755. Pour Water

Medium

## **Problem Description**

<u>Array</u>

Simulation

In this problem, we are provided with an elevation map represented by an array heights where each element heights [i] indicates the height of the terrain at index i. The width of each terrain column is 1. We are given a volume of water volume and a

starting position k where the water starts to fall. The water droplet will behave according to the following rules:

1. The water droplet initially falls onto the terrain or the water at index k.

- 2. The droplet then tries to move according to these conditions: It flows to the left if it eventually would fall to a lower height.
- If it can't move left, it flows to the right if it would eventually fall to a lower height in that direction.
  - The term eventually fall indicates that there is a path for the droplet to reach a lower altitude than its current one by moving
  - continuously in that direction. Water can only sit on top of terrain or other water, and it will always occupy a full index-width block. There are infinitely high walls on both sides of the array boundaries, hence water cannot spill outside the terrain array

If the droplet cannot move left or right to a lower height, it stays at its current position.

Intuition

The goal is to simulate the process of pouring volume units of water one by one at the index k and return the final distribution of

The solution is achieved through simulation of each water droplet's behavior as it falls onto the terrain and moves according to

## the defined rules. Here's how we approach the problem: 1. For every unit of water, we start at k and explore both the left and the right directions to find where the droplet would end up.

bounds.

water over the map.

2. We first check left – if there's a lower or equal height neighbor, we move in that direction. If we find a neighbor with strictly lower height, we mark it as a potential place for the water to drop. 3. If the droplet does find a place to fall on the left, we update the height array by adding one unit to the height at that position and then start the

process again with the next droplet. 4. In case we can't find a lower place for the droplet on the left, we switch direction and perform the same checks to the right.

5. If we find a spot where the water droplet can settle on the right, we do the same as we did when we found a spot on the left.

- 6. If there's no place to fall on both sides, the droplet stays at the initial position k, and we increase its height by one. 7. We continue this process until all volume units of water have been poured.
- Solution Approach
- The implementation given in the reference solution uses a direct approach to simulate the process of pouring water. The steps taken are a direct translation of the given rules into code. Here's a detailed description of how the solution works:
- The solution iterates over each unit of water in volume. For each iteration (representing a single unit of water), it attempts to find the correct position where the water should be poured.

elevation terrain in that direction, and updates j if it finds a lower elevation.

the droplet will eventually fall, and we can update the height at that position:

else block only if the loop is not terminated by a break.

## direction (d = -1) and then the right direction (d = 1) if needed:

**if** j != k:

else:

heights[j] += 1

next unit of water.

for d in (-1, 1): i = j = kwhile 0 <= i + d < len(heights) and heights[i + d] <= heights[i]:</pre> if heights[i + d] < heights[i]:</pre> j = i + d

We start from the position k, where the water droplet is initially placed. The solution uses a nested loop to first check the left

i += dThis while loop traverses the heights array either to the left or right from the current position i. It looks for lower or equal

If a new position j (which is lower than the current position k) is found in either left or right direction, it means that's where

break The condition jule k means that juhas updated to a new lower position, and we break out of the direction loop to process the

If we don't find such a position, it means the droplet will remain at position k, and the height at k needs to be incremented:

This process is repeated for every unit of volume until all water has been processed. This approach ensures that the droplet

will move according to the problem's constraints and the final heights array will reflect the new water levels after pouring all

- heights[k] += 1 Note that the else is attached to the for loop, and not the if statement - in Python, a for-else statement will execute the
- Finally, the modified heights array is returned, representing the elevation map after processing the water droplets:

return heights

volume units of water.

the simulation nature of this problem.

The algorithm efficiently uses a simple simulation iterating over the number of water units without any additional data structures. The traversal for finding the correct spot for each water droplet is done in linear time relative to the size of heights, and this is repeated for each water unit, giving us an overall time complexity of O(volume \* size of heights), which is quite appropriate given

**Example Walkthrough** Let's illustrate the solution approach with a small example. Suppose we have the following elevation map and parameters:

Checking to the left, we find that heights[1] is equal to heights[2], but since heights[0] is higher, the water cannot fall

Checking to the right, we also find that heights [3] and heights [4] are equal to or higher than heights [2]. So the droplet

• volume = 4 k = 2 (the starting position for pouring water)

For the second unit of water, we repeat the process starting at the updated index  $2 \left( \frac{1}{2} \right) = 2$ .

We find that heights [1] is lower than heights [2] now, so the droplet will fall to index 1.

We'll walk through the simulation process: For the first unit of water, we start at k which is index 2 (heights [2] = 1).

• heights = [2, 1, 1, 2, 1, 2]

## New heights = [2, 1, 2, 2, 1, 2]

10.

11.

**Python** 

Java

class Solution {

class Solution:

for \_ in range(volume):

for direction in (-1, 1):

break

further left.

Increment heights [1] by 1 since the water droplet settled there.

The third unit of water starts at k (index 2) again (heights[2] = 2).

Similar to before, the water droplet moves to the left and settles at index 1.

New heights = [2, 2, 2, 2, 1, 2]

and settle at index 4.

stays at k, and we increment heights [2] by 1.

New heights = [2, 3, 2, 2, 1, 2] The fourth and final unit of water starts at k (index 2).

Moving to the right, heights [3] is equal to heights [2], and heights [4] is lower, so the water droplet will move to the right

After pouring all 4 units of water, the final distribution of water over the elevation map is [2, 3, 2, 2, 2, 2]. This demonstrates

Increment heights [4] by 1. 12. • Final heights = [2, 3, 2, 2, 2, 2]

how the solution algorithm successfully follows the droplet rules to simulate water pouring onto the terrain.

This time, since heights [1] is now higher than heights [2], the droplet can't move left.

Solution Implementation

def pour\_water(self, heights: List[int], volume: int, drop\_position: int) -> List[int]:

# Try to pour water to the left (-1) first, and then to the right (+1)

# Increment the height of the water at the drop position itself

// Method to simulate pouring water over a set of columns represented by heights

int currentIndex = position, lowestIndex = position;

// Move from the position to the direction indicated by d

// Moving to the next column if the condition is met

# Move along the height array in the specified direction

current\_position = best\_position = drop\_position

# Initialize current position i and best position j to the drop position

# Loop for each unit of water volume to be poured

current\_position += direction

if best\_position != drop\_position:

heights[drop\_position] += 1

heights[best\_position] += 1

# Return the modified heights list after pouring the water

public int[] pourWater(int[] heights, int volume, int position) {

// Loop until all units of volume have been poured

currentIndex += direction;

while 0 <= current\_position + direction < len(heights) and heights[current\_position + direction] <= heights[current\_position + direction]</pre> # If the next position is lower, update the best possible position if heights[current\_position + direction] < heights[current\_position]:</pre> best\_position = current\_position + direction # Move to the next position

else: # This executes only if the 'break' was not hit, meaning water couldn't go left or right

// Check if the current index is within bounds and if the next column is equal or lower

while (currentIndex + direction >= 0 && currentIndex + direction < heights.length &&</pre>

heights[currentIndex + direction] <= heights[currentIndex]) {

# If we have found a position (other than the drop position) to pour water, increment the height there

```
while (volume-- > 0) {
    boolean poured = false; // Indicator if water has been poured
    // Two directions: left (d=-1), and right (d=1)
    for (int direction = -1; direction <= 1 && !poured; direction += 2) {
```

return heights

```
// If the next column is lower, update the lowest index
                    if (heights[currentIndex] < heights[lowestIndex]) {</pre>
                        lowestIndex = currentIndex;
                // Pouring water into the lowest column if it is different from the starting position
                if (lowestIndex != position) {
                    poured = true; // Water has been poured
                    heights[lowestIndex]++; // Increment the height of the lowest column
           // If water has not been poured in either direction, pour it at the position
            if (!poured) {
                heights[position]++;
        return heights; // Return the modified ground after pouring all units of volume
C++
#include <vector>
using std::vector;
class Solution {
public:
   // Method to simulate pouring water over the heights
    vector<int> pourWater(vector<int>& heights, int volume, int index) {
       // Loop to pour water up to the given volume
       while (volume-- > 0) {
            bool waterPoured = false; // Flag to check if water was poured
            // Check both directions, left first (-1) then right (1)
            for (int direction = -1; direction <= 1 && !waterPoured; direction += 2) {
                int currentPosition = index, lowestPosition = index;
```

// Move in the current direction as long as the next position is within bounds

if (heights[currentPosition + direction] < heights[currentPosition]) {</pre>

heights[currentPosition + direction] <= heights[currentPosition]) {

while (currentPosition + direction >= 0 && currentPosition + direction < heights.size() &&

// and the height at the next position is not greater than the current one.

// If the next position is lower, update the lowestPosition

// Move in the current direction as long as the next position is within bounds

heights[currentPosition + direction] <= heights[currentPosition]) {

// Pour water into the lowest position found, if it's different from the starting index

heights[lowestPosition]++; // Increment the height at the lowest position

if (heights[currentPosition + direction] < heights[currentPosition]) {</pre>

// and the height at the next position is not greater than the current one.

currentPosition + direction < heights.length &&

lowestPosition = currentPosition + direction;

waterPoured = true; // Water was successfully poured

def pour\_water(self, heights: List[int], volume: int, drop\_position: int) -> List[int]:

# Try to pour water to the left (-1) first, and then to the right (+1)

# Move along the height array in the specified direction

best\_position = current\_position + direction

position k until it finds a suitable place to drop the water or returns to the starting index k.

current\_position = best\_position = drop\_position

# Initialize current position i and best position j to the drop position

# If the next position is lower, update the best possible position

if heights[current\_position + direction] < heights[current\_position]:</pre>

// If the next position is lower, update the lowestPosition

lowestPosition = currentPosition + direction;

// Move to the next position

currentPosition += direction;

while (currentPosition + direction >= 0 &&

// Move to the next position

if (lowestPosition != index) {

// Return the modified heights array

for direction in (-1, 1):

for \_ in range(volume):

currentPosition += direction;

# Loop for each unit of water volume to be poured

# Move to the next position

current\_position += direction

```
// Pour water into the lowest position found, if it's different from the starting index
               if (lowestPosition != index) {
                   waterPoured = true; // Water was successfully poured
                    ++heights[lowestPosition]; // Increment the height at the lowest position
           // If we couldn't pour water to the left or right, pour it at the index (starting position)
           if (!waterPoured) {
               ++heights[index];
       // Return the modified heights vector
       return heights;
};
TypeScript
// Function to simulate pouring water over the heights
function pourWater(heights: number[], volume: number, index: number): number[] {
   // Loop to pour water up to the given volume
   while (volume-- > 0) {
        let waterPoured = false; // Flag to check if water was poured
       // Check both directions, left first (-1) then right (1)
        for (let direction = -1; direction <= 1 && !waterPoured; direction += 2) {
            let currentPosition = index;
            let lowestPosition = index;
```

```
// If we couldn't pour water to the left or right, pour it at the index (starting position)
if (!waterPoured) {
    heights[index]++;
```

class Solution:

return heights;

```
# If we have found a position (other than the drop position) to pour water, increment the height there
               if best_position != drop_position:
                   heights[best_position] += 1
                   break
           else: # This executes only if the 'break' was not hit, meaning water couldn't go left or right
               # Increment the height of the water at the drop position itself
               heights[drop_position] += 1
       # Return the modified heights list after pouring the water
       return heights
Time and Space Complexity
```

The time complexity of the given code is O(V \* N), where V is the volume of water to pour, and N is the length of the heights list.

This is because for each unit of volume, the code potentially traverses the heights list in both directions (-1 and 1) from the

while 0 <= current\_position + direction < len(heights) and heights[current\_position + direction] <= heights[current\_position]

and since there is a fixed volume V, the outer loop runs V times. Each time we are performing a comparison operation which is an 0(1) operation. Therefore, when combining these operations, the overall time complexity becomes 0(V \* N). The space complexity of the code is 0(1), assuming the input heights list is mutable and does not count towards space complexity (since we're just modifying it in place). This is because no additional significant space is allocated that grows with the size of the input; we only use a few extra variables (i, j, d, k) for indexing and comparison, which is constant extra space.

The inner while loop runs for at most N iterations (in the worst case where it goes from one end of the heights list to the other),