42. Trapping Rain Water

Hard Stack Array Two Pointers Dynamic Programming Monotonic Stack

Problem Description

Imagine you have a side-view silhouette of a series of blocks with various heights. These blocks represent the elevation map in the problem. When it rains, water will fill in the gaps between these blocks. The water can be trapped between the elevations where the adjacent blocks are higher. The width of each block is `1` unit. The task is to calculate the total amount of water that these blocks can hold without spilling over, after a rainfall.

To tackle this, picture placing water on top of the blocks. Some water will be trapped between taller blocks, while some will flow off the sides if there are no sufficient barriers. The amount of trapped water at any point depends on the tallest blocks to its left and to its right because these will act as barriers. At any given point, the amount of water that can be held above it is the difference between the height of the shorter of the two barriers (either left or right) and the height of the block itself.

Intuition

The solution to this problem is based on the observation that the amount of water above a bar is determined by the highest bar to the left and the highest bar to the right. No matter what the arrangements of other bars are, the water above a given bar can never exceed the height difference between the bar and the shortest of the two highest bars flanking it.

To approach this problem efficiently, we use dynamic programming. We create two arrays, `left` and `right`, which represent the highest bars up to that point from the left and right, respectively. These arrays are populated by traversing the height array twice—once from left to right, updating the `left` array with the maximum height seen so far, and once from right to left, updating the `right` array similarly.

Once we have these two arrays, the amount of water above each bar is simply the difference between it and the minimum of the highest bars to its left and right. The solution is then to accumulate this difference for each bar to get the total amount of trapped water.

To put it more concretely, the `left[i]` array gets updated as we go, ensuring that at each step `i`, we have the height of the tallest bar up to that point. The `right[i]` array does the same but in the opposite direction. When calculating the trapped water at index `i`, we look at `min(left[i], right[i])` which gives us the maximum water level possible at that point. Then, we subtract the height of the current bar `height[i]` to know how much water can actually be trapped there. Summing these values together for all indices gives us the total amount of trapped water.

Solution Approach

The solution uses a combination of arrays and iteration to calculate the volume of trapped rainwater.

Firstly, we define two arrays, `left` and `right`:

- `left[i]` contains the height of the tallest pillar to the left of the position with index `i`, including the pillar at `i` itself.
- `right[i]` contains the height of the tallest pillar to the right of the position with index `i`, also including the pillar at `i`.

We initialize `left[0]` with the height of the first pillar `height[0]`, since there's nothing to its left. Similarly, we initialize `right[n-1]` (where `n` is the number of pillars) with the height of the last pillar `height[n-1]`.

Next, two separate loops are used to populate these arrays:

- For `left[i]`, starting from `i=1` and moving towards the end of the array, we calculate the maximum height encountered so far using `max(left[i - 1], height[i])` and store it in `left[i]`. This ensures that `left[i]` contains the height of the tallest bar to the left including the current bar.

- For `right[i]`, we move in the opposite direction, starting from `i=n-2` down to `0`. We update `right[i]` with `max(right[i + 1], height[i])`. This accomplishes the same as the `left` array but for bars to the right.

With these arrays filled, we can then iterate through each index `i` and calculate the trapped water above the bar at `i`. The amount of trapped water here is `min(left[i], right[i]) - height[i]` because it is bounded by the shortest of the two tallest pillars on either side.

Finally, we sum up the calculated trapped water values for all `i` to find the total amount of trapped rainwater. The result is computed using `sum(min(l, r) - h for l, r, h in zip(left, right, height))`, taking advantage of Python's built-in functions and list comprehensions for concise code. By zipping the `left`, `right`, and `height` arrays together, we can iterate over them simultaneously, allowing us to easily calculate the minimum of `left[i]` and `right[i]`, subtract the `height[i]`, and sum all these values.

Throughout this process, we use simple arrays and loops, a technique that falls under the category of dynamic programming, where we break down the problem into simpler subproblems and iteratively build up solutions to larger problems leveraging the solutions to smaller problems.

Example Walkthrough

Let's take a small example to illustrate the solution approach: Suppose our height map for the blocks is given by `height = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]`.

Now, follow the steps below:

Firstly, we initialize our `left` and `right` arrays:

- Initialize `left` array with the same length as `height` with `left[0] = height[0]`, which is `0`.
- Initialize `right` array with the same length as `height` with `right[-1] = height[-1]`, which is `1`.

Secondly, we populate the `left` and `right` arrays:

- Start filling the `left` array from `left[1]` to `left[-1]`. For each `left[i]`, we consider `max(left[i - 1], height[i])`. This updates `left` to `[0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3]`.
- Fill the `right` array in reverse (starting from the second to last element and moving to the first). For each `right[i]`, we consider `max(right[i + 1], height[i])`. This updates `right` to `[3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 1]`.

With both arrays filled, we now calculate the trapped water above each index `i` in the `height` array:

- For each position `i`, calculate the trapped water as `min(left[i], right[i]) - height[i]`, which gives us the volume at each position as `[0, 0, 1, 0, 1, 2, 1, 0, 0, 1, 0, 0]`.

Lastly, sum the trapped water calculated for every position:

- Calculate the sum which is `0 + 0 + 1 + 0 + 1 + 2 + 1 + 0 + 0 + 1 + 0 + 0`, giving us a total of `6` units of water that the blocks can hold after rainfall.

By following the described solution approach step by step, we easily solve the problem using dynamic programming concepts and determine that our example silhouette can trap a total of 6 units of water.

Python Solution

```python
from typing import List

class Solution:
    def trap(self, height: List[int]) -> int:
        # Find the number of elements in height.
        num_elements = len(height)

        # Initialize arrays to store the maximum to the left and right of each element.
        max_left = [height[0]] * num_elements
        max_right = [height[-1]] * num_elements

        # Fill the max_left array with the maximum height to the left of each element.
        for i in range(1, num_elements):
            max_left[i] = max(max_left[i - 1], height[i])

        # Fill the max_right array with the maximum height to the right of each element.
        for i in range(num_elements - 2, -1, -1):
            max_right[i] = max(max_right[i + 1], height[i])

        # Calculate the total amount of trapped water using the height difference
        # between the minimum of max_left and max_right for each element and the height at that element.
        trapped_water = sum(min(max_left[i], max_right[i]) - height[i] for i in range(num_elements))

        # Return the total amount of trapped water.
        return trapped_water

```

Java Solution

```java
class Solution {
    public int trap(int[] height) {
        // The length of the given height array.
        int length = height.length;

        // Arrays to store the maximum height to the left and right of every bar.
        int[] maxLeft = new int[length];
        int[] maxRight = new int[length];

        // Initialize the first element of maxLeft with the first height
        // as there's nothing to the left of it.
        maxLeft[0] = height[0];
        // Initialize the last element of maxRight with the last height
        // as there's nothing to the right of it.
        maxRight[length - 1] = height[length - 1];

        // Populate the maxLeft array by finding the maximum height to the left
        // of the current position, including itself.
        for (int i = 1; i < length; ++i) {
            maxLeft[i] = Math.max(maxLeft[i - 1], height[i]);
        }

        // Populate the maxRight array by finding the maximum height to the right
        // of the current position, including itself. This loop runs backwards.
        for (int i = length - 2; i >= 0; --i) {
            maxRight[i] = Math.max(maxRight[i + 1], height[i]);
        }

        // Variable to store the total amount of trapped water.
        int totalWater = 0;

        // Calculate the trapped water at each position by finding the
        // minimum of maxLeft and maxRight at that position (which is the maximum
        // water level the position can hold) and subtracting the height of the bar.
        for (int i = 0; i < length; ++i) {
            totalWater += Math.min(maxLeft[i], maxRight[i]) - height[i];
        }

        // Return the total trapped water.
        return totalWater;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    int trap(vector<int>& height) {
        int num_elements = height.size(); // Number of elements in the height vector

        // Handling edge case: if the height vector is empty or has only one bar,
        // no water can be trapped.
        if(num_elements < 2) {
            return 0;
        }

        vector<int> left_max(num_elements), right_max(num_elements); // Initialize vectors to store the max height to the left and ri

        // Base case: The first element's left max will be itself and
        // the last element's right max will be itself
        left_max[0] = height[0];
        right_max[num_elements - 1] = height[num_elements - 1];

        // Fill left_max array such that left_max[i] contains highest bar height to the left of index 'i' including itself
        for (int i = 1; i < num_elements; ++i) {
            left_max[i] = max(left_max[i - 1], height[i]);
        }

        // Fill right_max array such that right_max[i] contains highest bar height to the right of index 'i' including itself
        for (int i = num_elements - 2; i >= 0; --i) {
            right_max[i] = max(right_max[i + 1], height[i]);
        }

        int total_water = 0; // Initialize total water to be trapped

        // Calculate trapped water at each index 'i' by finding the
        // smaller of the left and right max bars and subtracting the current height.
        // The result is added to total_water to find the cumulative water trapped across the entire array.
        for (int i = 0; i < num_elements; ++i) {
            total_water += min(left_max[i], right_max[i]) - height[i];
        }

        // Return the total amount of trapped water
        return total_water;
    }
};

```

Typescript Solution

```typescript
function trap(heights: number[]): number {
    const numElements = heights.length;
    // Initialize arrays to store the maximum height to the left/right of each index
    const maxLeftHeights: number[] = new Array(numElements).fill(heights[0]);
    const maxRightHeights: number[] = new Array(numElements).fill(heights[numElements - 1]);

    // Populate the maxLeftHeights array with the maximum heights to the left of each index
    for (let i = 1; i < numElements; ++i) {
        maxLeftHeights[i] = Math.max(maxLeftHeights[i - 1], heights[i]);
    }

    // Populate the maxRightHeights array with the maximum heights to the right of each index
    for (let i = numElements - 2; i >= 0; --i) {
        maxRightHeights[i] = Math.max(maxRightHeights[i + 1], heights[i]);
    }

    // Calculate the total amount of trapped water
    let totalWaterTrapped = 0;
    for (let i = 0; i < numElements; ++i) {
        // The water level at the current index is the minimum of the max heights to the left and right
        // Subtract the height of the current bar to get the water trapped at this index
        totalWaterTrapped += Math.min(maxLeftHeights[i], maxRightHeights[i]) - heights[i];
    }

    return totalWaterTrapped;
}

```

Time and Space Complexity

The provided code implements a solution to calculate the amount of water that can be trapped between the bars of different heights represented by the `height` list.

Time complexity: The time complexity of the solution is `O(n)` where `n` is the number of elements in the `height` list. The reasoning behind this time complexity is as follows:

- Creating the `left` and `right` lists takes `O(n)` each as they are initialized based on the first and last element respectively.
- Populating the `left` and `right` lists with the maximum height encountered so far from the left and right involves a single pass through the `height` list from left to right and right to left, which again takes `O(n)` time each.
- Finally, the `sum(min(l, r) - h for l, r, h in zip(left, right, height))` computation is also linear, as it involves a single pass through the zipped lists, for a total of `O(n)` time.

Overall, as all these steps are sequential and each of them takes `O(n)` time, the total time complexity is `O(n)`.

Space complexity: The space complexity of the solution is also `O(n)`. This is because additional space is used to store the `left` and `right` lists which both have the same length as the input list `height`. No other significant storage is used that depends on `n` (the length of the `height` list), therefore, the space complexity is `O(n)`.

To summarize, the code provided efficiently computes the water trapping problem with a linear time complexity and uses linear space to store interim results.