Swim in Rising Water

On an N x N `grid`, each square `grid[i][j]` represents the elevation at that point `(i,j)`.

Now 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 distance in zero time. Of course, you must stay within the boundaries of the grid during your swim.

You start at the top left square `(0, 0)`. What is the least time until you can reach the bottom right square `(N-1, N-1)`?

Example 1:

```
Input: [[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:

```
Input: [[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:
 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

The final route is marked in bold.
We need to wait until time 16 so that (0, 0) and (4, 4) are connected.
```

Note:

1. `2 <= N <= 50`.

2. grid[i][j] is a permutation of [0, ..., N*N - 1].

Solution 1

Binary Search + DFS, O(n^2logn), 14ms
Binary Search range [0, n*n-1] to find the minimum feasible water level. For each water level, verification using DFS or BFS is O(n^2). DFS is slightly faster in practice.

```
class Solution {
public:
    int swimInWater(vector<vector<int>>& grid) {
        int n = grid.size();
        int low = grid[0][0], hi = n*n-1;
        while (low < hi) {
            int mid = low + (hi-low)/2;
            if (valid(grid, mid))
               hi = mid;
            else
               low = mid+1;
        }
        return low;
    }
private:
    bool valid(vector<vector<int>>& grid, int waterHeight) {
        int n = grid.size();
        vector<vector<int>> visited(n, vector<int>(n, 0));
        vector<int> dir({-1, 0, 1, 0, -1});
        return dfs(grid, visited, dir, waterHeight, 0, 0, n);
    }
    bool dfs(vector<vector<int>>& grid, vector<vector<int>>& visited, vector<int>& d
ir, int waterHeight, int row, int col, int n) {
        visited[row][col] = 1;
        for (int i = 0; i < 4; ++i) {
            int r = row + dir[i], c = col + dir[i+1];
            if (r >= 0 && r < n && c >= 0 && c < n && visited[r][c] == 0 && grid[r]
[c] <= waterHeight) {
                if (r == n-1 && c == n-1) return true;
                if (dfs(grid, visited, dir, waterHeight, r, c, n)) return true;
            }
        }
        return false;
    }
};
```

Dijkstra using Priority Queue, O(n^2logn), 20 ms;
In every step, find lowest water level to move forward, so using PQ rather than queue

```
class Solution {
public:
    int swimInWater(vector<vector<int>>& grid) {
        int n = grid.size(), ans = max(grid[0][0], grid[n-1][n-1]);
        priority_queue<vector<int>, vector<vector<int>>, greater<vector<int>>> pq;
        vector<vector<int>> visited(n, vector<int>(n, 0));
        visited[0][0] = 1;
        vector<int> dir({-1, 0, 1, 0, -1});
        pq.push({ans, 0, 0});
        while (!pq.empty()) {
            auto cur = pq.top();
            pq.pop();
            ans = max(ans, cur[0]);
            for (int i = 0; i < 4; ++i) {
                int r = cur[1] + dir[i], c = cur[2] + dir[i+1];
                if (r >= 0 && r < n && c >= 0 && c < n && visited[r][c] == 0) {
                    if (r == n-1 && c == n-1) return ans;
                    pq.push({grid[r][c], r, c});
                    visited[r][c] = 1;
                }
            }
        }
        return -1;
    }
};
```

Dijkstra with BFS optimization, O(n^2logn), 11 ms

Similar to above solution, but we can use BFS, which is more efficient, to expand reachable region.

```
class Solution {
public:
    int swimInWater(vector<vector<int>>& grid) {
        int n = grid.size(), ans = max(grid[0][0], grid[n-1][n-1]);
        priority_queue<vector<int>, vector<vector<int>>, greater<vector<int>>> pq;
        vector<vector<int>> visited(n, vector<int>(n, 0));
        visited[0][0] = 1;
        vector<int> dir({-1, 0, 1, 0, -1});
        pq.push({ans, 0, 0});
        while (!pq.empty()) {
            auto cur = pq.top();
            pq.pop();
            ans = max(ans, cur[0]);
            queue<pair<int, int>> myq;
            myq.push({cur[1], cur[2]});
            while (!myq.empty()) {
                auto p = myq.front();
                myq.pop();
                if (p.first == n-1 && p.second == n-1) return ans;
                for (int i = 0; i < 4; ++i) {
                    int r = p.first + dir[i], c = p.second + dir[i+1];
                    if (r >= 0 && r < n && c >= 0 && c < n && visited[r][c] == 0) {
                        visited[r][c] = 1;
                        if (grid[r][c] <= ans)
                           myq.push({r, c});
                        else
                           pq.push({grid[r][c], r, c});
                    }
                }
            }
        }
        return -1;
    }
};
```

written by zestypanda original link here

Solution 2

```
final static int[][] steps = {{0,1},{0,-1}, {1,0},{-1,0}};
public int swimInWater(int[][] grid) {
    int n = grid.length;
    int[][] max = new int[n][n];
    for(int[] line : max)
        Arrays.fill(line, Integer.MAX_VALUE);
    dfs(grid, max, 0, 0, grid[0][0]);
    return max[n-1][n-1];
}

private void dfs(int[][] grid, int[][] max, int x, int y, int cur) {
    int n = grid.length;
    if (x < 0 || x >= n || y < 0 || y >= n || Math.max(cur, grid[x][y]) >= max[x][y]
)
        return;
    max[x][y] = Math.max(cur, grid[x][y]);
    for(int[] s : steps) {
        dfs(grid, max, x + s[0], y + s[1], max[x][y]);
    }
}
```

written by tyuan73 original link here

Solution 3

```
class Solution:
    def swimInWater(self, grid):
        def zhaobaba(x, y):
            if baba[x][y] == (x, y):
                return baba[x][y]
            baba[x][y] = zhaobaba(baba[x][y][0], baba[x][y][1])
            return baba[x][y]

        def hebing(a, b):
            a = zhaobaba(a[0], a[1])
            b = zhaobaba(b[0], b[1])
            if size[a[0]][a[1]] < size[b[0]][b[1]]:
                baba[a[0]][a[1]] = b
                size[b[0]][b[1]] += size[a[0]][a[1]]
            else:
                baba[b[0]][b[1]] = a
                size[a[0]][a[1]] += size[b[0]][b[1]]

        WEI = [[-1, 0], [0, -1], [1, 0], [0, 1]]

        pos = {}
        m = len(grid)
        n = len(grid[0])
        for i in range(m):
            for j in range(n):
                pos[grid[i][j]] = (i, j)

        baba = []
        size = []
        for i in range(m):
            new_list = []
            for j in range(n):
                new_list.append((i, j))
            baba.append(new_list)
            size.append([1] * n)

        for i in range(m * n):
            x, y = pos[i]
            for wei in WEI:
                tx = x + wei[0]
                ty = y + wei[1]
                if tx >= 0 and tx < m and ty >= 0 and ty < n:
                    if grid[tx][ty] <= i:
                        hebing((x, y), (tx, ty))
                        if zhaobaba(0, 0) == zhaobaba(m - 1, n - 1):
                            return i
```

written by figurative original link here