84. Largest Rectangle in Histogram

Hard Stack Array Monotonic Stack

Leetcode Link

Problem Description

The task is to find the largest rectangular area in a histogram, which consists of a series of adjacent bars of varying heights. Each bar has a uniform width of 1 unit, and the height of each bar corresponds to each integer in the given `heights` array. To elaborate, imagine drawing rectangles starting from the base of each bar and extending them as wide as possible without crossing the height of shorter bars adjacent to it. The goal is to determine the maximum area covered by such a rectangle.

Intuition

The intuition behind the solution involves using a stack and a concept called "Monotonic Stack" which helps us process the bars in a way that we can calculate the maximum width for each bar where it remains the tallest bar within that width.

Here is the reasoning for each step in the algorithm:

- We initialize `left` and `right` arrays to keep track of the bounds for the largest rectangle with `height[i]` as the smallest bar. `left[i]` will store the index of the first bar to the left that is shorter than `height[i]`, and `right[i]` will store the index of the first bar to the right that is shorter than `height[i]`.
- We traverse the heights while using a stack `stk` to maintain indices of bars in a non-decreasing order. The stack will help us in finding the left and right bounds for each bar.
- For each bar `h` with index `i`:
  - We pop from the stack while the bar at the top of the stack has a height greater than or equal to `h`. Each time we pop, we update the `right` bound for the popped index because we just found a shorter bar on the right.
  - If the stack is not empty after popping, it means the bar on the top of the stack is the first bar to the left of `h` that is shorter, so we update the `left[i]` to that index.
  - We then push `i` onto the stack, since it might be the potential left bound for a future bar.
- After we have the left and right bounds for each bar, the maximum possible width for each bar is given as `(right[i] - left[i] - 1)`. We multiply this with the height `heights[i]` of the bar to get the area of the largest rectangle with the bar at `i` being the shortest bar. We calculate this for every bar.
- Finally, we return the maximum area out of all the areas we have computed.

This algorithm is efficient because each bar is pushed and popped from the stack exactly once, and the left and right bounds are updated during this process. We're leveraging the stack to perform all necessary computations as we iterate through the histogram, which allows us to compute the largest rectangle in linear time.

Solution Approach

The solution is implemented in Python using a single-pass algorithm with a stack data structure to maintain a history of bars that are yet to find a shorter bar on the right. The goal is to calculate the correct `left` and `right` bounds for each bar, which will then allow us to calculate the area of the largest rectangle that can be formed with each bar as the shortest one in that rectangle.

Here's a step-by-step breakdown of the algorithm:

1. **Initializing Data Structures:** A stack `stk` is initialized as an empty list, which will store indices of bars. Two lists `left` and `right` are also initialized to store the left and right bounds for each bar (`-1` for `left` and `n` for `right`, respectively, where `n` is the number of bars).

2. **Traversing Heights:** We loop through each bar using its index `i` and height `h`:
   - While the top of the stack is not empty and the current height `h` is less than or equal to the height at the index on top of the stack, we pop from the stack. This indicates that we have found a right boundary for the rectangle concerning the bar at the index that was just popped.
   - For each index we pop from the stack, we use it to set the corresponding `right` bound for that bar to the current index (`i`), since we now know that this is the first bar to the right that is shorter.
   - If there are still elements left in the stack after popping, it indicates that the current bar `h` is larger than the bar at the top of the stack. Hence, the `left` bounding for the current bar `h` is now known to be the index on top of the stack.
   - Finally, we add the current index `i` to the top of the stack.

3. **Calculating the Largest Area:** Once the traversal is done, we have `left` and `right` arrays filled with the bounds of potential rectangles for each bar. To calculate the largest area:
   - Iterate over each index `i` and corresponding height `v`, and calculate the width for the largest rectangle that can be formed with height `v` by subtracting `left[i]` from `right[i]` and subtracting `1` (as the boundaries are exclusive).
   - Calculate the area for each rectangle by multiplying its height (`heights[i]`) with the width calculated above.
   - Use the `max` function to determine the largest area from these.

4. **Return the Largest Area:** After iteration, return the maximum area calculated, which is the area of the largest rectangle in the histogram.

The code snippet provided uses list comprehension at the end to perform the calculation of the areas and find the maximum all in one line, showcasing an efficient Pythonic approach:

```
return max(h * (right[i] - left[i] - 1) for i, h in enumerate(heights))
```

This algorithm uses a stack efficiently that follows the Last-In-First-Out (LIFO) principle, whereby each element is pushed and popped only once, resulting in a time complexity of $O(n)$ where n is the number of bars in the histogram.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the `heights` array:

```
1 heights = [2, 1, 5, 6, 2, 3]
```

This represents a histogram with 6 bars where the heights of the bars are `[2, 1, 5, 6, 2, 3]`.

1. **Initializing Data Structures:**

```
1 stk = []
2 left = [-1] * len(heights)
3 right = [len(heights)] * len(heights)
```

2. **Traversing Heights:** We start traversing the `heights` array.

   - For `heights[0]`, since the stack is empty, no action is taken other than pushing the index onto the stack:

     ```
     stk = [0]
     ```

   - For `heights[1]`, the current height `1` is less than the height at the top of the stack `2`. We pop from the stack (popping `0`):

     ```
     stk = []
     ```

     We update `right[0] = 1` and since `stk` is now empty, we push the current index `1` onto the stack:

     ```
     stk = [1]
     ```

   - For `heights[2]` (`5`), the stack action yields:

     ```
     stk = [1, 2]
     ```

   - For `heights[3]` (`6`):

     ```
     stk = [1, 2, 3]
     ```

   - For `heights[4]` (`2`), we start popping since `2` is less than `6` and `5`. After popping `3` and `2`:

     ```
     stk = [1]
     ```

     We update `right[3] = 4`, `right[2] = 4`. The stack still has `1` in it, so `left[4] = 1`. Push `4` onto the stack:

     ```
     stk = [1, 4]
     ```

   - For `heights[5]` (`3`):

     ```
     stk = [1, 4, 5]
     ```

3. **Calculating the Largest Area:**

   - We've completed the traversal, `left = [-1, -1, 1, 2, 1, 4]` and `right = [6, 2, 4, 4, 6, 6]`.
   - Calculate the widths and areas for each bar; for example:
     - For `i=2` (`height = 5`):

       ```
       width = right[2] - left[2] - 1 = 4 - 1 - 1 = 2

       area = heights[2] * width = 5 * 2 = 10
       ```

     - Repeat this for each bar and find the max area:

       ```
       areas = [2 * (1 - (-1) - 1), 1 * (6 - (-1) - 1), ... ]
       ```

       After evaluating all areas, the largest one is `10` (for `heights[2]`).

4. **Return the Largest Area:**
   - The maximum calculated area is `10`, which is the area of the largest rectangle in the histogram we started with.

Python Solution

```
from typing import List

class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        # Get the total number of bars in the histogram
        num_bars = len(heights)

        # Initialize stacks for indexes of bars
        stack = []

        # Initialize arrays to record the first smaller bar on the left of each bar
        smaller_left_index = [-1] * num_bars

        # Initialize arrays to record the first smaller bar on the right of each bar
        smaller_right_index = [num_bars] * num_bars

        # Iterate over all heights to compute the smaller_left_index and smaller_right_index
        for index, height in enumerate(heights):
            # Pop elements from the stack while the current height is less than
            # the top element's height in the stack to find the right boundary
            while stack and heights[stack[-1]] >= height:
                smaller_right_index[stack[-1]] = index
                stack.pop()
            # If the stack is not empty, the current element at the top is the previous
            # bar of smaller height (left boundary)
            if stack:
                smaller_left_index[index] = stack[-1]
            # Push this bar onto stack
            stack.append(index)

        # Calculate the maximum area of rectangle in histogram
        max_area = 0
        for a, b in enumerate(heights):
            # Update max_area with the larger area found
            max_area = max(max_area, b * (smaller_right_index[a] - smaller_left_index[a] - 1))

        return max_area
```

Java Solution

```
class Solution {
    public int largestRectangleArea(int[] heights) {
        int maxArea = 0; // This variable will store the maximum area found.
        int length = heights.length; // Total number of bars.

        // Stack to keep track of indices of the bars.
        Deque<Integer> stack = new ArrayDeque<>();

        // Arrays to keep track of the left and right boundaries of each bar.
        int[] leftBoundary = new int[length];
        int[] rightBoundary = new int[length];

        // Initialize right boundaries as the length of the array.
        Arrays.fill(rightBoundary, length);

        // Iterate over all bars to calculate left and right boundaries.
        for (int i = 0; i < length; ++i) {
            // Pop elements from the stack until the current bar is taller than the stack's top
            // and set their right boundary to the current bar's index.
            while (!stack.isEmpty() && heights[stack.peek()] >= heights[i]) {
                rightBoundary[stack.pop()] = i;
            }
            // If the stack is empty, then there's no smaller bar to the left,
            // so the left boundary is -1. Otherwise, it's the stack's top index.
            leftBoundary[i] = stack.isEmpty() ? -1 : stack.peek();
            // Push the current index onto the stack.
            stack.push(i);
        }

        // Calculate the largest rectangle area for each bar using their boundaries.
        for (int i = 0; i < length; ++i) {
            // Calculate width of the current bar's largest rectangle.
            int width = rightBoundary[i] - leftBoundary[i] - 1;
            // Calculate area and update maxArea if it's larger.
            maxArea = Math.max(maxArea, heights[i] * width);
        }

        // Return the maximum area found.
        return maxArea;
    }
}
```

C++ Solution

```
#include <vector>
#include <stack>
using namespace std;

class Solution {
public:
    int largestRectangleArea(vector<int>& heights) {
        // The maximum area found so far.
        int maxArea = 0;
        int n = heights.size(); // Number of bars.
        // `numBars` is heights.size();
        // Stack to keep track of the indices of the bars.
        stack<int> indexStack;
        // Vector to store the index of the left nearest smaller bar for each bar.
        vector<int> leftNearestSmaller(numBars, -1);
        // Vector to store the index of the right nearest smaller bar for each bar.
        vector<int> rightNearestSmaller(numBars, numBars);

        for (int i = 0; i < numBars; ++i) {
            // Pop elements from the stack until the current bar is taller than the stack's top bar.
            while (!indexStack.empty() && heights[indexStack.top()] >= heights[i]) {
                // Set the right nearest smaller bar for the popped bar.
                rightNearestSmaller[indexStack.top()] = i;
                indexStack.pop();
            }
            // If the stack is not empty, then the current top is the left nearest smaller bar.
            if (!indexStack.empty()) leftNearestSmaller[i] = indexStack.top();
            // Push current bar to stack.
            indexStack.push(i);
        }

        // Calculate the maximum area for each bar considering the nearest smaller bars to the left and right.
        for (int i = 0; i < numBars; ++i) {
            maxArea = max(maxArea, heights[i] * (rightNearestSmaller[i] - leftNearestSmaller[i] - 1));
        }

        // Return the maximum area found.
        return maxArea;
    }
};
```

Typescript Solution

```
// TypeScript doesn't have predefined Stack class, so we use an array to simulate stack behavior.
let indexOfTop: number[] = [];
let maxArea: number = 0;
let n: number = numbers.length;

// Variables leftNearest and rightNearest are declared globally to be used in functions below.
let leftNearest: number[];
let rightNearest: number[];

// Finds the largest rectangle area in a histogram.
function largestRectangleArea(numbers: number[]): number {
    heights = [...numbers]; // Initialize global heights variable with the input.
    indexStack = []; // Reset the stack for the new calculation.
    maxArea = 0; // Reset the maximum area for the new calculation.
    n = heights.length; // Update the length.
    leftNearest = new Array(numbers.fill(-1));
    rightNearest = new Array(numbers.fill(numbers));

    // Fill in values for leftNearest and rightNearest arrays.
    populateNearestSmallerIndices(numbers);

    // Calculate the maximum area for each bar considering the nearest smaller bars to the left and right.
    for (let i = 0; i < n; ++i) {
        maxArea = Math.max(maxArea, heights[i] * (rightNearest[i] - leftNearest[i] - 1));
    }

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

// Populates the nearest smaller indices on both left and right for each bar
function populateNearestSmallerIndices(numbers: number) {
    for (let i = 0; i < numbers; ++i) {
        // Pop elements from the stack until the current bar is taller than the stack's top bar.
        while (indexStack.length > 0 && heights[indexStack[indexStack.length - 1]] >= heights[i]) {
            // Set the right nearest smaller bar for the popped bar.
            rightNearest[indexStack.pop()] = numbers[i];
        }
        // If the stack is not empty, then the current top is the left nearest smaller bar.
        if (indexStack.length > 0) leftNearest[i] = indexStack[indexStack.length - 1];
        // Push current bar to stack.
        indexStack.push(i);
    }
}
```

Time and Space Complexity

The time complexity of the `largestRectangleArea` function can be analyzed by looking at the operations performed inside the function.

- The function initializes three lists named `left`, `right`, and `stk`, and also iterates over the input `heights` list twice (once for populating the `left` and `right` lists, and once for calculating the maximum area). Each element of the `heights` list is processed exactly once during these iterations, leading to $O(n)$ time for each loop.
- The stack `stk` is used to keep track of the indices of the rectangles in ascending order of their heights. For each element in `heights`, the stack may perform a push operation (`stk.append(i)`). Additionally, while the current height is less than or equal to the height of the rectangle corresponding to the index at the top of the stack, pop operations occur, and the `right` bound for the rectangle is updated. Despite this, each element is added to the stack once and removed from the stack at most once over the entire run of the loop, leading to a total of $O(n)$ operations.

Based on these points, the overall time complexity of the `largestRectangleArea` function is $O(n)$.

As for the space complexity:

- The extra space is used for the `stk`, `left`, and `right` lists, each of size n, where n is the number of elements in the input `heights` list. Thus, the space complexity is $O(n)$, correlating with the size of the input.