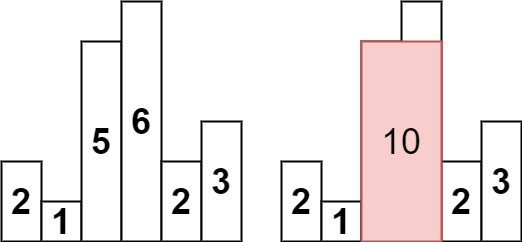
<https://leetcode.com/problems/largest-rectangle-in-histogram/description/>

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:**



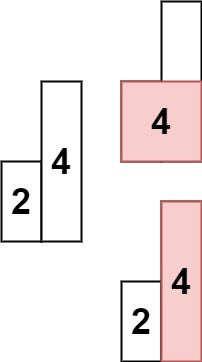
Input: heights = [2,1,5,6,2,3]

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 <= 10^5

0 <= heights[i] <= 10^4

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

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

**Style 1: for loop (not standard while loop template)**

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/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

**Style 2: while loop template**

class Solution {

    public int largestRectangleArea(int[] heights) {

        int maxArea = 0;

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

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

            // Use a sentinel value at the end to ensure all bars are processed

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

            // Calculate the area for all elements in the stack that are taller

            // than the current height

            while(!stack.isEmpty() && h < heights[stack.peek()]) {

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

                int width = stack.isEmpty() ? i : i - 1 - stack.peek();

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

            }

            stack.push(i);

        }

        return maxArea;

    }

}

**Refer to chatGPT**

**Explanation:**

Stack Initialization:

A Stack<Integer> is used to keep track of the indices of the histogram bars.

maxArea is initialized to store the maximum rectangle area found.

Iterating Through Histogram Bars:

We iterate through each bar in the histogram using a loop that runs from 0 to n (inclusive). The extra iteration (i == n) is used to handle the case when the stack still contains indices of bars that need to be processed. By setting h to 0 when i == n, it forces the processing of all remaining bars in the stack.

Calculating the Area:

While the stack is not empty and the current bar height h is less than the height of the bar represented by the top index of the stack (heights[stack.peek()]), we pop the stack.

The popped value represents the height of the bar. We calculate the width of the rectangle as the difference between the current index i and the new top of the stack minus one. If the stack is empty after popping, it means the rectangle extends from the beginning (width = i).

Update maxArea with the maximum of the current area calculated and the previously recorded maximum area.

Pushing Current Index:

Push the current index i onto the stack.

Return the Result:

After the loop, maxArea contains the largest rectangle area that can be formed in the histogram.

**Example:**

For an input heights = [2, 1, 5, 6, 2, 3], the output would be 10, which is the area of the largest rectangle (formed between bars of height 5 and 6).

This stack-based approach runs in O(n) time complexity, making it efficient for large inputs.

**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:

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.

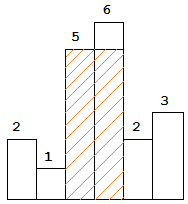
How to determine rectangle **width**? The maximum width we can have here would be made of all connecting bars with height greater than or equal to heights[tp]. heights[s.peek() + 1] >= heights[tp] because the index on top of the stack right now s.peek() is the first index left of tp with height smaller 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>

**我们简单说一下这个题的核心思路，以及为何用到单增 stack (单增的定义是指存储的坐标所对应的竖棍的高度维持单增，不是坐标本身单增) ？**

**以下图坐标为 4 的竖棍为例，在扫描到坐标4的时候，此前坐标 0 到 3 中所形成的最大的方块如何计算呢？**

**可以将坐标为 4 的竖棍的高度 2 当做一个基准线，我们从坐标 0 到 3 中所有的方块组合中寻找最大的一个，我们采用单增 stack，里面存储的是所有大于当前基准线竖棍高度的左边的竖棍的坐标，比如这里，扫描到坐标为 4 的竖棍时，stack 中存储的是 1, 2, 3 这三个坐标，分别对应的是高度为 1, 5, 6 这三个竖棍，因为在存储到 3 这个坐标时，之前坐标 1 和 2 对应的两个竖棍高度 1, 5 都比 6 小，按照单增 stack 的定义这样存储符合定义，所以 1, 2, 3 这三个坐标都被存储在 stack 中，直到遇到坐标为 4 的竖棍时，该竖棍的高度为 2，它的高度比当前 stack 顶端存储的坐标 2 所代表的竖棍高度 6 小，不符合单增 stack 的定义，所以开始从 stack 中抛出顶端存储的坐标 2，在抛出的同时计算了对应方块的高度 (6) 和宽度 (4 - 1 - stack.peek() = 4 - 1 - 2 = 1，这里的 stack.peek() 很有意思，它本质上代表了当前截取的方块的左边界对应的坐标)，对应方块的大小为 6 \* 1 = 6，抛出的过程到此就结束了吗？并不是，我们知道的根据单增 stack 的定义，任何时候存储在 stack 顶端的坐标所对应的竖棍的高度都不应该大于基准线竖棍的高度，而就算抛出了顶端的坐标 3 以后，新的顶端的坐标 2 所对应的竖棍的高度是 5，依然大于基准线竖棍的高度 2，所以这个坐标也应该被抛出，同样，抛出的同时计算了对应方块的高度 (5) 和宽度 (4 - 1 - stack.peek() = 4 - 1 - 1 = 2)，对应方块的大小为 5 \* 2 = 10，这里也解释了为何采用 while 内循环的意义，必须保证穷尽所有的可能，那么抛出的过程到此结束了吗？对，此时才结束，因为剩下的 stack 顶端存储的坐标是 1，所对应的竖棒的高度是 1，小于基准线竖棒的高度 2，不用再抛出，所对应的物理意义在于，用盛水的盆子左右两边的高度来理解，左边界的竖棍高度小于右边的基准线的竖棍高度了，没办法盛下水了，抛出的时候已经把左边界的竖棍大于右边的基准线竖棍高度的 (能盛下水) 所有可能全部抛出了，从右向左望过去已经没有任何竖棍阻挡视野了，也就意味着没有能盛水的空间了，这也是使用单增 stack 所要达到的目的，保证没有比当前竖棍高的竖棍坐标存放在 stack 上，保证在扫描到当前坐标的时候，前面经历过的所有作为左边界的竖棍 (盛水的盆子的左边) 都被寻找到并计算过**



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:

For each bar, take its height as the rectangle's height. Then find the left and right boundaries of this rectangle.

(The second top bar in stack) is always the first bar lower than (the top bar in stack) on the left.

(The bar that i points to) is always the first bar lower than (the top bar in stack) on the right.

After step 2 and 3, we know the left and right boundaries, then know the width, then know the area.

**Refer to**

[L2281.Sum of Total Strength of Wizards](note://WEB4d6d3e6a86ec6eb31f53096b54e62fc5)

[L496.Next Greater Element I (Ref.L739)](note://A4AEA71013F14D71BFC337D45B5BEEAD)

[L739.Daily Temperatures](note://6FB3A11E2B7746DCA411301F941DD948)