778. Swim in Rising Water

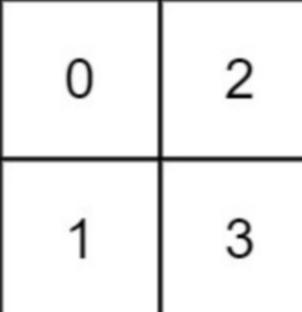
Leetcode Link

You are given an $n \times n$ 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:



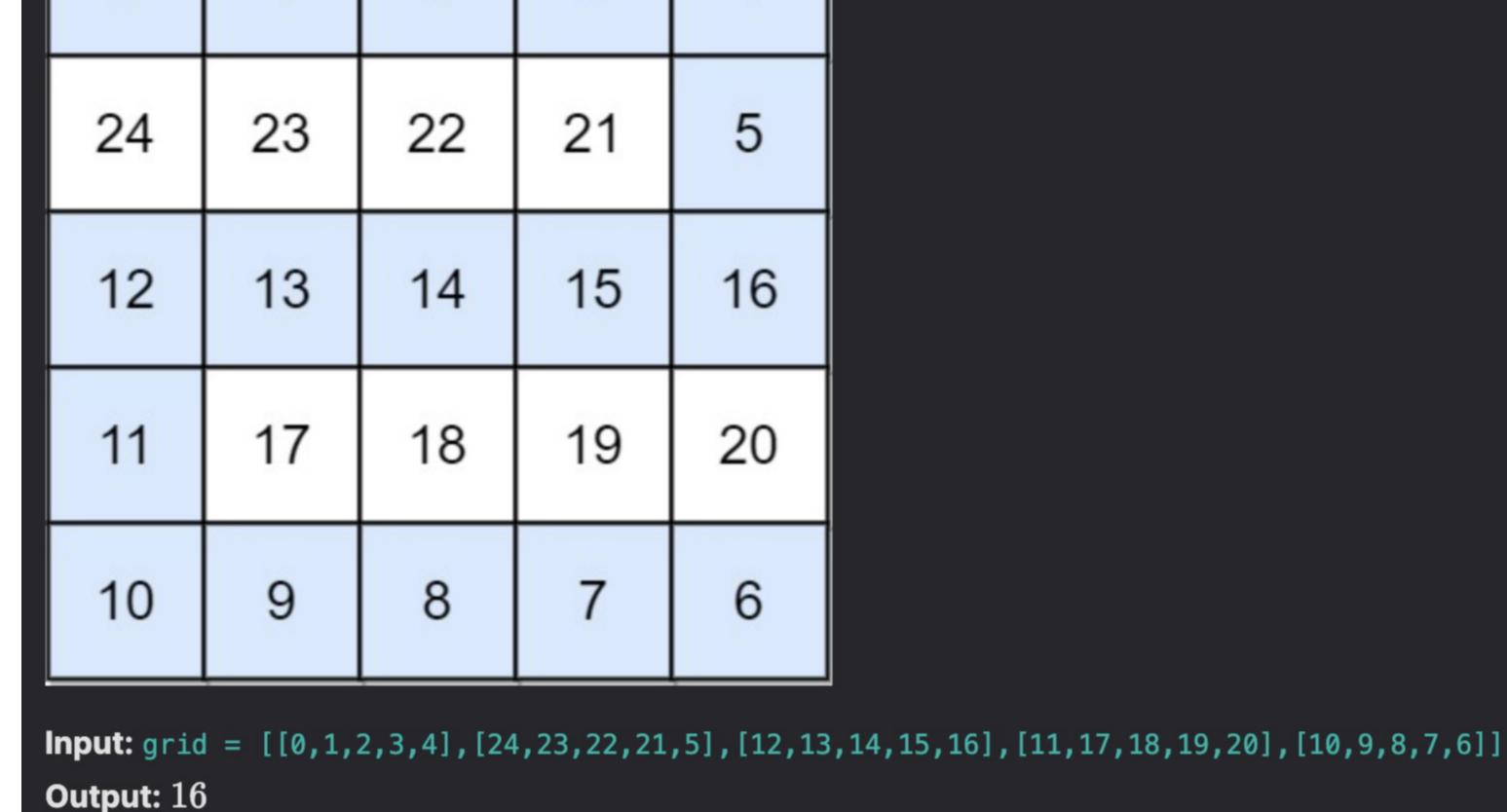
Output: 3

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

Explanation:

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:

At time 0, you are in grid location (0, 0). You cannot go anywhere else because 4-directionally adjacent neighbors have a higher



We need to wait until time 16 so that (0, 0) and (4, 4) are connected.

Constraints:

Explanation: The final route is shown.

• n == grid.length

3

4

• n == grid[i].length • $1 \le n \le 50$

```
• 0 \leq \text{grid[i][j]} < n^2
  • Each value grid[i][j] is unique.
Solution
```

Brute Force

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**

In simpler terms, this problem is asking us what's the minimum value of $m{t}$ such that you can travel from the top-left square to the

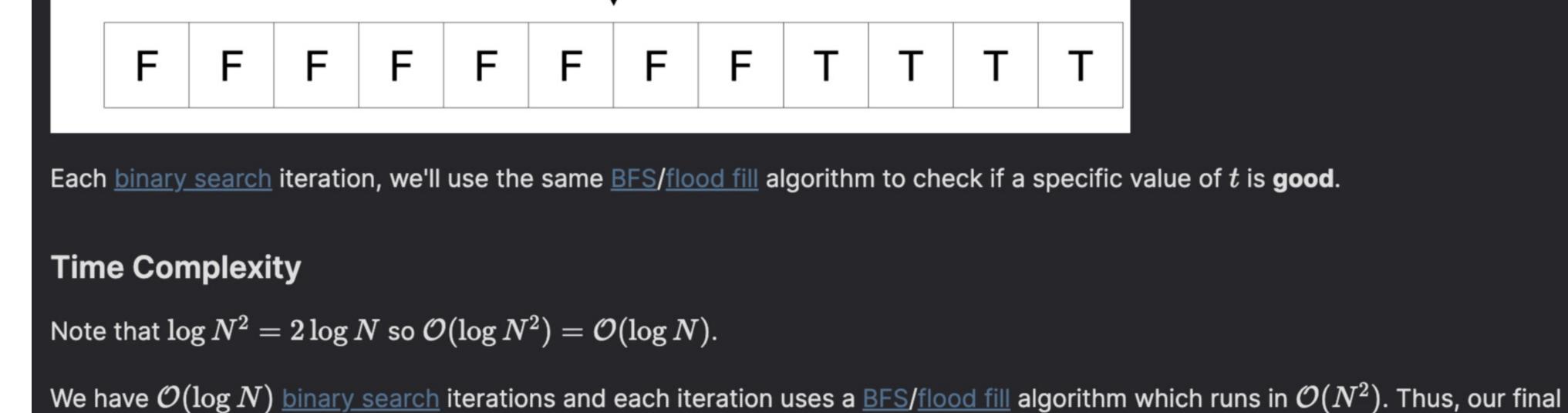
Let's denote a value t as ${f good}$ if the bottom-right square is reachable from the top-left square only through squares with elevations

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 $t \ge k$ are all **good**. What does this mean for us? It means we can binary search it since our binary search

condition is satisfied.

... k-2 k-1 k k+1 ... Is the value of t good?



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 $\mathcal{O}(N^2)$.

const vector<int> deltaRow = $\{-1, 0, 1, 0\}$;

const vector<int> deltaCol = $\{0, 1, 0, -1\}$;

q.push({0, 0}); // starting cell

int curRow = cur[0];

int curCol = cur[1];

vector<int> cur = q.front();

return false;

vis[0][0] = true;

q.pop();

while (!q.empty()) {

bool isEndReachable(vector<vector<int>>& grid, int t) {

if (grid[0][0] > t) { // starting elevation can't exceed t

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

int n = grid.size(); vector<vector<bool>> vis(n, vector<bool>(n)); 9 10 queue<vector<int>> q;

11

12

13

14

15

16

17

1 class Solution {

```
for (int i = 0; i < 4; i++) {
 18
 19
                     int newRow = curRow + deltaRow[i];
                     int newCol = curCol + deltaCol[i];
 20
 21
                     if (newRow < 0 || newRow >= n || newCol < 0 || newCol >= n) { // outside of boundary
 22
                         continue;
 23
                     if (vis[newRow][newCol]) { // visited node before
 24
 25
                         continue;
 26
 27
                     if (grid[newRow][newCol] > t) { // check if cell can be traversed
 28
                         continue;
 29
 30
                     vis[newRow][newCol] = true;
 31
                     q.push({newRow, newCol});
 32
 33
 34
             return vis[n-1][n-1];
 35
 36
 37
        public:
         int swimInWater(vector<vector<int>>& grid) {
 38
 39
             int n = grid.size();
 40
             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
 41
             int mid = (low + high) / 2;
 42
             while (low + 1 < high) {
 43
                 if (isEndReachable(grid, mid)) {
 44
 45
                     high = mid;
 46
                 } else {
 47
                     low = mid;
 48
 49
                 mid = (low + high) / 2;
 50
 51
             return high;
 52
 53 };
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;
  8
             boolean[][] vis = new boolean[n][n];
  9
 10
             Queue<int[]> q = new LinkedList<int[]>();
 11
             int[] start = {0, 0}; // starting cell
 12
             q.add(start);
 13
             vis[0][0] = true;
 14
             while (!q.isEmpty()) {
 15
                 int[] cur = q.poll();
 16
                 int curRow = cur[0];
 17
                 int curCol = cur[1];
 18
                 for (int i = 0; i < 4; i++) {
 19
                     int newRow = curRow + deltaRow[i];
 20
                     int newCol = curCol + deltaCol[i];
 21
                     if (newRow < 0 || newRow >= n || newCol < 0 || newCol >= n) { // outside of boundary
```

30 31 32 33

continue;

continue;

continue;

if (vis[newRow][newCol]) { // visited node before

if (grid[newRow][newCol] > t) { // check if cell can be traversed

22

23

24

25

26

27

28

29

25

26

27

28

29

30

31

32

33

34

35

40

```
vis[newRow][newCol] = true;
                     int[] destination = {newRow, newCol};
                     q.add(destination);
 34
 35
             return vis[n - 1][n - 1];
 36
 37
         public int swimInWater(int[][] grid) {
 38
             int n = grid.length;
 39
             int low = -1; // all values smaller or equal to low are not good
 40
             int high = n * n; // all values greater or equal to high are good
 41
             int mid = (low + high) / 2;
             while (low + 1 < high) {</pre>
 42
 43
                 if (isEndReachable(grid, mid)) {
                     high = mid;
 44
 45
                 } else {
 46
                     low = mid;
 47
 48
                 mid = (low + high) / 2;
 49
 50
             return high;
 51
 52 }
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
  8
  9
                 n = len(grid)
 10
                 vis = [[False] * n for a in range(n)]
                 q = [(0, 0)] # starting cell
 11
 12
                 vis[0][0] = True
 13
                 while len(q):
 14
                     (curRow, curCol) = q.pop()
 15
                     for i in range(4):
 16
                         newRow = curRow + deltaRow[i]
 17
                         newCol = curCol + deltaCol[i]
 18
                         if newRow < 0 or newRow >= n or newCol < 0 or newCol >= n:
 19
                             # outside of boundary
 20
                             continue
                         if vis[newRow][newCol]: # visited node before
 21
 22
                             continue
                         if grid[newRow][newCol] > t: # check if cell can be traversed
 23
 24
                             continue
```

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

vis[newRow][newCol] = True

q.append([newRow, newCol])

low = -1 # all values smaller or equal to low are not good

high = n * n # all values greater or equal to high are good

return vis[n-1][n-1]

if isEndReachable(grid, mid):

mid = (low + high) // 2

high = mid

while low + 1 < high:

n = len(grid)