JavaScript Longest Increasing Path in a Matrix

Challenge

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.

> **Important**
>
> You may not move diagonally or move outside the boundary (ex. wrap-around is not allowed).

1st Example

```
Input: matrix = [[9,9,4],[6,6,8],[2,1,1]]
Output: 4
Explanation: The longest increasing path is [1, 2, 6, 9].
```

2nd Example

```
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.
```

3rd Example

```
Input: matrix = [[1]]
Output: 1
```

Constraints

- `m == matrix.length`
- `n == matrix[i].length`
- `1 <= m, n <= 200`
- `0 <= matrix[i][j] <= 2³¹ - 1`

Solution

```
const longestIncreasingPath = (matrix) => {
    const row   = matrix.length,
          cols  = matrix[0].length,
          cache = new Array(row);

    for (let i = 0; i < cache.length; i++) {
        cache[i] = new Array(cols);
    }

    const dfs = (row, cols, min = -Infinity) => {
        if (row < 0 ||
            cols < 0 ||
            row >= matrix.length ||
            cols >= matrix[row].length) {
            return 0;
        }
```

Solution continues on next page...

```
        if (matrix[row][cols] <= min) {
            return 0;
        }

        if (cache[row][cols] != null) {
            return cache[row][cols];
        }

        let top    = dfs(row + 1, cols, matrix[row][cols]),
            bottom = dfs(row - 1, cols, matrix[row][cols]),
            right  = dfs(row, cols + 1, matrix[row][cols]),
            left   = dfs(row, cols - 1, matrix[row][cols]),
            count  = 1 + Math.max(top, bottom, left, right);

        cache[row][cols] = count;

        return count;
    };

    let longestPath = 0;

    for (let i = 0; i < row; i++) {
        for (let j = 0; j < cols; j++) {
            longestPath = Math.max(longestPath, dfs(i,j));
        }
    }

    return longestPath;
};
```

Explanation

I've created a function called `longestIncreasingPath` that calculates the length of the longest increasing path in a matrix.

The function begins by initializing variables for the number of rows

(`row`) and columns (`cols`) in the matrix, as well as an empty 2D array called `cache` to store computed results.

Next, the function defines an inner recursive function called `dfs` that takes parameters for the current row, column, and a minimum value. This function handles the depth-first search to find the longest increasing path.

Inside the `dfs` function, several conditional statements are used to handle base cases and edge cases. If the current row or column is out of bounds, the function returns `0`. If the value at the current position is not increasing (less than or equal to the minimum value), the function returns `0`. If the value has already been computed and stored in the `cache` array, the function retrieves the result from the cache.

If none of the base cases are met, the function proceeds with recursive calls to `dfs` for the positions below, above, right, and left of the current position. The minimum value is updated accordingly.

The function calculates the count as `1` plus the maximum length among the four recursive calls. This represents the length of the longest increasing path starting from the current position.

The length of the longest increasing path for the current position is stored in the `cache` array.

The `dfs` function is called for each position in the matrix using nested loops. The length of the longest increasing path found so far is updated if a longer path is found.

Finally, the function returns the length of the longest increasing path as the output.

In summary, the `longestIncreasingPath` function uses a recursive depth-first search approach to find the length of the longest increasing path in a matrix. It leverages memoization by storing computed results in the `cache` array to avoid redundant calculations.

Author: Trevor Morin