
 11
         [0, -1], // left
 13 ];
 14
 15 /**
     * Performs a depth-first search to find the longest increasing path from a given cell.
     * @param row - The row index of the current cell.
     * @param column - The column index of the current cell.
     * @param matrix - The input matrix of numbers.
     * @returns The length of the longest increasing path from the current cell.
 21 */
    function dfs(row: number, column: number, matrix: Matrix): number {
        // If we already computed the longest path from this cell, return the cached value.
 23
```

### 76 for (let i = 0; i < rows; ++i) { 77 for (let j = 0; j < columns; ++j) { // Use DFS to find the longest path from the current cell and update the global maximum. 78 79 globalMaxLength = Math.max(globalMaxLength, dfs(i, j, matrix)); 80 81

return globalMaxLength;

# Time and Space Complexity **Time Complexity**

The time complexity of this algorithm is 0(m \* n) where m is the number of rows and n is the number of columns in the input matrix. This is because in the worst case scenario, we perform a depth-first search (DFS) starting from each cell in the matrix. The memoization using @cache ensures that each cell (i, j) is only processed once for its longest increasing path, and the DFS from that cell terminates once it has explored all increasing paths from that cell. Hence, each cell's DFS contributes 0(1) amortized time, and since we have m \* n cells, the overall time is O(m \* n). Space Complexity

The algorithm has a space complexity of 0(m \* n) as well, due to the memoization cache that potentially stores the results for each cell in the matrix. The space required is for the recursive call stack and the cache storage. In the worst case, the recursive call stack could go as deep as m \* n if there is an increasing path through every cell. Thus, with the cache and call stack combined, the space complexity is 0(m \* n).