Problem Description

Hard

Array

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.

The task is to find the largest rectangular area in a histogram, which consists of a series of adjacent bars of varying heights. Each

Intuition

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.

The intuition behind the solution involves using a stack and a concept called "Monotonic Stack" which helps us process the bars in a

For each bar h with index 1:

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.

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

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,

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

- 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]
- 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,

- 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

- 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:

2. Traversing Heights: We loop through each bar using its index i and height h:

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).

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

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),

stack, we pop from the stack. This indicates that we have found a right boundary for the rectangle concerning the bar at the

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

Finally, we add the current index i to the top of the stack.

Use the max function to determine the largest area from these.

since we now know this bar is taller than the current bar h.

index that was just popped.

rectangles for each bar. To calculate the largest area: Iterate over each index i and corresponding height h, and calculate the width for the largest rectangle that can be formed with height h 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.

3. Calculating the Largest Area: Once the traversal is done, we have left and right arrays filled with the bounds of potential

the stack. Hence, the left boundary for the current bar h is now known to be the index on top of the stack.

one line, showcasing an efficient Pythonic approach: return max(h * (right[i] - left[i] - 1) for i, h in enumerate(heights))

The code snippet provided uses list comprehension at the end to perform the calculation of the areas and find the maximum all in

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

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

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.

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

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

• 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):

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

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

stk = [1]

stk = [1, 2]

stk = [1, 2, 3]

stk = [1, 4, 5]

4. Return the Largest Area:

stack = []

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

27

28

29

30

31

33

34

35

36

37

38

6

9

10

11

12

19

20

21

23

24

25

26

27

28

29

30

31

32

33

32

33

34

35

36

37

40

39 };

12

13

14

15

16

17

18

19

20

27

29

32

33

34

35

36

37

38

39

40

41

42

44

43 }

28 }

Python Solution

3. Calculating the Largest Area:

stk = [1]

stk = []

stk = [0]

histogram.

Example Walkthrough

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

1. Initializing Data Structures:

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

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

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

stk = [1, 4]For heights [5] (3):

The maximum calculated area is 10, which is the area of the largest rectangle in the histogram we started with.

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

• For i=2 (height = 5): width = right[2] - left[2] - 1 = 4 - 1 - 1 = 2

• We've completed the traversal, left = [-1, -1, 1, 2, 1, 4] and right = [6, 2, 4, 4, 6, 6].

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

smaller_left_index = [-1] * num_bars

smaller_right_index = [num_bars] * num_bars

while stack and heights[stack[-1]] >= height:

smaller_left_index[index] = stack[-1]

Calculate the maximum area of rectangle in histogram

Update max_area with the larger area found

int length = heights.length; // Total number of bars.

// Stack to keep track of indices of the bars.

rightBoundary[stack.pop()] = i;

// Push the current index onto the stack.

stack.push(i);

for (int i = 0; i < length; ++i) {

for (int i = 0; i < numBars; ++i)

// Return the maximum area found.

10 // Finds the largest rectangle area in a histogram.

leftNearest = new Array(numBars).fill(-1);

populateNearestSmallerIndices(numBars);

for (let i = 0; i < numBars; ++i) {</pre>

// Push current bar to stack.

indexStack.push(i);

Time and Space Complexity

rightNearest = new Array(numBars).fill(numBars);

return maxArea; // Return the maximum area found.

function populateNearestSmallerIndices(numBars: number) {

const numBars = heights.length;

function largestRectangleArea(inputHeights: number[]): number {

indexStack = []; // Reset the stack for the new calculation.

// Fill in values for leftNearest and rightNearest arrays.

maxArea = 0; // Reset the maximum area for the new calculation.

// Populate the nearest smaller indices on both left and right for each bar

rightNearest[indexStack.pop() as number] = i;

// Set the right nearest smaller bar for the popped bar.

// Pop elements from the stack until the current bar is taller than the stack's top bar.

// 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];

while (indexStack.length > 0 && heights[indexStack[indexStack.length - 1]] >= heights[i]) {

return maxArea;

2 let indexStack: number[] = [];

Typescript Solution

3 let maxArea: number = 0;

let heights: number[];

7 let leftNearest: number[];

8 let rightNearest: number[];

Deque<Integer> stack = new ArrayDeque<>();

int[] leftBoundary = new int[length];

int[] rightBoundary = new int[length];

int maxArea = 0; // This variable will store the maximum area found.

// Arrays to keep track of the left and right boundaries of each bar.

// and set their right boundary to the current bar's index.

leftBoundary[i] = stack.isEmpty() ? -1 : stack.peek();

while (!stack.isEmpty() && heights[stack.peek()] >= heights[i]) {

// If the stack is empty, then there's no smaller bar to the left.

// Otherwise, the stack's top is the previous smaller bar's index.

// Calculate the largest rectangle area for each bar using their boundaries.

bar of smaller height (left boundary)

smaller_right_index[stack[-1]] = index

for index, height in enumerate(heights):

stack.pop()

stack.append(index)

Push this bar onto stack

for i, h in enumerate(heights):

public int largestRectangleArea(int[] heights) {

if stack:

max_area = 0

Calculate the widths and areas for each bar; for example:

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]).

Initialize arrays to record the first smaller bar on the left of each bar

Initialize arrays to record the first smaller bar on the right of each bar

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

Iterate over all heights to compute the smaller_left_index and smaller_right_index

If the stack is not empty, the current element at the top is the previous

max_area = max(max_area, h * (smaller_right_index[i] - smaller_left_index[i] - 1))

return max_area

13 // Initialize right boundaries as the length of the array. Arrays.fill(rightBoundary, length); 14 15 // Iterate over all bars to calculate left and right boundaries. 16 for (int i = 0; i < length; ++i) { 17 // Pop elements from the stack until the current bar is taller than the stack's top 18

Java Solution

class Solution {

```
// Calculate width of the current bar's largest rectangle.
34
               int width = rightBoundary[i] - leftBoundary[i] - 1;
35
               // Calculate area and update maxArea if it's larger.
36
37
               maxArea = Math.max(maxArea, heights[i] * width);
38
39
           // Return the maximum area found.
40
           return maxArea;
42
43 }
44
C++ Solution
1 #include <vector>
2 #include <stack>
   using namespace std;
   class Solution {
6 public:
       int largestRectangleArea(vector<int>& heights) {
           // The maximum area found so far.
           int maxArea = 0;
           // The total number of bars.
10
           int numBars = heights.size();
11
           // Stack to keep track of the indices of the bars.
           stack<int> indexStack;
13
14
           // Vector to store the index of the left nearest smaller bar for each bar.
           vector<int> leftNearest(numBars, -1);
15
           // Vector to store the index of the right nearest smaller bar for each bar.
16
           vector<int> rightNearest(numBars, numBars);
17
18
           for (int i = 0; i < numBars; ++i) {</pre>
               // Pop elements from the stack until the current bar is taller than the stack's top bar.
20
               while (!indexStack.empty() && heights[indexStack.top()] >= heights[i]) {
21
                    // Set the right nearest smaller bar for the popped bar.
                    rightNearest[indexStack.top()] = i;
23
24
                    indexStack.pop();
25
               // If the stack is not empty, then the current top is the left nearest smaller bar.
26
               if (!indexStack.empty()) leftNearest[i] = indexStack.top();
27
               // Push current bar to stack.
28
               indexStack.push(i);
29
30
31
```

// Calculate the maximum area for each bar considering the nearest smaller bars to the left and right.

maxArea = max(maxArea, heights[i] * (rightNearest[i] - leftNearest[i] - 1));

1 // TypeScript doesn't have predefined Stack class, so we use an array to simulate stack behavior.

6 // Variables leftNearest and rightNearest are declared globally to be used in functions below.

heights = inputHeights; // Initialize global heights variable with the input.

21 22 // Calculate the maximum area for each bar considering the nearest smaller bars to the left and right. 23 for (let i = 0; i < numBars; ++i) {</pre> 24 maxArea = Math.max(maxArea, heights[i] * (rightNearest[i] - leftNearest[i] - 1)); 25 26

```
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
```

heights, the stack may perform a push operation (stk.append(i)). Additionally, while the current height is less than or equal to

rectangle is updated. Despite this, each element is added to the stack once and removed from the stack at most once over the

the height of the rectangle corresponding to the index at the top of the stack, pop operations occur, and the right bound for the

- Based on these points, the overall time complexity of the largestRectangleArea function is O(n).
 - 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.
- 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
 - As for the space complexity:

entire run of the loop, leading to a total of O(n) operations.