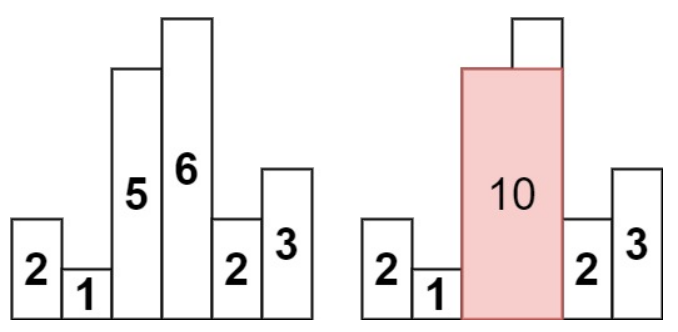
<https://leetcode.com/problems/largest-rectangle-in-histogram/description/>

Given an array of integers heights representing the histogram's bar height where the width of each bar is 1, return *the area of the largest rectangle in the histogram*.

**Example 1:**



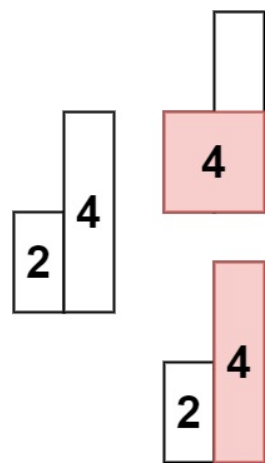
Input: heights = [2,1,5,6,2,3]

Output: 10

Explanation: The above is a histogram where width of each bar is 1.

The largest rectangle is shown in the red area, which has an area = 10 units.

**Example 2:**



Input: heights = [2,4]

Output: 4

**Constraints:**

* 1 <= heights.length <= 105
* 0 <= heights[i] <= 104

**Attempt 1: 2023-03-24**

**Solution 1: Increasing Monotonic Stack (360 min)**

class Solution {

public int largestRectangleArea(int[] heights) {

int maxArea = 0;

Stack<Integer> stack = new Stack<Integer>();

// 'i <= heights.length' includes '==' condition

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

// Corner case to add dummy height = 0 when i == heights.length

// which helps calculation of rectangle area close to right end

int h = (i == heights.length ? 0 : heights[i]);

if(stack.isEmpty() || h >= heights[stack.peek()]) {

stack.push(i);

} else {

int height = heights[stack.pop()];

int width = 0;

if(stack.isEmpty()) {

width = i;

} else {

//int right\_boundary = i - 1;

//int left\_boundary = stack.peek();

//width = right\_boundary - left\_boundary + 1;

// Why left boundary is stack.peek() ?

// Take {2,1,5,6,2,3} as example, when process the 2nd 2

// which index = 4, originally we have stored index = {1,2,3}

// on stack, corresponding to {1,5,6}, because 2 < 6, pop

// peak index = 3, now we have {1,2} stored on stack, the

// left boundary index is stack.peek() as 2, which represents

// the rectangle height = 5, now why we say height as 5 rectangle

// must be left boundary, because of maintaining an increasing

// monotonic stack, if any rectangle before height as 5 rectangle

// exist, its corresponding index would be popped out from the

// stack already before we push current height as 5 rectangle's

// index = 2 onto stack based on 'else branch logic', so the remain

// height as 5 rectangle on index = 2 is definitely the "shortest"

// rectangle still existing on stack, but also be the "highest"

// rectangle pushing into stack till now, can be used as left boundary

width = i - 1 - stack.peek();

}

maxArea = Math.max(maxArea, height \* width);

// Why we have to use i-- ?

// When we in the else branch, means the h (height[i]) is

// smaller than height[s.peek()], what we do is updating

// the maxArea, but the height[i] is still waiting to be

// put into the stack, we do i-- to counteract the i++

// statement in the for loop, so that we will get the same

// i in the next time

// Take {2,1,5,6,2,3} as example, when try tp push 1 onto stack,

// since 1 < 2 go into "else branch" but no push logic, only pop

// logic in "else branch", to push it we have to waiting till

// next iteration, to compensate auto i++, we preparatorily

// decrease as i-- in current iteration

i--;

}

}

return maxArea;

}

}

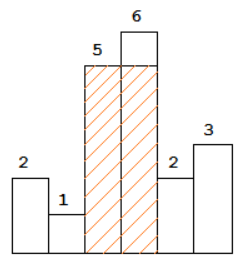
**Refer to**

<https://leetcode.com/problems/largest-rectangle-in-histogram/solutions/28900/short-and-clean-o-n-stack-based-java-solution/>

Two key points that I found helpful while understanding the solution:

1. When a bar is popped, we calculate the area with the popped bar at index tp as shortest bar. Now we know the rectangle **height** is heights[tp], we just need rectangle width to calculate the area.
2. How to determine rectangle **width**? The maximum width we can have here would be made of all connecting bars with height greater than or equal to heights[tp]. heights[s.peek() + 1] >= heights[tp] because the index on top of the stack right now s.peek() is the first index left of tp with height smaller than tp's height (if s.peek() was greater then it should have already been popped out of the stack). heights[i - 1] >= heights[tp] because index i is the first index right of tp with height smaller than tp's height (if i was greater then tp would have remained on the stack). Now we multiply height heights[tp] by width i - 1 - s.peek() to get the **area**.

<https://leetcode.com/problems/largest-rectangle-in-histogram/solutions/28900/short-and-clean-o-n-stack-based-java-solution/comments/27725>



I would like to explain i - 1 - stack.peek() in this way:

Let's take the second bar(2) as the example. bar(2) means the bar with height = 2.

When i is at bar(2), the top in stack is bar(6). If we take bar(6) as the current height of the rectangle, we need to find the left and right boundary bar of the rectangle. It is always true that both left and right boundary bar has a height either equal or larger than bar(6).

Right Boundary: bar(2), which i points to now, is the first bar that is lower than bar(6) on its right, so the right boundary is (i - 1).Left Boundary: bar(5), which is the top in stack after bar(6) is popped, is the first bar that is lower than bar(6) on its left, so the left boundary is (stack.peek() + 1).

Now we have found both left and right boundaries, then let's get the width of the rectangle. The width = index of right boundary - index of left boundary + 1 = (i - 1) - (stack.peek() + 1) + 1 = i - 1 - stack.peek().Then the area = (height of bar(6)) \* (i - 1 - stack.peek()).

Let's move to the next loop. The top in stack now is bar(5), and i points to bar(2). So we pop bar(5) out from the stack and take bar(5)'s height as the rectangle's height.

Right Boundary: bar(2) is the first bar that is lower than bar(5) on its right. Then the right boundary is (i - 1).Left Boundary: bar(1), which is the top in stack after bar(5) is popped, is the first bar that is lower than bar(5) on its left. So the left boundary is (stack.peek() + 1).

The width = (i - 1) - (stack.peek() + 1) = i - 1 - stack.peek() and area = (height of bar(5)) \* (i - 1 - stack.peek())).

There is only bar(1) left in the stack, and it is lower than bar(2), which i points to now. So we continue pushing bars to stack till the height starts to decrease or the end of the histogram. Now the stack has bar(1), bar(2), bar(3) and bar(3) is on the top. i points to the position on the right of bar(3).

We pop bar(3) out of the stack and take its height as the rectangle's height. The right boundary is (i - 1); the left boundary is bar(2) which is (stack.peek() + 1).

Next, pop bar(2) out of the stack and take its height as the rectangle's height. Remember that the boundary's height is either equal or larger than bar(2)'s height. The right boundary is bar(3), which is (i - 1). The left boundary is bar(1)'s right which is (stack.peek() + 1).

What if the stack is empty when looking for the left boundary? Let's take the example when stack contains bar(1) only. It's obvious that the right boundary is (i - 1). After popping bar(1) out, the stack is empty. It means there is no bar that is lower than bar(1) on its left. So the left boundary is all the way to the left, which is the most left of the histogram with index = 0. Then the width = (i - 1) - 0 + 1 = i. The area = (height of bar(1)) \* i.

So the idea is that:

1. For each bar, take its height as the rectangle's height. Then find the left and right boundaries of this rectangle.
2. (The second top bar in stack) is always the first bar lower than (the top bar in stack) on the left.
3. (The bar that i points to) is always the first bar lower than (the top bar in stack) on the right.
4. After step 2 and 3, we know the left and right boundaries, then know the width, then know the area.

**Refer to**

<https://leetcode.com/problems/largest-rectangle-in-histogram/solutions/28900/short-and-clean-o-n-stack-based-java-solution/comments/27741>

can someone tell why we need i--?

when we in the else branch, means the h (height[i]) is smaller than height[s.peek()], what we do is updating the maxArea, but the height[i] is still waiting to be put into the stack, we do i-- to counteract the i++ statement in the for loop, so that we will get the same i in the next time