# PrepLetter: Cracking Stack-Based "Area" Problems – The Monotonic Stack Pattern

Hi there, Prepper!

Today we're going deep into an interview favorite: stack-based problems that find maximum areas or volumes in histograms and grids. You'll see this pattern pop up in some of the most notorious coding interview questions. The core technique you'll master today? The **Monotonic Stack**!

Let's get started.

## Topic Introduction

The **monotonic stack** is a specialized stack that keeps its elements in strictly increasing or decreasing order (often by value or index). This property helps us efficiently find the "next greater" or "next smaller" element for each item in a sequence. This is especially handy when dealing with *histogram bars* or *matrix columns* where you need to calculate areas or trapped water.

**Why use it?**
In problems where you need to quickly find the nearest taller or shorter bar for every bar in a histogram, a monotonic stack gets the job done in linear time, avoiding the brute-force pitfalls of nested loops.

**How does it work?**
• As you move through the sequence, you push indices onto the stack.
• Whenever you encounter a "break" in the monotonic property (like a lower bar in a histogram), you pop from the stack and calculate whatever you need (area, water, etc).
• This way, each element is pushed and popped at most once.

**Simple Example:**
Given an array of heights `[2,1,5,6,2,3]`, a monotonic stack can help you find, for each bar, the first smaller bar to the left and right.

Let's see this in action, starting from the classic histogram and ramping up!

## Problem 1: Largest Rectangle in Histogram

[LeetCode #84: Largest Rectangle in Histogram]

**Problem Statement (in plain English):**
Given an array of non-negative integers representing the heights of bars in a histogram (width of each bar is 1), find the area of the largest rectangle that can be formed within the histogram.

### Walkthrough Example

**Input:** `[2,1,5,6,2,3]`

**Output:** `10`

**Why?**

If you take the bars at heights 5 and 6, you can form a rectangle of width 2 and height 5 (the shorter of the two), so area = 2 x 5 = 10.

Try another example yourself:

**Input:** `[2,4]`

What’s the largest rectangle? (Hint: Try all possible bars and widths!)

**Take a moment to try solving this on your own before reading the solution.**

## Solution: The Monotonic Stack Approach

This problem is a classic use-case for the monotonic increasing stack. Here’s the step-by-step logic:

- Use a stack to store indices of bars.
- For each bar, if it’s taller than the bar at the stack's top, push its index.
- If it's shorter, pop from the stack and calculate the area considering the popped bar as the smallest bar. Do this until the stack is empty or the current bar is taller.
- To handle all bars, we append a zero-height bar at the end.

**Why does this work?**

Whenever we pop a bar, we know it can’t be extended further right, so it’s time to calculate its "max rectangle".

## Python Code

```python
def largestRectangleArea(heights):
    stack = []  # Will store indices
    max_area = 0
    heights.append(0)  # Sentinel to empty out the stack at the end

    for i, h in enumerate(heights):
        # While current bar is lower than the last bar in the stack
        while stack and heights[stack[-1]] > h:
            height = heights[stack.pop()]
            # Width is from last item in stack (if any) +1 up to i
            width = i if not stack else i - stack[-1] - 1
            max_area = max(max_area, height * width)
        stack.append(i)
    return max_area
```

**Time Complexity:** O(N) – each bar is pushed and popped at most once.

**Space Complexity:** O(N) – for the stack.

## Code Explanation

- We iterate through all bars, including the extra zero at the end.
- If the stack is empty or the current bar is higher, we push.
- If the current bar is lower, we pop (meaning the popped bar's rectangle can't extend further).
- Width is calculated as the distance between the current index and the previous lower bar (from the stack).
- We always keep track of the largest area found.

**Test this code yourself with this example:**
`heights = [2,1,2]`
Manually trace the stack and area calculations!

**Did you know?**
You could also solve this problem using **divide and conquer** or **segment trees** for range minima, but monotonic stack is by far the most efficient and intuitive for this scenario. Try reading about those after you're done!

Let's level up!

## Problem 2: Maximal Rectangle

LeetCode #85: Maximal Rectangle

**Problem Statement (in plain English):**
Given a 2D binary matrix filled with '0's and '1's, find the largest rectangle containing only '1's and return its area.

**How is this different?**
It's the histogram problem, but now in 2D! Each row can be seen as a base of a histogram, where each column's height is incremented if it's a '1', or reset if it's a '0'.

### Walkthrough Example

**Input:**

```
[
  ["1","0","1","0","0"],
  ["1","0","1","1","1"],
  ["1","1","1","1","1"],
  ["1","0","0","1","0"]
]
```

**Output:** `6`

**Why?**
The maximal rectangle is in the 3rd row, spanning columns 2 to 5 with height 2 (i.e., a rectangle of 2 rows x 3 columns).

Try this small matrix:

```
[
  ["1","1"],
  ["1","1"]
]
```

What's the maximal rectangle area?

**Take a moment to try solving this on your own before reading the solution.**

## Solution: Build Histograms Row-by-Row + Monotonic Stack

We treat each row as the base of a histogram. For each row, we build a "heights" array, where each value is the number of consecutive '1's above (including current row). Then, for each row's histogram, run the largest rectangle code from Problem 1.

**Step by Step:**

- Initialize a heights array of zeros.
- For each row:
    - Update heights: add 1 if current cell is '1', else reset to 0.
    - Run largestRectangleArea(heights) to get the max area for this row.
    - Update the global max area.

## Python Code

```python
def maximalRectangle(matrix):
    if not matrix or not matrix[0]:
        return 0
    n_cols = len(matrix[0])
    heights = [0] * n_cols
    max_area = 0

    for row in matrix:
        for i in range(n_cols):
            # If '1', increment height; else reset to 0
            heights[i] = heights[i] + 1 if row[i] == '1' else 0
        # Use the histogram solution from Problem 1
        area = largestRectangleArea(heights)
        max_area = max(max_area, area)
    return max_area
```

**Time Complexity:** O(rows * cols) – each cell is visited once, and each row's histogram is processed in O(cols).
**Space Complexity:** O(cols) for the heights array and stack.

## Code Explanation

- For each row, we're building a histogram on the fly.
- After updating the heights, we reuse the exact logic from Problem 1.
- This shows the direct connection: the 2D maximal rectangle reduces to a series of 1D largest rectangle in histogram problems!

**Try this yourself:**

```
[
  ["0","1","1","0"],
  ["1","1","1","1"]
]
```

What’s the maximal rectangle area? How do the heights look after each row?

**Did you guess?**
This problem can also be solved with dynamic programming by maintaining left/right boundaries of rectangles. But the histogram + monotonic stack method, as you just learned, is more intuitive if you’ve nailed the pattern.

Let’s take a twist and see how stacks help us capture not areas, but volumes!

## Problem 3: Trapping Rain Water

LeetCode #42: Trapping Rain Water

**Problem Statement (in plain English):**
Given an array representing the elevation map where the width of each bar is 1, compute how much water it can trap after raining.

**How is this different?**
Instead of looking for the largest rectangle, you’re looking to trap water between the bars. But the stack pattern still applies — you look for the next higher bar to form a "container".

### Walkthrough Example

**Input:** `[0,1,0,2,1,0,1,3,2,1,2,1]`
**Output:** `6`

**Why?**
Water is trapped in valleys. For example, between the bars at positions 1 and 3, you can trap 1 unit; between 3 and 7, more water can be trapped. Add up all the trapped water.

Try another example for practice:
**Input:** `[4,2,0,3,2,5]`
How much water is trapped?

**Take a moment to try solving this on your own before reading the solution.**

### Solution: Monotonic Stack for Water Trapping

Here's the intuition:

- Use a stack to track bars that could form the left boundary of a water container.
- When you find a bar taller than the top of the stack, pop the stack and calculate the water trapped above the popped bar. The water volume is determined by the distance between the current bar (right boundary) and the new top of the stack (left boundary), and the height difference.

## Python Code

```python
def trap(height):
    stack = []
    water = 0
    for i, h in enumerate(height):
        # Found a right boundary
        while stack and h > height[stack[-1]]:
            top = stack.pop()
            if not stack:
                break  # No left boundary
            # Distance between current and new top
            distance = i - stack[-1] - 1
            # Water height is min(left, right) - current
            bounded_height = min(h, height[stack[-1]]) - height[top]
            water += distance * bounded_height
        stack.append(i)
    return water
```

**Time Complexity:** O(N) – each bar is pushed and popped at most once.
**Space Complexity:** O(N) – for the stack.

## Code Explanation

- Traverse the array, using a stack to track indices of bars.
- When you find a bar taller than the one at the top of the stack, you start popping and calculate trapped water.
- The trapped water is between the bar at the current index (right boundary), the bar at the new stack top (left boundary), and the height of the just-popped bar (bottom).
- Add up all the water as you go.

**Trace this yourself:**
Input: `[2,1,0,2]`
What does the stack look like? How much water is trapped at each step?

**Did you realize?**
This can also be solved using two-pointer techniques for O(1) extra space. If you’re curious, try implementing that after this article!

## Summary and Next Steps

You just learned how a single pattern — the monotonic stack — can unlock seemingly different problems:

- Finding maximal rectangles in histograms and matrices
- Calculating trapped rainwater

**Key Takeaways:**

- Monotonic stacks help you efficiently find the nearest "smaller" or "larger" neighbor for every element.
- Many area/volume problems can be reduced to this pattern.
- In 2D matrices, problems can sometimes be transformed into repeated applications of the 1D pattern.

**Common Pitfalls:**

- Forgetting to process all remaining items in the stack at the end (hence the dummy zero in histogram!)
- Confusing the roles of left and right boundaries or miscalculating widths.
- Not resetting heights correctly when a '0' is encountered in the maximal rectangle.

**Action List:**

- **Solve all three problems yourself:** Try writing the code from scratch without peeking!
- **Experiment:** Change inputs, especially edge cases (all zeros, all ones, valleys, and peaks).
- **Explore Related Problems:** Try "Largest Rectangle in Matrix of 1s" variations or "Next Greater Element" questions.
- **Study Other Solutions:** See how others implement these, especially the two-pointer technique for rainwater.
- **Keep practicing:** If maximal rectangle feels tricky, focus on the histogram version first, then build up.
- **Be patient:** These are challenging problems. Don't get discouraged — every attempt builds your intuition!

You’re now equipped with the monotonic stack superpower. Use it wisely, and keep stacking up your skills!

Happy prepping,

PrepLetter