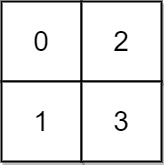
<https://leetcode.com/problems/swim-in-rising-water/description/>

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:**



**Input:** grid = [[0,2],[1,3]]

**Output:** 3

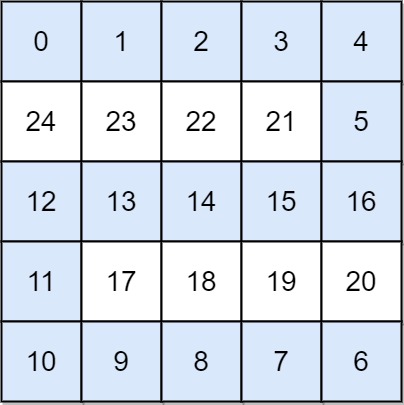
**Explanation:**

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:**



**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 <= n <= 50

0 <= grid[i][j] < n^2

Each value grid[i][j] is unique.

**Attempt 1: 2024-12-17**

**Solution 1: Djikstra (120 min, a bit modification based on problem requirement, but still follow classic Djikstra template and no visited array required)**

**虽然是用minPQ选择路径（基于下一个cell的elevation最小值筛选），但本题却是非典型Djikstra，不是传统Dijkstra求最短路径，而是找到拥有最小的最大值的路径（因为不同的路径可能拥有不同的最大值），并返回该最小的最大值**

class Solution {

    public int swimInWater(int[][] grid) {

        int n = grid.length;

        int[] dx = new int[] {0, 0, 1, -1};

        int[] dy = new int[] {1, -1, 0, 0};

        // Dijkstra algorithm initialize with all cells time as max value

        // as n^2 + 1 (because 0 <= grid[i][j] < n^2), except the start

        // cell [0, 0] as grid[0][0]

        int[][] times = new int[n][n];

        for(int i = 0; i < n; i++) {

            Arrays.fill(times[i], n \* n + 1);

        }

        times[0][0] = grid[0][0];

        int result = 0;

        // Dijkstra algorithm requires minimum heap, sort by time(based on elevation)

        PriorityQueue<int[]> minPQ = new PriorityQueue<>((a, b) -> a[2] - b[2]);

        minPQ.offer(new int[]{0, 0, grid[0][0]});

        while(!minPQ.isEmpty()) {

            int[] cur = minPQ.poll();

            int x = cur[0];

            int y = cur[1];

            int time = cur[2];

            // Critical difference then classic Djikstra (e.g L1368)

            // The additional line exactly represents what is the problem

            // looking for, we are not using Djikstra to find 'minimum

            // distance path'(the minimum required wait time path here),

            // even we still use Djikstra's minimum priority queue to

            // find the only wanted minimum required wait time path,

            // but we are not return any minimum accumulated time, instead,

            // have to return the 'maximum time(elevation)' value along

            // this 'minimum time required path', so each time we poll

            // out one 'confirmed' cell from this path(since all cells

            // added into minimum priority queue are cells on this path

            // and add previously), we compare and find the maximum one

            result = Math.max(time, result);

            // If we've reached the bottom-right cell, return the time

            if(x == n - 1 && y == n - 1) {

                return result;

            }

            // Skip if encounter same cell again and cell's time is outdated

            if(time > times[x][y]) {

                continue;

            }

            for(int k = 0; k < 4; k++) {

                int new\_x = x + dx[k];

                int new\_y = y + dy[k];

                    int new\_time = grid[new\_x][new\_y];

                    // Dijkstra algorithm only update the path if

                    // a smaller time is found till current cell

                    if(new\_time < times[new\_x][new\_y]) {

                        times[new\_x][new\_y] = new\_time;

                        minPQ.offer(new int[] {new\_x, new\_y, new\_time});

                    }

                }

            }

        }

        return result;

    }

}

Time Complexity: O(n^2 \* logn)

1.minPQ contains at most n^2 elements, pop time complexity each time is O(logn^2) = O(2\*logn)

2.At most we will pop n^2 times

O(n^2\*2\*logn) = O(n^2\*logn)

Space Complexity: O(n^2)

**Solution 2: Binary Search + DFS (10 min)**

**基于题目定义的以下2个条件，我们可以使用 Binary Search 来寻找最小的但足以覆盖最终路径中的最大值的中位数 time**

**1. 0 <= grid[i][j] < n^2**

**2. Each value grid[i][j] is unique.**

class Solution {

    int[] dx = new int[] {0, 0, 1, -1};

    int[] dy = new int[] {1, -1, 0, 0};

    public int swimInWater(int[][] grid) {

        int n = grid.length;

        // 0 <= grid[i][j] < n^2

        int lo = grid[0][0];

        int hi = n \* n - 1;

        // Find lower boundary

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            // If current 'mid'(wait time) able to reach,

            // we try to move 'hi' backward to 'mid - 1'

            // to attempt if smaller 'mid' able to reach,

            // otherwise try to move 'lo' forward to

            // 'mid + 1' to attempt if larger 'mid' able

            // to reach

            if(canReach(grid, mid, n)) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        return lo;

    }

    private boolean canReach(int[][] grid, int mid, int n) {

        boolean[][] visited = new boolean[n][n];

        return helper(grid, mid, n, 0, 0, visited);

    }

    // DFS to check if we can reach (n-1, n-1)

    private boolean helper(int[][] grid, int waitTime, int n, int x, int y, boolean[][] visited) {

        // Out of bounds, already visited, or exceeds waitTime

        if(x < 0 || x >= n || y < 0 || y >= n || grid[x][y] > waitTime || visited[x][y]) {

            return false;

        }

        // Check if we reached the bottom-right cell

        if(x == n - 1 && y == n - 1) {

            return true;

        }

        // Mark current cell as visited

        visited[x][y] = true;

        for(int k = 0; k < 4; k++) {

            int new\_x = x + dx[k];

            int new\_y = y + dy[k];

            if(helper(grid, waitTime, n, new\_x, new\_y, visited)) {

                return true;

            }

        }

        // No path found

        return false;

    }

}

Time Complexity: O(n^2 \* logn)

Space Complexity: O(n^2)

**Refer to**

<https://leetcode.com/problems/swim-in-rising-water/solutions/113758/c-two-solutions-binary-search-dfs-and-dijkstra-bfs-o-n-2logn-11ms/>

**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);

}

visited[row][col] = 1;

for (int i = 0; i < 4; ++i) {

int r = row + dir[i], c = col + dir[i+1];

if (r == n-1 && c == n-1) return true;

if (dfs(grid, visited, dir, waterHeight, r, c, n)) return true;

}

}

return false;

}

};

**2. 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]);

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 == n-1 && c == n-1) return ans;

pq.push({grid[r][c], r, c});

visited[r][c] = 1;

}

}

}

return -1;

}

};

**3. 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]);

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];

visited[r][c] = 1;

if (grid[r][c] <= ans)

myq.push({r, c});

else

pq.push({grid[r][c], r, c});

}

}

}

}

return -1;

}

};

**Refer to**

<https://leetcode.com/problems/swim-in-rising-water/solutions/1285099/easy-solution-w-explanation-optimization-from-brute-force-to-binary-search-beats-100/>

❌ ***Solution - I (Brute-Force)***

We can observe that the problem basically asks us to find a path such that the maximum waterLevel w\_lvl (value of a cell) of some intermediate cell of that path is minimized.

We can start with the brute force approach and explore all the possible paths to reach the end of the grid. The path which minimizes the maximum value of some cell in a path will be chosen and that water level - w\_lvl will be our answer.

int n;

int swimInWater(vector<vector<int>>& grid) {

    n = size(grid);

    vector<vector<int>> vis(n, vector<int>(n));

    return solve(grid, vis, 0, 0);

}

int solve(vector<vector<int>>& grid, vector<vector<int>> vis, int i, int j,

          int w\_lvl = 0) {

    if (i < 0 || j < 0 || i >= n || j >= n || vis[i][j])

        return 10000; // out-of-bounds / already visited

    if (i == n - 1 && j == n - 1)

        return max(w\_lvl, grid[i][j]);

    vis[i][j] =

        true; // mark as visited so we don't keep visiting back-and-forth

    w\_lvl = max(w\_lvl,

                grid[i][j]); // update max intermediate value for current path

    // explore all 4 possible options from current cell and choose the path with

    // minimum w\_lvl -

    return min(

        {solve(grid, vis, i + 1, j, w\_lvl), solve(grid, vis, i - 1, j, w\_lvl),

         solve(grid, vis, i, j + 1, w\_lvl), solve(grid, vis, i, j - 1, w\_lvl)});

}

Time Complexity : O(4n^2), for each cell of the grid, we have 4 choices. So, in the worst case we may need 4\*4\*4\*...n2times

Space Complexity : O(n^2) for the optimzed verison, O(n^4) for the first version since copy for vis is made for each recursive call

✔️ ***Solution - II (Search lowest valid water-level)***

Exploring all the possible path options is very time-consuming. Instead of exploring all the available paths and then choosing the path with lowest maximum water level - w\_lvl , we can instead set an upper-limit to w\_lvl ourselves, starting from 1 till n\*n-1. The lowest water level required to reach the end will be our answer.

Here, instead of exploring all paths, we have limited ourselves to only exploring a single path which has maximum intermediate cell value = w\_lvl and returning as soon as we find the first path that takes us to the end. Thus, we have limited our search space to a great extent.

Another small optimization is to start checking directly from max of max( grid[0][0], grid[n-1][n-1], 2\*(n-1) ) instead of w\_lvl=1 (Credits to @ud240 & @nicolattu). That's because every path will contain grid[0][0] and grid[n-1][n-1], and 2\*(n-1) because grid elements are always a permutation of 0...n\*n-1 and thus there will always be a cell with value >= 2\*(n-1) in every path.

int n, moves[4][2]{{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

int swimInWater(vector<vector<int>>& grid) {

    n = size(grid);

    vector<vector<int>> vis(n, vector<int>(n));

    int minReq = max({2 \* (n - 1), grid[0][0], grid[n - 1][n - 1]});

    for (int w\_lvl = minReq; w\_lvl < n \* n; w\_lvl++) {

        if (dfs(grid, vis, 0, 0, w\_lvl))

            return w\_lvl;

        for\_each(begin(vis), end(vis), [](auto& v) {

            fill(begin(v), end(v), 0);

        }); // reset the vis array back to not-visited

    }

    return n \* n;

}

         int w\_lvl) {

    if (i < 0 || j < 0 || i >= n || j >= n || vis[i][j] || grid[i][j] > w\_lvl)

        return false; // out-of-bound / already visited / cell value > max w\_lvl

                      // allowed

    if (i == n - 1 && j == n - 1)

        return true;

    vis[i][j] = true;

    for (int k = 0; k < 4;

         k++) // search all available option till any of it leads us to the end

        if (dfs(grid, vis, i + moves[k][0], j + moves[k][1], w\_lvl))

            return true;

    return false;

}

Time Complexity : O(n^4), we are exploring all water-level values from 1 to n\*n. For each water-level MAX, we call dfs() having O(n2) time complexity in the worst case. Thus, overall time complexity becomes O(n\*n) \* O(n^2) = O(n^4)

Space Complexity : O(n^2)

✔️ ***Solution - III (Binary-Search)***

Instead of searching linearly as we did in the above approach, we can use binary search to find the lowest valid water-level. Our search space is sorted and it the range [1...n\*n-1] and thus binary search can be applied here.

In previous approach, for the worst-case scenario we would have called dfs() for n\*n times.

Binary search will reduce worst-case number of calls required to O(logn2) = O(logn).

Also, I have replaced the vis below with a boolean C-style array (for faster runtime) of 50x50 which is the max grid dimensions. Since the constraints are small, I have opted for fixed size declaration intead dynamic allocation, the syntax of which is slightly messy.

int n, moves[4][2]{{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

int swimInWater(vector<vector<int>>& grid) {

    n = size(grid);

    int l = max({2 \* (n - 1), grid[0][0], grid[n - 1][n - 1]}), r = n \* n - 1,

        mid;

    bool vis[50][50]{};

    while (l <= r) {

        mid = (l + r) / 2;

        if (solve(grid, vis, 0, 0, mid))

            r = mid - 1; // we are able to reach end with MAX=mid   => try lower

                         // to minimze it

        else

            l = mid + 1; // we fail to reach the end with MAX=mid   => try

                         // higher so we can reach the end

        memset(vis, false, sizeof vis); // reset vis back to all false

    }

    return l;

}

bool solve(vector<vector<int>>& grid, bool vis[][50], int i, int j, int w\_lvl) {

    if (i < 0 || j < 0 || i >= n || j >= n || vis[i][j] || grid[i][j] > w\_lvl)

        return false;

    if (i == n - 1 && j == n - 1)

        return true;

    vis[i][j] = true;

    for (int k = 0; k < 4; k++)

        if (solve(grid, vis, i + moves[k][0], j + moves[k][1], w\_lvl))

            return true;

    return false;

}

Time Complexity : O(n^2logn), O(n^2) for dfs call and O(logn) for binary search over 1...n\*n.

Space Complexity : O(n^2)

**Refer to**

<https://leetcode.com/problems/swim-in-rising-water/solutions/113770/c-python-priorityqueue/comments/381671>

Q: Anyone can explanin

1) why priority queue is came up to solve this problem but not other algos like dp

2) How can reason that the solution by this priority queue is global optimal?

Thank you guys very much for providing the solutions and the useful discussion. BTW I used recursion + memozation but TLE when N exceeds 3.

<https://leetcode.com/problems/swim-in-rising-water/solutions/113770/c-python-priorityqueue/comments/459772>

A1: I guess the idea comes from the Dijstra's algorithm for solving shortest path problem.

<https://leetcode.com/problems/swim-in-rising-water/solutions/113770/c-python-priorityqueue/comments/673999>

A2: If we can only go down and right, DP is nice. But we can go 4 directional so DP is super slow. There are so many states possible

**这个说法很不错，在**[L2556.Disconnect Path in a Binary Matrix by at Most One Flip](note://WEB3322914f94be5e34556afdcf5b35b500)**的DP解中得到验证**

<https://leetcode.com/problems/disconnect-path-in-a-binary-matrix-by-at-most-one-flip/solutions/3142814/clever-diagonals-with-diagram-explanation/>

**Refer to**

[L1368.Minimum Cost to Make at Least One Valid Path in a Grid (Ref.L743,L2290)](note://WEB4b3aa9003dbf5915537540decf09b434)

[L2556.Disconnect Path in a Binary Matrix by at Most One Flip](note://WEB3322914f94be5e34556afdcf5b35b500)