Topic Introduction

Today's PrepLetter is all about **stack-based area and volume problems**—those classic interview questions where you need to find the largest rectangle, maximal rectangle, or even the volume of water that can be trapped between bars. These problems all share a clever use of stacks to efficiently process sequences, usually histograms or matrices, to find local maxima, minima, or boundaries.

What is a Stack?
A stack is a simple data structure that follows the Last-In, First-Out (LIFO) principle. You add (push) and remove (pop) elements only from the top. This structure is especially helpful when you need to keep track of "what's next" in a sequence, such as matching parentheses or processing nested intervals.

How Does It Work?
Imagine a stack of plates: you can add a plate on top or remove the top plate, but you can't take one from the middle. In programming, stacks are often used to remember the last "choice point" or to efficiently manage ranges or boundaries, especially when you need to quickly check and remove elements in a specific order.

When and Why Are Stacks Useful in Interviews?
Interviewers love stack problems because they test your ability to optimize brute-force solutions using clever data structures. Stacks can reduce time complexity from quadratic to linear by helping you keep track of useful information—like the location of previous smaller or larger elements. They are particularly helpful in problems involving histograms, intervals, or nested structures.

Simple Example (Not from our problem list):
Suppose you want to find the **Next Greater Element** for each item in an array (the nearest element to the right that is bigger). Using a stack, you can process the array from right to left and quickly determine, for each element, the next greater one in O(n) time.

Let's dive into our three problems for today, all united by the stack-based approach for maximum area and volume calculations:

- Largest Rectangle in Histogram
- Maximal Rectangle
- Trapping Rain Water

Why These Three?
Each uses a stack to solve what at first glance seems like a daunting brute-force problem:

- The **Largest Rectangle in Histogram** uses a monotonic stack to calculate the largest rectangle in a 1D histogram.
- **Maximal Rectangle** extends this idea to a 2D binary matrix, processing each row as a histogram.
- **Trapping Rain Water** finds the volume of water trapped between bars by identifying the boundaries using a stack.

We'll start with the simplest (1D histogram), build up to 2D, and finish with a twist on volume calculation.

Problem 1: Largest Rectangle in Histogram

Problem Statement (Rephrased):
Given a list of bar heights representing a histogram, find the area of the largest rectangle that can be formed by one or more contiguous bars.

LeetCode Problem Link

Example Input:

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

Example Output:

`10`

(The rectangle covering bars at positions 2 and 3: min height 5, width 2, area 10.)

How to Think About It:

For every bar, you want to know:

- How far can I extend to the left and right before I hit a shorter bar?

Then, area = (width between those boundaries) x (height of the bar).

Try It Manually:

Take `heights = [2, 4, 2, 1]` and draw the histogram. For each bar, try to find the largest rectangle it can be the smallest bar for.

Brute Force Approach:

For each bar, expand left and right as far as the bar is the smallest, calculating area each time.

- **Time Complexity:** O(n^2)
- Not efficient for large inputs.

Optimal Approach: Monotonic Stack

Core Pattern:

- Use a stack to keep track of indices of increasing bar heights.
- When you find a bar shorter than the one on top of the stack, you know you've found the right boundary for the top bar.
- Pop the stack and calculate area for that bar.

Step-by-Step Logic:

- Add a zero height at the end to flush out remaining bars in the stack.
- For each bar:
  - While the current bar is less than the bar at the top of the stack:
    - Pop the stack, calculate the area using the height of the popped bar.
    - Width = current index - index of previous smaller bar - 1.
  - Push the current index onto the stack.
- Track the maximum area found.

Clean Python Solution:

```
def largestRectangleArea(heights):
    # Add a zero at the end to pop all bars from the stack
    heights.append(0)
    stack = []
    max_area = 0

    for i, h in enumerate(heights):
```

```
        # Pop bars from the stack until the current one is taller
        while stack and heights[stack[-1]] > h:
            height = heights[stack.pop()]
            # If stack is empty, width is 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)

Space Complexity: O(n) (for the stack)

What's Going On Here?

- We add a zero at the end so even the tallest bars get processed.
- For each bar, if it's taller than the last one on the stack, we just push its index.
- If it's shorter, we pop from the stack, calculate the area for the popped bar (it's the smallest in its range), and update max_area.
- The width is determined by the distance between the current index and the index of the previous bar in the stack.

Trace with Example:

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

Stack and area updates look like this:

- i=0, h=2: stack=[0]
- i=1, h=1: pop 0 (height=2), width=1, area=2; stack=[]; stack=[1]
- i=2, h=5: stack=[1,2]
- i=3, h=6: stack=[1,2,3]
- i=4, h=2: pop 3 (height=6), width=1, area=6; pop 2 (height=5), width=2, area=10; stack=[1,4]
- Continue...

Try this Test Case by Hand:

`heights = [2, 4, 2, 1]`

Give It a Go:

Take a moment to solve this on your own before looking at the code. Try to trace the stack and see how each area is calculated!

Problem 2: Maximal Rectangle

How This Problem Connects:

This is the 2D version of the histogram problem. Each row of the matrix can be treated as the base of a histogram, where the height of each bar is the count of consecutive 1s above.

Problem Statement (Rephrased):

Given a 2D binary matrix filled with 0s and 1s, find the largest rectangle containing only 1s and return its area.

LeetCode Problem Link

Example Input:

```
matrix = [
  ["1","0","1","0","0"],
  ["1","0","1","1","1"],
  ["1","1","1","1","1"],
  ["1","0","0","1","0"]
]
```

Example Output:

`6`

(The rectangle covering rows 2-3 and columns 2-4.)

Try It Yourself:

What would you get for:

```
matrix = [
  ["0","1"],
  ["1","0"]
]
```

Brute Force:

Check every possible rectangle in the matrix to see if it only contains 1s.

- **Time Complexity:** O((mn)^2)

Optimal Approach: Stack-Based Histogram per Row

Step-by-Step Logic:

- For each row in the matrix:
  - Update a `heights` array: for each column, if matrix[row][col] is '1', add 1 to heights[col]; else, set heights[col] to 0.
  - For each updated `heights`, use the Largest Rectangle in Histogram algorithm to compute the maximum rectangle ending at that row.
- Track the overall maximum rectangle.

Pseudocode:

```
initialize heights = [0] * number_of_columns
max_area = 0
for each row in matrix:
    for each col:
        if matrix[row][col] == '1':
            heights[col] += 1
        else:
            heights[col] = 0
    area = largestRectangleArea(heights)
    max_area = max(max_area, area)
return max_area
```

(You can reuse the function from Problem 1 for largestRectangleArea.)

Trace with Example:

After each row, heights looks like:

- Row 0: [1,0,1,0,0]
- Row 1: [2,0,2,1,1]
- Row 2: [3,1,3,2,2]
- Row 3: [4,0,0,3,0]

At each step, run the histogram algorithm.

Dry-run Test Case:

Try this:

```
matrix = [
  ["1","1"],
  ["1","1"]
]
```

What is the maximal rectangle area?

Time Complexity: O(mn) (since we process each cell once per row)

Space Complexity: O(n) (for the heights array and stack)

Problem 3: Trapping Rain Water

What's Different:

Now, instead of rectangles, we're computing the total amount of water that can be trapped between bars in a histogram after it rains.

Problem Statement (Rephrased):

Given n non-negative integers representing heights of bars, compute how much water it can trap after raining.

LeetCode Problem Link

Example Input:

```
height = [0,1,0,2,1,0,1,3,2,1,2,1]
```

Example Output:

```
6
```

Try This Case:

```
height = [4,2,0,3,2,5]
```

Brute Force:

For each bar, look for the tallest bar to its left and right, then water trapped is min(left_max, right_max) - height.

- **Time Complexity:** O(n^2)

Optimal Stack-Based Approach

Step-by-Step Logic:

- Initialize a stack and total_water = 0.
- Iterate through each bar:

- While the stack is not empty and current bar is higher than the bar at the top of the stack:
  - Pop the top of the stack (this is the "bottom" of a valley).
  - If the stack is empty after popping, break.
  - Calculate the distance between the current bar and the new top of the stack.
  - Calculate the bounded height: min(current bar, new top) - height at popped index.
  - Add (distance x bounded height) to total_water.
- Push current index onto the stack.

Pseudocode:

```
initialize stack = empty
total_water = 0
for i from 0 to n-1:
    while stack not empty and height[i] > height[stack.top]:
        top = stack.pop()
        if stack is empty:
            break
        distance = i - stack.top - 1
        bounded_height = min(height[i], height[stack.top]) - height[top]
        total_water += distance * bounded_height
    stack.push(i)
return total_water
```

Trace Example:

For `height = [0,1,0,2,1,0,1,3,2,1,2,1]`, walk through how water is trapped at each valley.

Try It Yourself:

Test `height = [2,0,2]` — how much water is trapped?

Time Complexity: O(n)

Space Complexity: O(n) (for the stack)

Summary and Next Steps

You've just explored three classic stack-based problems that challenge you to find maximum area or volume in a histogram or matrix:

- **Largest Rectangle in Histogram:** Monotonic stack to find the largest rectangle for each bar.
- **Maximal Rectangle:** Extend the histogram idea to a 2D matrix by building up heights row by row.
- **Trapping Rain Water:** Use a stack to find and measure valleys efficiently.

Key Patterns to Remember:

- **Monotonic stacks** are your best friend for range queries in sequences (like heights).
- Process each element only once for O(n) time efficiency.
- For 2D problems, try reducing them to 1D by accumulating values per row or column.
- In "trapping" problems, boundaries matter—use stacks to track left and right walls.

Common Traps:

- Forgetting to flush the stack at the end (missing the last rectangle).
- Off-by-one errors in width calculation.
- Not resetting heights correctly between rows in the 2D case.

Action List

- Solve all three problems on your own—even the one with code provided.
- Try to solve Problems 2 and 3 using a different technique (like two-pointer for Trapping Rain Water).
- Explore related problems: "Largest Rectangle in Binary Matrix," "Sum of Subarray Minimums," etc.
- Compare your solutions with others on LeetCode or similar sites—especially for edge cases and style.
- If you get stuck, don’t worry! Practice builds insight and confidence. Keep at it!

Happy practicing! Tomorrow, we’ll tackle another core interview pattern.