778. Swim in Rising Water

You are given an `n x n` integer matrix `grid` where each value `grid[i][j]` represents the elevation at that point `(i, j)`.

The rain starts to fall. At time `t`, the depth of the water everywhere is `t`. You can swim from a square to another 4-directionally adjacent square if and only if the elevation of both squares individually are at most `t`. You can swim infinite distances in zero time. Of course, you must stay within the boundaries of the grid during your swim.

Return the least time until you can reach the bottom right square `(n - 1, n - 1)` if you start at the top left square `(0, 0)`.

Example 1:

0	2
1	3

Input: `grid = [[0,2],[1,3]]`
Output: 3
Explanation:
At time 0, you are in grid location `(0, 0)`. You cannot go anywhere else because 4-directionally adjacent neighbors have a higher elevation than `t = 0`. You cannot reach point `(1, 1)` until time 3. When the depth of water is 3, we can swim anywhere inside the grid.

Example 2:

0	1	2	3	4
24	23	22	21	5
12	13	14	15	16
11	17	18	19	20
10	9	8	7	6

Input: `grid = [[0,1,2,3,4],[24,23,22,21,5],[12,13,14,15,16],[11,17,18,19,20],[10,9,8,7,6]]`
Output: 16
Explanation: The final route is shown.
We need to wait until time 16 so that `(0, 0)` and `(4, 4)` are connected.

Constraints:

- `n == grid.length`
- `n == grid[i].length`
- $1 \le n \le 50$
- $0 \le$ `grid[i][j]` $< n^2$
- Each value `grid[i][j]` is unique.

Solution

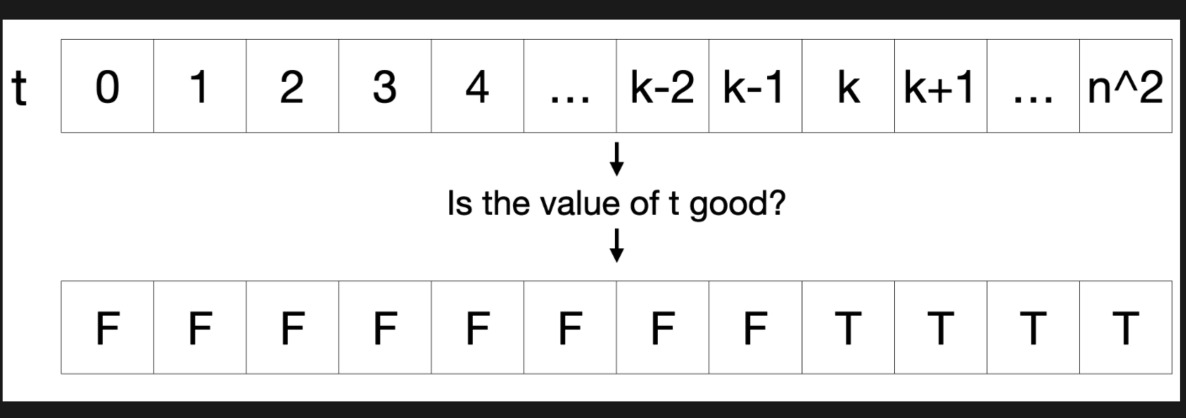
Brute Force

In simpler terms, this problem is asking us what's the minimum value of t such that you can travel from the top-left square to the bottom-right square by only travelling between adjacent cells that never exceed an elevation of t. To solve this problem, we can try all values of t from 0 to $n^2 - 1$. For each value of t, we can run a BFS/flood fill algorithm to check the connectivity between the top-left and bottom-right squares. It's important that in this algorithm, we only traverse squares with elevations that don't exceed t.

Since there are $\mathcal{O}(N^2)$ values of t to try and our BFS/flood fill algorithm runs in $\mathcal{O}(N^2)$, this gives us a time complexity of $\mathcal{O}(N^4)$.

Full Solution

Let's denote a value t as **good** if the bottom-right square is reachable from the top-left square only through squares with elevations not exceeding t. Let's denote the minimum **good** value t as k. We can observe that all values t such that $t < k$ are never **good** and that all values t such that $t \geq k$ are all **good**. What does this mean for us? It means we can binary search it since our binary search condition is satisfied.



Each binary search iteration, we'll use the same BFS/flood fill algorithm to check if a specific value of t is **good**.

Time Complexity

Note that $\log N^2 = 2\log N$ so $\mathcal{O}(\log N^2) = \mathcal{O}(\log N)$.

We have $\mathcal{O}(\log N)$ binary search iterations and each iteration uses a BFS/flood fill algorithm which runs in $\mathcal{O}(N^2)$. Thus, our final time complexity is $\mathcal{O}(N^2 \log N)$.

Time Complexity: $\mathcal{O}(N^2 \log N)$

Space Complexity

Our BFS/flood fill algorithm will take $\mathcal{O}(N^2)$ space so our space complexity is $\mathcal{O}(N^2)$.

Space Complexity: $\mathcal{O}(N^2)$

C++ Solution

```
class Solution {
    const vector<int> deltaRow = {-1, 0, 1, 0};
    const vector<int> deltaCol = {0, 1, 0, -1};
    bool isEndReachable(vector<vector<int>>& grid, int t) {
        if (grid[0][0] > t) {  // starting elevation can't exceed t
            return false;
        }
        int n = grid.size();
        vector<vector<bool>> vis(n, vector<bool>(n));
        queue<vector<int>> q;
        q.push({0, 0});  // starting cell
        vis[0][0] = true;
        while (!q.empty()) {
            vector<int> cur = q.front();
            q.pop();
            int curRow = cur[0];
            int curCol = cur[1];
            for (int i = 0; i < 4; i++) {
                int newRow = curRow + deltaRow[i];
                int newCol = curCol + deltaCol[i];
                if (newRow < 0 || newRow >= n || newCol < 0 || newCol >= n) {  // outside of boundary
                    continue;
                }
                if (vis[newRow][newCol]) {  // visited node before
                    continue;
                }
                if (grid[newRow][newCol] > t) {  // check if cell can be traversed
                    continue;
                }
                vis[newRow][newCol] = true;
                q.push({newRow, newCol});
            }
        }
        return vis[n - 1][n - 1];
    }

   public:
    int swimInWater(vector<vector<int>>& grid) {
        int n = grid.size();
        int low = -1;      // all values smaller or equal to low are not good
        int high = n * n;  // all values greater or equal to high are good
        int mid = (low + high) / 2;
        while (low + 1 < high) {
            if (isEndReachable(grid, mid)) {
                high = mid;
            } else {
                low = mid;
            }
            mid = (low + high) / 2;
        }
        return high;
    }
};
```

Java Solution

```
class Solution {
    static int[] deltaRow = {-1, 0, 1, 0};
    static int[] deltaCol = {0, 1, 0, -1};
    boolean isEndReachable(int[][] grid, int t) {
        if (grid[0][0] > t) { // starting elevation can't exceed t
            return false;
        }
        int n = grid.length;
        boolean[][] vis = new boolean[n][n];
        Queue<int[]> q = new LinkedList<int[]>();
        int[] start = {0, 0}; // starting cell
        q.add(start);
        vis[0][0] = true;
        while (!q.isEmpty()) {
            int[] cur = q.poll();
            int curRow = cur[0];
            int curCol = cur[1];
            for (int i = 0; i < 4; i++) {
                int newRow = curRow + deltaRow[i];
                int newCol = curCol + deltaCol[i];
                if (newRow < 0 || newRow >= n || newCol < 0 || newCol >= n) { // outside of boundary
                    continue;
                }
                if (vis[newRow][newCol]) { // visited node before
                    continue;
                }
                if (grid[newRow][newCol] > t) { // check if cell can be traversed
                    continue;
                }
                vis[newRow][newCol] = true;
                int[] destination = {newRow, newCol};
                q.add(destination);
            }
        }
        return vis[n - 1][n - 1];
    }
    public int swimInWater(int[][] grid) {
        int n = grid.length;
        int low = -1; // all values smaller or equal to low are not good
        int high = n * n; // all values greater or equal to high are good
        int mid = (low + high) / 2;
        while (low + 1 < high) {
            if (isEndReachable(grid, mid)) {
                high = mid;
            } else {
                low = mid;
            }
            mid = (low + high) / 2;
        }
        return high;
    }
}
```

Python Solution

```
class Solution:
    def swimInWater(self, grid: List[List[int]]) -> int:
        deltaRow = [-1, 0, 1, 0]
        deltaCol = [0, 1, 0, -1]

        def isEndReachable(grid, t):
            if grid[0][0] > t:  # starting elevation can't exceed t
                return False
            n = len(grid)
            vis = [[False] * n for a in range(n)]
            q = [(0, 0)]  # starting cell
            vis[0][0] = True
            while len(q):
                (curRow, curCol) = q.pop()
                for i in range(4):
                    newRow = curRow + deltaRow[i]
                    newCol = curCol + deltaCol[i]
                    if newRow < 0 or newRow >= n or newCol < 0 or newCol >= n:
                        # outside of boundary
                        continue
                    if vis[newRow][newCol]:  # visited node before
                        continue
                    if grid[newRow][newCol] > t:  # check if cell can be traversed
                        continue
                    vis[newRow][newCol] = True
                    q.append([newRow, newCol])
            return vis[n - 1][n - 1]

        n = len(grid)
        low = -1  # all values smaller or equal to low are not good
        high = n * n  # all values greater or equal to high are good
        mid = (low + high) // 2
        while low + 1 < high:
            if isEndReachable(grid, mid):
                high = mid
            else:
                low = mid
            mid = (low + high) // 2
        return high
```