LOGIC LABS

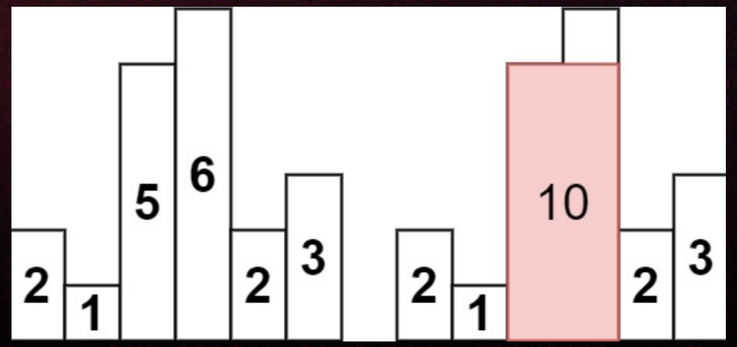
WHERE LOGIC MEETS PRACTICE

Question 1

Given an array of integers representing histogram bar heights (each with width 1), find the area of the largest rectangle that can be formed within the histogram boundaries.

Topics - Array | Stack | NSE | PSE

☑ Test Cases -



Example 1: Mixed Heights

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

Thinking:

- At first glance, the tallest bars are 5 and 6 standing next to each other.
- If we take them together, the shorter one is 5, and since they are side by side, the width = 2.
- So area = 5 × 2 = 10.
- No other combination gives bigger area than this.

Answer: 10

Example 2: All Bars Same

Input:
[5, 5, 5, 5]

Thinking:

- Every bar has the same height, 5.
- If we take all 4 bars together, width = 4.
- So the rectangle becomes 5 × 4 = 20.
- That's clearly the biggest possible rectangle here.

Answer: 20

Codingninjas 10X CLUB ITER CHAPTER

LOGIC LABS

WHERE LOGIC MEETS PRACTICE

HINTS

Hint 1

For each bar, consider it as the shortest bar in a potential rectangle. You need to find how far left and right you can extend before hitting a shorter bar - essentially finding the nearest smaller element on both sides. Once you have left and right boundaries for each bar, calculate the area using: height of current bar × width between the boundaries.

SIKSHA 'O' ANUSANDHAN (DEEMED TO BE UNIVERSITY)

LOGIC LABS

WHERE LOGIC MEETS PRACTICE

HINTS

Hint 1

For each bar, consider it as the shortest bar in a potential rectangle. You need to find how far left and right you can extend before hitting a shorter bar - essentially finding the nearest smaller element on both sides. Once you have left and right boundaries for each bar, calculate the area using: height of current bar × width between the boundaries.

Hint 2

Use a stack to efficiently find Previous Smaller Element (PSE) and Next Smaller Element (NSE) indices for each bar in O(N) time instead of O(N^2) nested loops.

Codingninjas 10X CLUB ITER CHAPTER

LOGIC LABS

WHERE LOGIC MEETS PRACTICE

HINTS

Hint 1

For each bar, consider it as the shortest bar in a potential rectangle. You need to find how far left and right you can extend before hitting a shorter bar - essentially finding the nearest smaller element on both sides. Once you have left and right boundaries for each bar, calculate the area using: height of current bar × width between the boundaries.

Hint 2

Use a stack to efficiently find Previous Smaller Element (PSE) and Next Smaller Element (NSE) indices for each bar in O(N) time instead of O(N^2) nested loops.

Hint 3

Advanced approach: Instead of storing PSE and NSE indices in arrays, use the PSE concept to solve in one single iteration - calculate area directly when you pop elements from the stack.

BRUTE FORCE

The NSE/PSE Strategy The brute force approach uses two key concepts:

- PSE (Previous Smaller Element): The nearest bar to the left that is shorter than the current bar
- NSE (Next Smaller Element): The nearest bar to the right that is shorter than the current bar

Real-World Analogy: The Security Guard's View Think of each histogram bar as a security guard of a certain height. Each guard can see from their left boundary (PSE + 1) to their right boundary (NSE - 1) without obstruction, as long as everyone in that range is at least as tall as them. The "area of influence" for each guard is their height multiplied by their viewing width.

BRUTE FORCE

```
public class LargestRectangleHistogram {
    public static int largestRectangleAreaMethod1(int[] heights) {
        int n = heights.length;
        if (n == 0) return 0;
        int[] pse = previousSmallerElementIndices(heights);          // Find Previous Smaller Element indices
        int[] nse = nextSmallerElementIndices(heights);              // Find Next Smaller Element indices
        int maxArea = 0;                                             // Calculate maximum area
        for (int i = 0; i < n; i++) {
            int width = nse[i] - pse[i] - 1;
            int area = heights[i] * width;
            maxArea = Math.max(maxArea, area);
        }
    return maxArea;
    }
    private static int[] previousSmallerElementIndices(int[] heights) {      //For each element, find index of previous smaller element
      int n = heights.length;
      int[] pse = new int[n];
      Stack<Integer> stack = new Stack<>();
      for (int i = 0; i < n; i++) {
```

BRUTE FORCE

```
            while (!stack.isEmpty() && heights[stack.peek()] >= heights[i]) {        // Remove indices with values >= current height
                stack.pop();
            }
            pse[i] = stack.isEmpty() ? -1 : stack.peek();                            // PSE index (-1 if no smaller element to left)
            stack.push(i);
        }
    return pse;
}
private static int[] nextSmallerElementIndices(int[] heights) {                      //For each element, find index of next smaller element
    int n = heights.length;
    int[] nse = new int[n];
    Stack<Integer> stack = new Stack<>();
    Arrays.fill(nse, n);                                                             // Initialize with n (boundary condition)
    for (int i = n - 1; i >= 0; i--) {                                               // Traverse right to left
        while (!stack.isEmpty() && heights[stack.peek()] >= heights[i]) {            // Remove indices with values >= current height
            stack.pop();
        }
        nse[i] = stack.isEmpty() ? n : stack.peek();                                 // NSE index (n if no smaller element to right)
        stack.push(i);
    }
    return nse;
}
}
```

BRUTE FORCE

TIME COMPLEXITY

O(3n)

The code has three main steps: finding PSE (O(n)), finding NSE (O(n)), and calculating the max area (O(n)). Each step processes the array once, so total time is O(3n), which simplifies to O(n) since constants are ignored in big-O notation.

SPACE COMPLEXITY

O(3n)

It uses two arrays (pse and nse), each of size n, plus a stack that can hold up to n elements in the worst case. Total extra space is O(3n), which simplifies to O(n).

OPTIMIZED SOLUTION

The Stack Innovation Instead of pre-computing NSE and PSE for all bars, we can use a monotonic stack to process bars on-demand. When we encounter a bar that's shorter than the stack top, we've found that bar's NSE, and we can immediately calculate its maximum rectangle.

The Genius of the -1:- Sentinel Real-World Analogy: The Reception Desk System Imagine a hotel reception where guests (histogram bars) are queued by height in descending order. The reception desk has a permanent "ground floor" marker at position -1. When a shorter guest arrives, all taller guests ahead of them must check out (get processed), because the new guest blocks their "view to the right." The ground floor marker ensures we always know the left boundary of any guest's territory. The stack-based solution uses these key innovations: 1. Monotonic Stack: Maintains bars in increasing height order 2. Sentinel Value: Pushes -1 initially to handle boundary cases elegantly 3. On-demand Processing: Calculates rectangles when NSE is found

OPTIMIZED SOLUTION

```
public class Solution {
    public int largestRectangleArea(int[] heights) {
        Stack<Integer> stack = new Stack<>();
        stack.push(-1);                                          // Brilliant sentinel to handle left boundaries
        int maxArea = 0;                                         // Main processing loop
        for (int i = 0; i < heights.length; i++) {               // Pop and process all bars taller than current bar
            while (stack.peek() != -1 && heights[i] <= heights[stack.peek()]) {
                int height = heights[stack.pop()];               // Bar being processed
                int width = i - stack.peek() - 1;                // NSE - PSE - 1
                maxArea = Math.max(maxArea, height * width);
            }
            stack.push(i);
        }
        while (stack.peek() != -1) {                             // Process remaining bars (those extending to array end)
            int height = heights[stack.pop()];
            int width = heights.length - stack.peek() - 1;       // NSE = n
            maxArea = Math.max(maxArea, height * width);
        }
        return maxArea;
    }
}
```

OPTIMIZED SOLUTION

TIME COMPLEXITY

O(2n)

The solution processes the array in two passes: one main loop (O(n)) and a while loop for remaining bars (O(n) in total, as each element is popped once). The combined work is O(2n), which simplifies to O(n). The stack operations are constant time per element.

SPACE COMPLEXITY

O(n)

It uses a single stack that, in the worst case, holds up to n indices. The space for the stack is O(n), and no extra arrays are needed, making it efficient.

Practice Problems and Extensions

1. Maximal Rectangle: Extend this to 2D binary matrices
2. Trapping Rain Water: Similar stack-based approach
3. Sum of Subarray Minimums: Uses the same NSE/PSE concept

Conclusion

The evolution from brute force to optimal solution demonstrates several key algorithmic principles:

- Pattern Recognition: Identifying the NSE/PSE pattern
- Data Structure Selection: Choosing stacks for monotonic properties
- Boundary Handling: Using sentinel values to eliminate edge cases
- Amortized Analysis: Understanding why O(n) operations can achieve linear time

Codingninjas 10X CLUB ITER CHAPTER

LOGIC LABS

WHERE LOGIC MEETS PRACTICE

SIKSHA 'O' ANUSANDHAN (DEEMED TO BE UNIVERSITY)

Fill the feedback form, because your thoughts matter

Codingninjas 10X CLUB ITER CHAPTER

LOGIC LABS

WHERE LOGIC MEETS PRACTICE

SIKSHA 'O' ANUSANDHAN (DEEMED TO BE UNIVERSITY)

THANK YOU

TEAM - LOGIC LABS

cn.10x.iter

Coding Ninjas 10X ITER

CN X ITER