755. Pour Water

Medium Array Simulation

Leetcode Link

Problem Description

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

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 bounds.

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 water over the map.

Intuition

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.
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.

The provided code snippet successfully simulates this approach by incrementally placing each unit of water following the rules and updating the `heights` array.

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:

1. 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.
2. We start from the position `k`, where the water droplet is initially placed. The solution uses a nested loop to first check the left direction (`d = -1`) and then the right direction (`d = 1`) if needed:

```
1  for d in (-1, 1):
2      i = j = k
3      while 0 <= i + d < len(heights) and heights[i + d] <= heights[i]:
4          if heights[i + d] < heights[j]:
5              j = i + d
6          i += d
```

   This while loop traverses the `heights` array either to the left or right from the current position `i`. It looks for lower or equal elevation terrain in that direction, and updates `j` if it finds a lower elevation.

3. If a new position `j` (which is lower than the current position `i`) is found in either left or right direction, it means that's where the droplet will eventually fall, and we can update the height at that position:

```
1  if j != k:
2      heights[j] += 1
3      break
```

   The condition `j != k` means that `j` has updated to a new lower position, and we break out of the direction loop to process the next unit of water.

4. 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:

```
1  else:
2      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 `else` block only if the loop is not terminated by a `break`.

5. 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 `volume` units of water.
6. Finally, the modified `heights` array is returned, representing the elevation map after processing the water droplets:

```
1  return heights
```

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 the simulation nature of this problem.

Example Walkthrough

Let's illustrate the solution approach with a small example.

Suppose we have the following elevation map and parameters:

- `heights` = [2, 1, 1, 2, 1, 2]
- `volume` = 4
- `k` = 2 (the starting position for pouring water)

We'll walk through the simulation process:

1. For the first unit of water, we start at `k` which is index 2 (`heights[2] = 1`).
2. Checking to the left, we find that `heights[1]` is equal to `heights[2]`, but since `heights[0]` is higher, the water cannot fall further left.
3. Checking to the right, we also find that `heights[3]` and `heights[4]` are equal to or higher than `heights[2]`. So the droplet stays at `k`, and we increment `heights[2]` by 1.

    - New `heights` = [2, 1, 2, 2, 1, 2]
4. For the second unit of water, we repeat the process starting at the updated index 2 (`heights[2] = 2`).
5. We find that `heights[1]` is lower than `heights[2]` now, so the droplet will fall to index 1.
6. Increment `heights[1]` by 1 since the water droplet settled there.

    - New `heights` = [2, 2, 2, 2, 1, 2]
7. The third unit of water starts at `k` (index 2) again (`heights[2] = 2`).
8. Similar to before, the water droplet moves to the left and settles at index 1.

    - New `heights` = [2, 3, 2, 2, 1, 2]
9. The fourth and final unit of water starts at `k` (index 2).
10. This time, since `heights[1]` is now higher than `heights[2]`, the droplet can't move left.
11. 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 and settle at index 4.
12. Increment `heights[4]` by 1.

    - Final `heights` = [2, 3, 2, 2, 2, 2]

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

Python Solution

```
1  class Solution:
2      def pour_water(self, heights: List[int], volume: int, drop_position: int) -> List[int]:
3          # Loop for each unit of water volume to be poured
4          for _ in range(volume):
5              # Try to pour water to the left (-1) first, and then to the right (1)
6              for direction in (-1, 1):
7                  # Initialize current position i and best position j to the drop position
8                  current_position = best_position = drop_position
9                  # Move along the height array in the specified direction
10                 while 0 <= current_position + direction < len(heights) and heights[current_position + direction] <= heights[current_
11                     # If the next position is lower, update the best possible position
12                     if heights[current_position + direction] < heights[best_position]:
13                         best_position = current_position + direction
14                     # Move to the next position
15                     current_position += direction
16                 # If a lower position than the drop position was found, increment the height there
17                 if best_position != drop_position:
18                     heights[best_position] += 1
19                     break
20             else:  # This executes only if the 'break' was not hit, meaning water couldn't go left or right
21                 # Increment the height of the water at the drop position itself
22                 heights[drop_position] += 1
23         # Return the modified heights array after pouring the water
24         return heights
25
```

Java Solution

```
1  class Solution {
2
3      // Method to simulate pouring water over a set of columns represented by heights
4      public int[] pourWater(int[] heights, int volume, int position) {
5          // Loop until all units of volume have been poured
6          while (volume-- > 0) {
7              boolean poured = false; // Indicator if water has been poured
8
9              // Two directions: left (d=-1), and right (d=1)
10             for (int direction = -1; direction <= 1 && !poured; direction += 2) {
11                 int currentIndex = position, lowestIndex = position;
12
13                 // Move from the position to the direction indicated by d
14                 // Check if the current index is within bounds and if the next column is equal or lower
15                 while (currentIndex + direction >= 0 && currentIndex + direction < heights.length &&
16                         heights[currentIndex + direction] <= heights[currentIndex]) {
17
18                     // Moving to the next column if the condition is met
19                     currentIndex += direction;
20
21                     // If the next column is lower, update the lowest index
22                     if (heights[currentIndex] < heights[lowestIndex]) {
23                         lowestIndex = currentIndex;
24                     }
25                 }
26
27                 // Pouring water into the lowest column if it is different from the starting position
28                 if (lowestIndex != position) {
29                     poured = true; // Water has been poured
30                     heights[lowestIndex]++; // Increment the height of the lowest column
31                 }
32             }
33
34             // If water has not been poured in either direction, pour it at the position
35             if (!poured) {
36                 heights[position]++;
37             }
38         }
39         return heights; // Return the modified ground after pouring all units of volume
40     }
41 }
42
```

C++ Solution

```
1  #include <vector>
2  using std::vector;
3
4  class Solution {
5  public:
6      // Method to simulate pouring water over the heights
7      vector<int> pourWater(vector<int>& heights, int volume, int index) {
8          // Loop to pour water up to the given volume
9          while (volume-- > 0) {
10             bool waterPoured = false; // Flag to check if water was poured
11
12             // Check both directions, left first (-1) then right (1)
13             for (int direction = -1; direction <= 1 && !waterPoured; direction += 2) {
14                 int currentPosition = index, lowestPosition = index;
15
16                 // Move in the current direction as long as the next position is within bounds
17                 // and the height at the next position is not greater than the current one.
18                 while (currentPosition + direction >= 0 && currentPosition + direction < heights.size() &&
19                        heights[currentPosition + direction] <= heights[currentPosition]) {
20                     // If the next position is lower, update the lowestPosition
21                     if (heights[currentPosition + direction] < heights[lowestPosition]) {
22                         lowestPosition = currentPosition + direction;
23                     }
24
25                     // Move to the next position
26                     currentPosition += direction;
27                 }
28
29                 // Pour water into the lowest position found, if it's different from the starting index
30                 if (lowestPosition != index) {
31                     waterPoured = true;  // Water was successfully poured
32                     ++heights[lowestPosition]; // Increment the height at the lowest position
33                 }
34             }
35
36             // If we couldn't pour water to the left or right, pour it at the index (starting position)
37             if (!waterPoured) {
38                 ++heights[index];
39             }
40         }
41
42         // Return the modified heights vector
43         return heights;
44     }
45 };
46
```

Typescript Solution

```
1  // Function to simulate pouring water over the heights
2  function pourWater(heights: number[], volume: number, index: number): number[] {
3      // Loop to pour water up to the given volume
4      while (volume-- > 0) {
5          let waterPoured = false; // Flag to check if water was poured
6
7          // Check both directions, left first (-1) then right (1)
8          for (let direction = -1; direction <= 1 && !waterPoured; direction += 2) {
9              let currentPosition = index;
10             let lowestPosition = index;
11
12             // Move in the current direction as long as the next position is within bounds
13             // and the height at the next position is not greater than the current one.
14             while (currentPosition + direction >= 0 &&
15                    currentPosition + direction < heights.length &&
16                    heights[currentPosition + direction] <= heights[currentPosition]) {
17                 // If the next position is lower, update the lowestPosition
18                 if (heights[currentPosition + direction] < heights[lowestPosition]) {
19                     lowestPosition = currentPosition + direction;
20                 }
21
22                 // Move to the next position
23                 currentPosition += direction;
24             }
25
26             // Pour water into the lowest position found, if it's different from the starting index
27             if (lowestPosition != index) {
28                 waterPoured = true;  // Water was successfully poured
29                 heights[lowestPosition]++;  // Increment the height at the lowest position
30             }
31         }
32
33         // If we couldn't pour water to the left or right, pour it at the index (starting position)
34         if (!waterPoured) {
35             heights[index]++;
36         }
37     }
38
39     // Return the modified heights array
40     return heights;
41 }
42
```

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

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), 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 `O(1)` operation. Therefore, when combining these operations, the overall time complexity becomes `O(V * N)`.

The space complexity of the code is `O(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.