11. Container With Most Water

Medium Greedy Array Two Pointers

Problem Description

You are presented with an integer array called `height` which represents the heights of vertical lines placed at positions indexed from 1 to n (0-indexed in the array). Imagine that each `height[i]` is linked to a line on a chart that extends upward from the x-axis to a point `(i, height[i])`. Your task is to find two lines that, along with the x-axis, enclose the greatest possible area, which represents the maximum water that can be trapped between them without allowing any spillage over the sides of the lines (the container cannot be slanted). The goal is to calculate and return this maximum trapped water area.

Intuition

To solve this problem efficiently, we use a two-pointer technique. We place one pointer at the beginning of the array and the other at the end, and these pointers represent the potential container boundaries. At each step, the trapped water is determined by the distance between the pointers (which is the container's width) and the height of the smaller line (since the water level can't be higher than the smaller of the two boundaries). This is the area that could potentially be the maximum.

To maximize the area, after calculating the trapped water at each step and comparing it to the maximum we've seen so far, we move the pointer at the shorter line towards the other pointer. This is because keeping the pointer at the taller line stationary and moving the shorter one might lead us to find a taller line and thus a larger area. There's no advantage in moving the taller pointer first, as it would only reduce the potential width without guaranteeing a taller line to increase height. We repeat this process of calculating, updating the maximum water area, and moving the shorter line pointer towards the other pointer until the two pointers meet, at which point we've considered every possible container and the maximum stored water has been found.

By approaching this problem with each step optimized to either maintain or improve the potential maximum area, we are able to arrive at the solution efficiently, resulting in an algorithm that runs in linear time relative to the number of lines.

Solution Approach

The implementation of the solution follows the two-pointer approach. Here's a step-by-step guide to how the solution works:

1. Initialize two pointers: `i` is set to the start of the array (`0`), and `j` is set to the end of the array (`len(height) - 1`).
2. Initialize a variable `ans` to keep track of the maximum area discovered so far. Initially, `ans` is set to `0`.
3. Enter a loop that continues as long as `i` is less than `j`. This loop allows us to explore all possible combinations of lines from `i` to `j` to maximize the area.
4. Inside the loop, calculate the area trapped between the lines at pointers `i` and `j` using the formula: `area = (j - i) * min(height[i], height[j])`. This calculates the width of the container (`j - i`) and multiplies it by the height, which is the smaller of the two heights at `height[i]` and `height[j]`.
5. Update `ans` with the maximum of its current value and the calculated area. `ans = max(ans, area)` ensures that `ans` holds the highest value of trapped water area at each step.
6. Determine which pointer to move. We need to move the pointer corresponding to the shorter line since this is the limiting factor for the height of the trapped water. We do this using a conditional statement:
   - If `height[i] < height[j]`, then we increment `i` (`i += 1`) to potentially find a taller line.
   - Else, decrement `j` (`j -= 1`) for the same reason from the other end.
7. Continue looping until the pointers meet. At this point, `ans` would have the maximum area that can be trapped between any two lines.

This solution uses a greedy approach, and its efficiency stems from the fact that at each stage, the move made is the best possible move to increase or at least maintain the potential of the maximum area. By incrementally adjusting the width and height of the considered container, it efficiently narrows down to the optimal solution.

The algorithm has a linear-time complexity, O(n), as each element is visited at most once, and there's a constant amount of work done within the loop.

Example Walkthrough

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

Consider the integer array `height = [1, 8, 6, 2, 5, 4, 8, 3, 7]`.

1. We start with two pointers: `i` at the start (`0`), representing the first line, and `j` at the end (`8`), representing the last line. Thus, `i` points to `height[0]` which is `1`, and `j` points to `height[8]` which is `7`.
2. We set `ans = 0`, as we have not calculated any area yet.
3. Now we start our loop where `i < j`. Since `0 < 8`, we enter the loop.
4. We calculate the area between lines at pointers `i` and `j`. The width is `j - i` which is `8 - 0 = 8`, and the height is the smaller of two heights at `height[i]` and `height[j]`, so `min(1, 7) = 1`. Thus, the area is `8 * 1 = 8`.
5. We update `ans` to be the maximum of its current value and the calculated area. So, `ans = max(0, 8) = 8`.
6. Since `height[i]` is less than `height[j]`, we move the `i` pointer to the right to potentially find a taller line. Now `i` becomes `1`.
7. Our two pointers now are at `i = 1` and `j = 8`. We will continue this process until `i` and `j` meet.
8. Repeat steps 4-6:
   - New area at pointers `i = 1` and `j = 8`: `area = (8 - 1) * min(8, 7) = 7 * 7 = 49`.
   - Update `ans` to be `max(8, 49) = 49`.
   - Since `height[1]` is greater than `height[8]`, we move `j` to the left (now `j` is `7`).
9. Continue iterations:
   - New area at pointers `i = 1` and `j = 7`: `area = (7 - 1) * min(8, 3) = 6 * 3 = 18`.
   - `ans` remains `49` since `49 > 18`.
   - `height[1]` is greater than `height[7]`, so we move `j` to the left (now `j` is `6`).
10. The process continues in this manner, always moving the pointer at the shorter height until `i` and `j` are equal.

At the end of these iterations, `ans` holds the maximum area that can be trapped, which in this example, is `49`. This is the largest amount of water that can be trapped between two lines without spilling.

Solution Implementation

Python

~~~~python
class Solution:
    def maxArea(self, height: List[int]) -> int:
        # Initialize two pointers, one at the beginning and one at the end of the height array
        left, right = 0, len(height) - 1
        # Initialize maximum area to 0
        max_area = 0

        # Use a while loop to iterate until the two pointers meet
        while left < right:
            # Calculate the area formed between the two pointers
            current_area = (right - left) * min(height[left], height[right])
            # Update the maximum area if current area is larger
            max_area = max(max_area, current_area)

            # Move the pointer that points to the shorter line inward,
            # since this might lead to a greater area
            if height[left] < height[right]:
                left += 1
            else:
                right -= 1

        # Return the maximum area found
        return max_area
```

Please note that the type hint `List[int]` requires importing `List` from the `typing` module in Python 3.5+. If you're using Python

```python
from typing import List  # This line is needed for the type hint (Python 3.5 - 3.8)

class Solution:
    # ... rest of the code remains the same
~~~~

Java

~~~~java
class Solution {

    // Method to find the maximum area formed between the vertical lines
    public int maxArea(int[] height) {
        // Initialize two pointers at the beginning and end of the array
        int left = 0;
        int right = height.length - 1;
        // Variable to keep track of the maximum area
        int maxArea = 0;

        // Iterate until the two pointers meet
        while (left < right) {
            // Calculate the area with the shorter line as the height and the distance between the lines as the width
            int currentArea = Math.min(height[left], height[right]) * (right - left);
            // Update the maximum area if the current area is larger
            maxArea = Math.max(maxArea, currentArea);

            // Move the pointer that points to the shorter line towards the center
            if (height[left] < height[right]) {
                left++;
            } else {
                right--;
            }
        }

        // Return the maximum area found
        return maxArea;
    }
}
~~~~

C++

~~~~cpp
#include <vector>
#include <algorithm> // Include algorithm for std::min and std::max

class Solution {
public:
    int maxArea(vector<int>& heights) {
        int left = 0; // Starting from the leftmost index
        int right = heights.size() - 1; // Starting from the rightmost index
        int maxArea = 0; // Initialize the maximum area to 0

        // Continue looping until the left and right pointers meet
        while (left < right) {
            // Calculate the current area with the minimum of the two heights
            int currentArea = std::min(heights[left], heights[right]) * (right - left);
            // Update the maximum area if the current area is larger
            maxArea = std::max(maxArea, currentArea);

            // Move the pointers inward. If left height is less than right height
            // then we move the left pointer to right hoping to find a greater height
            if (heights[left] < heights[right]) {
                ++left;
            } else { // Otherwise, move the right pointer to the left
                --right;
            }
        }

        return maxArea; // Return the maximum area found
    }
};
~~~~

TypeScript

~~~~typescript
function maxArea(height: number[]): number {
    // Initialize two pointers, one at the start and one at the end of the array
    let leftIndex = 0;
    let rightIndex = height.length - 1;
    // Initialize the variable to store the maximum area
    let maxArea = 0;

    // Iterate until the two pointers meet
    while (leftIndex < rightIndex) {
        // Calculate the area with the current pair of lines
        const currentArea = Math.min(height[leftIndex], height[rightIndex]) * (rightIndex - leftIndex);
        // Update maxArea if the current area is larger
        maxArea = Math.max(maxArea, currentArea);

        // Move the pointer that's at the shorter line inwards
        // If the left line is shorter than the right line
        if (height[leftIndex] < height[rightIndex]) {
            ++leftIndex; // Move the left pointer to the right
        } else {
            --rightIndex; // Move the right pointer to the left
        }
    }

    // Return the maximum area found
    return maxArea;
}
~~~~

~~~~python
class Solution:
    def maxArea(self, height: List[int]) -> int:
        # Initialize two pointers, one at the beginning and one at the end of the height array
        left, right = 0, len(height) - 1
        # Initialize maximum area to 0
        max_area = 0

        # Use a while loop to iterate until the two pointers meet
        while left < right:
            # Calculate the area formed between the two pointers
            current_area = (right - left) * min(height[left], height[right])
            # Update the maximum area if current area is larger
            max_area = max(max_area, current_area)

            # Move the pointer that points to the shorter line inward,
            # since this might lead to a greater area
            if height[left] < height[right]:
                left += 1
            else:
                right -= 1

        # Return the maximum area found
        return max_area
```

Please note that the type hint `List[int]` requires importing `List` from the `typing` module in Python 3.5+. If you're using Python

```python
from typing import List  # This line is needed for the type hint (Python 3.5 - 3.8)

class Solution:
    # ... rest of the code remains the same
~~~~

Time and Space Complexity

The given Python code implements a two-pointer technique to find the maximum area of water that can be contained between two lines, given an array of line heights.

Time Complexity

The function initializes two pointers at the start and end of the array respectively and iterates inwards until they meet, performing a constant number of operations for each pair of indices. Since the pointers cover each element at most once, the iteration is linear relative to the number of elements `n` in the `height` array.

Hence, the time complexity is `O(n)`.

Space Complexity

The code uses a fixed number of integer variables (`i`, `j`, `ans`, and `t`) and does not allocate any additional memory that scales with the size of the input array.

Thus, the space complexity is `O(1)`.