Longest Increasing Path in a Matrix

Given an integer matrix, find the length of the longest increasing path.

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

Example 1:

```
nums = [
  [9,9,4],
  [6,6,8],
  [2,1,1]
]
```

Return `4`
The longest increasing path is `[1, 2, 6, 9]`.

Example 2:

```
nums = [
  [3,4,5],
  [3,2,6],
  [2,2,1]
]
```

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

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

To get max length of increasing sequences:

1. Do `DFS` from every cell
2. Compare every 4 direction and skip cells that are out of boundary or smaller
3. Get matrix `max` from every cell's `max`
4. Use `matrix[x][y] <= matrix[i][j]` so we don't need a `visited[m][n]` array
5. The key is to `cache` the distance because it's highly possible to revisit a cell

Hope it helps!

```
public static final int[][] dirs = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

public int longestIncreasingPath(int[][] matrix) {
    if(matrix.length == 0) return 0;
    int m = matrix.length, n = matrix[0].length;
    int[][] cache = new int[m][n];
    int max = 1;
    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) {
            int len = dfs(matrix, i, j, m, n, cache);
            max = Math.max(max, len);
        }
    }
    return max;
}

public int dfs(int[][] matrix, int i, int j, int m, int n, int[][] cache) {
    if(cache[i][j] != 0) return cache[i][j];
    int max = 1;
    for(int[] dir: dirs) {
        int x = i + dir[0], y = j + dir[1];
        if(x < 0 || x >= m || y < 0 || y >= n || matrix[x][y] <= matrix[i][j]) co
ntinue;
        int len = 1 + dfs(matrix, x, y, m, n, cache);
        max = Math.max(max, len);
    }
    cache[i][j] = max;
    return max;
}
```

written by yavinci original link here

Solution 2

The idea is simple and intuitive:
1. For each cell, try it's left, right, up and down for smaller number.
2. If it's smaller, means we are on the right track and we should keep going. If larger, stop and return.
3. Treat each cell as a start cell. Calculate and memorize the longest distance for this cell, so we don't need to calculate it again in the future.

Questions and advices are welcome.

```
public class Solution {
    public int longestIncreasingPath(int[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return 0;
        }
        int[][] cache = new int[matrix.length][matrix[0].length];
        int max = 0;
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[0].length; j++) {
                int length = findSmallAround(i, j, matrix, cache, Integer.MAX_VAL
UE);
                max = Math.max(length, max);
            }
        }
        return max;
    }
    private int findSmallAround(int i, int j, int[][] matrix, int[][] cache, int
pre) {
        // if out of bond OR current cell value larger than previous cell value.
        if (i < 0 || i >= matrix.length || j < 0 || j >= matrix[0].length || matr
ix[i][j] >= pre) {
            return 0;
        }
        // if calculated before, no need to do it again
        if (cache[i][j] > 0) {
            return cache[i][j];
        } else {
            int cur = matrix[i][j];
            int tempMax = 0;
            tempMax = Math.max(findSmallAround(i - 1, j, matrix, cache, cur), tem
pMax);
            tempMax = Math.max(findSmallAround(i + 1, j, matrix, cache, cur), tem
pMax);
            tempMax = Math.max(findSmallAround(i, j - 1, matrix, cache, cur), tem
pMax);
            tempMax = Math.max(findSmallAround(i, j + 1, matrix, cache, cur), tem
pMax);
            cache[i][j] = ++tempMax;
            return tempMax;
        }
    }
}
```

written by EdickCoding original link here

Solution 3

We can find longest decreasing path instead, the result will be the same. Use `dp` to record previous results and choose the max `dp` value of smaller neighbors.

```
def longestIncreasingPath(self, matrix):
    def dfs(i, j):
        if not dp[i][j]:
            val = matrix[i][j]
            dp[i][j] = 1 + max(
                dfs(i - 1, j) if i and val > matrix[i - 1][j] else 0,
                dfs(i + 1, j) if i < M - 1 and val > matrix[i + 1][j] else 0,
                dfs(i, j - 1) if j and val > matrix[i][j - 1] else 0,
                dfs(i, j + 1) if j < N - 1 and val > matrix[i][j + 1] else 0)
        return dp[i][j]

    if not matrix or not matrix[0]: return 0
    M, N = len(matrix), len(matrix[0])
    dp = [[0] * N for i in range(M)]
    return max(dfs(x, y) for x in range(M) for y in range(N))
```

written by kitt original link here