1631. Path With Minimum Effort

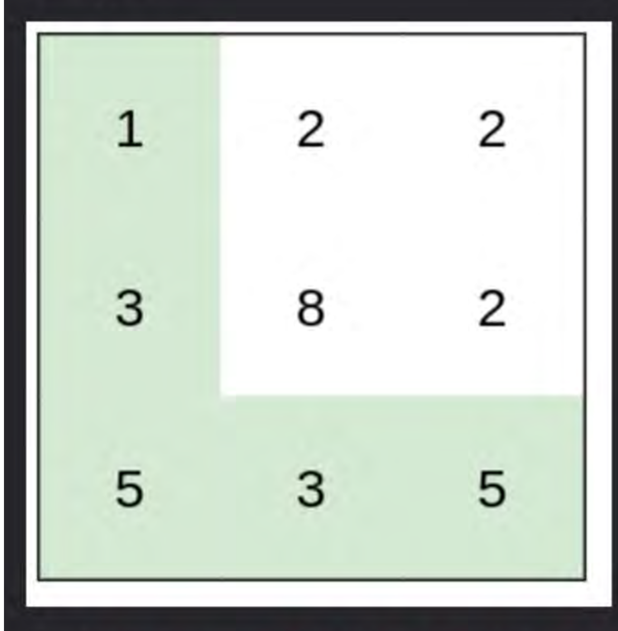
Leetcode Link

You are a hiker preparing for an upcoming hike. You are given `heights`, a 2D array of size `rows x columns`, where `heights[row][col]` represents the height of cell `(row, col)`. You are situated in the top-left cell, `(0, 0)`, and you hope to travel to the bottom-right cell, `(rows-1, columns-1)` (i.e., **0-indexed**). You can move **up**, **down**, **left**, or **right**, and you wish to find a route that requires the minimum **effort**.

A route's **effort** is the **maximum absolute difference** in heights between two consecutive cells of the route.

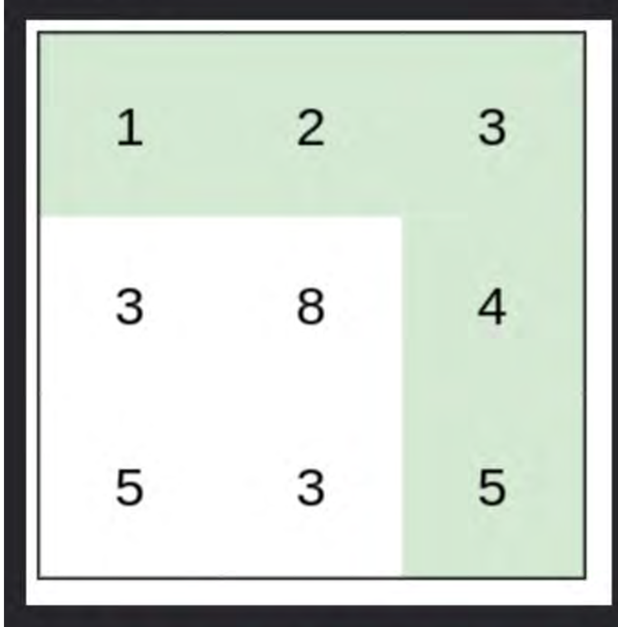
Return the minimum **effort** required to travel from the top-left cell to the bottom-right cell.

Example 1:



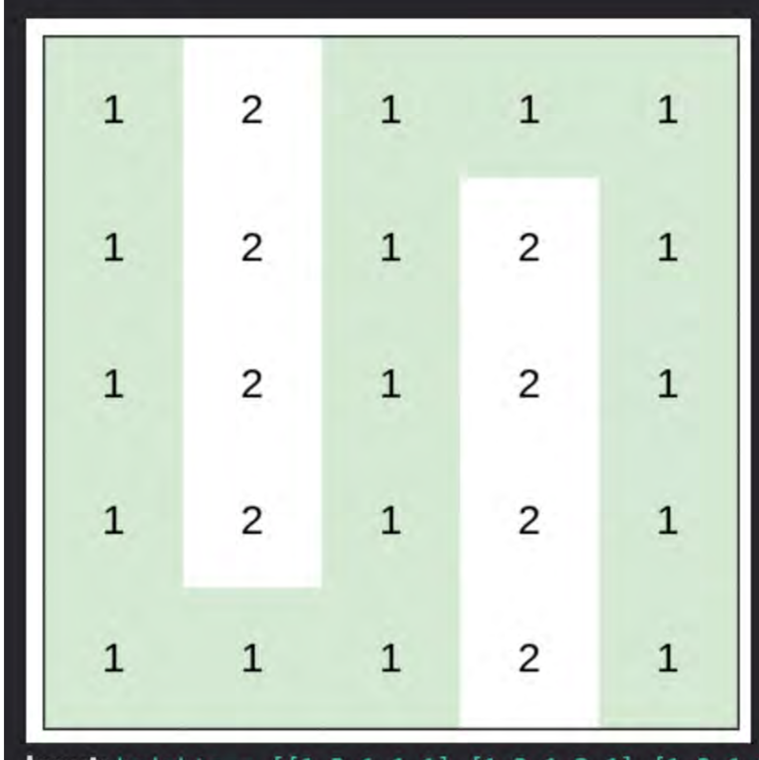
Input: `heights = [[1,2,2],[3,8,2],[5,3,5]]`
Output: 2
Explanation: The route of `[1,3,5,3,5]` has a maximum absolute difference of 2 in consecutive cells. This is better than the route of `[1,2,2,2,5]`, where the maximum absolute difference is 3.

Example 2:



Input: `heights = [[1,2,3],[3,8,4],[5,3,5]]`
Output: 1
Explanation: The route of `[1,2,3,4,5]` has a maximum absolute difference of 1 in consecutive cells, which is better than route `[1,3,5,3,5]`.

Example 3:



Input: `heights = [[1,2,1,1,1],[1,2,1,2,1],[1,2,1,2,1],[1,2,1,2,1],[1,1,1,2,1]]`
Output: 0
Explanation: This route does not require any effort.

Constraints:

`rows == heights.length`
`columns == heights[i].length`
$1 \leq \text{rows}, \text{columns} \leq 100$
$1 \leq$ `heights[i][j]` $\leq 10^6$

Solution

Brute Force

Our most simple brute force for this problem would be to try all different routes that start from the top-left cell and end in the bottom-right cell. We'll then find the efforts for all these routes and return the smallest.

Full Solution

Instead of thinking of finding the efforts of all possible routes, we should think about finding routes that have some specific effort. Given some effort a, how do we check if there exists a route that has an effort smaller or equal to a?

Let's first denote the distance between two adjacent cells as the absolute difference between their heights.

A route will have an effort smaller or equal to a if it travels from the top-left cell to the bottom-right cell and all adjacent cells in the route have a distance that doesn't exceed a. To check if such routes exist, we can check if there exists a path from the top-left cell to the bottom-right cell such that we never travel between cells with a distance that exceeds a. We can accomplish this with a BFS/flood fill algorithm.

We'll denote an effort a as **good** if there exists a route with an effort smaller or equal to a.

The minimum effort a that's **good** is the final answer that we return. To find the minimum effort, we can implement a binary search. Why can we binary search this value? Let's say our minimum **good** effort is b. Our binary search condition is satisfied since all values **strictly** less than b are **not good** and all values greater or equal to b are **good**.

Every binary search iteration, we can check whether or not some effort is **good** by running the BFS/flood fill algorithm mentioned above.

Time Complexity

Let's denote R as number of rows, C as number of colmns, and M as maximum height in `heights`.

Since BFS/flood fill will run in $\mathcal{O}(RC)$ and we have $\mathcal{O}(\log M)$ binary search iterations, our final time complexity will be $\mathcal{O}(RC \log M)$.

Time Complexity: $\mathcal{O}(RC \log M)$

C++ Solution

```cpp
class Solution {
    const vector<int> deltaRow = {-1, 0, 1, 0};
    const vector<int> deltaCol = {0, 1, 0, -1};
    bool isValidEffort(vector<vector<int>>& heights, int mid) {
        int rows = heights.size();
        int columns = heights[0].size();                        // dimensions for heights
        vector<vector<bool>> vis(rows, vector<bool>(columns));  // keeps track of whether or not we visited a node
        queue<int> qRow;
        queue<int> qCol;
        qRow.push(0);
        qCol.push(0);  // BFS starts in top-left cell
        //We can also use queue<pair<int,int>> to store both the row & col in one queue
        vis[0][0] = true;
        while (!qRow.empty()) {
            int curRow = qRow.front();
            qRow.pop();
            int curCol = qCol.front();
            qCol.pop();
            for (int dir = 0; dir < 4; dir++) {
                int newRow = curRow + deltaRow[dir];
                int newCol = curCol + deltaCol[dir];
                if (newRow < 0 || newRow >= rows || newCol < 0 || newCol >= columns) {  // check if cell is in boundary
                    continue;
                }
                if (vis[newRow][newCol]) {  // check if cell has been visited
                    continue;
                }
                if (abs(heights[newRow][newCol] - heights[curRow][curCol]) > mid) {  // check if distance exceeds limit
                    continue;
                }
                vis[newRow][newCol] = true;
                qRow.push(newRow);
                qCol.push(newCol);
                // process next node
            }
        }
        return vis[rows - 1][columns - 1];
    }

   public:
    int minimumEffortPath(vector<vector<int>>& heights) {
        int low = -1;         // every effort less or equal to low will never be good
        int high = (int)1e6;  // every effort greater or equal to high will always be good
        int mid = (low + high) / 2;
        while (low + 1 < high) {
            if (isValidEffort(heights, mid)) {
                high = mid;
            } else {
                low = mid;
            }
            mid = (low + high) / 2;
        }
        return high;
    }
};
```

Java Solution

```java
class Solution {
    final int[] deltaRow = {-1, 0, 1, 0};
    final int[] deltaCol = {0, 1, 0, -1};
    private boolean isValidEffort(int[][] heights, int mid){
        int rows = heights.length;
        int columns = heights[0].length; // dimensions for heights
        boolean[][] vis = new boolean[rows][columns]; // keeps track of whether or not we visited a node
        Queue<Integer> qRow = new LinkedList();
        Queue<Integer> qCol = new LinkedList();
        qRow.add(0);
        qCol.add(0); // BFS starts in top-left cell
        vis[0][0] = true;
        while (!qRow.isEmpty()) {
            int curRow = qRow.poll();
            int curCol = qCol.poll();
            for (int dir = 0; dir < 4; dir++) {
                int newRow = curRow + deltaRow[dir];
                int newCol = curCol + deltaCol[dir];
                if (newRow < 0 || newRow >= rows || newCol < 0
                    || newCol >= columns) { // check if cell is in boundary
                    continue;
                }
                if (vis[newRow][newCol]) { // check if cell has been visited
                    continue;
                }
                if (Math.abs(heights[newRow][newCol] - heights[curRow][curCol]) > mid){
                    // check if distance exceeds limit
                    continue;
                }
                vis[newRow][newCol] = true;
                qRow.add(newRow);
                qCol.add(newCol);
                // process next node
            }
        }
        return vis[rows-1][columns-1];
    }
    public int minimumEffortPath(int[][] heights) {
        int rows = heights.length;
        int columns = heights[0].length; // dimensions for heights
        int low = -1; // every effort less or equal to low will never be good
        int high = (int) 1e6; // every effort greater or equal to high will always be good
        int mid = (low + high) / 2;
        while (low + 1 < high) {
            if (isValidEffort(heights,mid)) {
                high = mid;
            } else {
                low = mid;
            }
            mid = (low + high) / 2;
        }
        return high;
    }
}
```

Python Solution

```python
import collections
class Solution:
    def minimumEffortPath(self, heights: List[List[int]]) -> int:
        rows = len(heights)
        columns = len(heights[0])  # dimensions for heights
        low = -1  # every effort less or equal to low will never be good
        high = 10 ** 6  # every effort greater or equal to high will always be good
        mid = (low + high) // 2
        def isValidEffort(heights, mid):
            vis = [[False] * columns for a in range(rows)]
            # keeps track of whether or not we visited a node
            qRow = collections.deque([0])
            qCol = collections.deque([0])  # BFS starts in top-left cell
            vis[0][0] = True
            while qRow:
                curRow = qRow.popleft()
                curCol = qCol.popleft()
                for [deltaRow, deltaCol] in [(-1, 0), (0, 1), (1, 0), (0, -1)]:
                    newRow = curRow + deltaRow
                    newCol = curCol + deltaCol
                    if (newRow < 0 or newRow >= rows or newCol < 0 or newCol >= columns):
                        # check if cell is in boundary
                        continue
                    if vis[newRow][newCol] == True:  # check if cell has been visited
                        continue
                    if (abs(heights[newRow][newCol] - heights[curRow][curCol]) > mid):
                        # check if distance exceeds limit
                        continue
                    vis[newRow][newCol] = True
                    qRow.append(newRow)
                    qCol.append(newCol)
                    # process next node
            return vis[rows-1][columns-1]
        while low + 1 < high:
            if isValidEffort(heights,mid):
                high = mid
            else:
                low = mid
            mid = (low + high) // 2
        return high

```

Got a question? Ask the Teaching Assistant anything you don't understand.