# Question

Given *n* non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.

  
Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3].

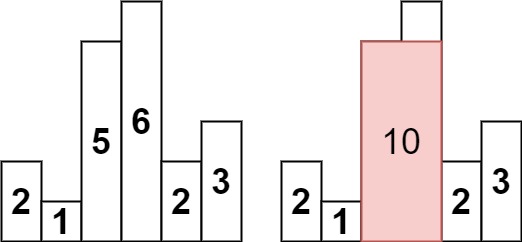
  
The largest rectangle is shown in the shaded area, which has area = 10 unit.

**Example:**

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

**Output:** 10

**Example 1:**



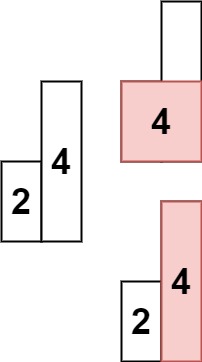
**Input:** heights = [2,1,5,6,2,3]

**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

# Solution

## **Solution Article**

We need to find the rectangle of largest area that can be formed by using the given bars of histogram.

#### **Approach 1: Brute Force**

Firstly, we need to take into account the fact that the height of the rectangle formed between any two bars will always be limited by the height of the shortest bar lying between them which can be understood by looking at the figure below:

Thus, we can simply start off by considering every possible pair of bars and finding the area of the rectangle formed between them using the height of the shortest bar lying between them as the height and the spacing between them as the width of the rectangle. We can thus, find the required rectangle with the maximum area.

|  |
| --- |
| public class Solution {  public int largestRectangleArea(int[] heights) {  int maxarea = 0;  for (int i = 0; i < heights.length; i++) {  for (int j = i; j < heights.length; j++) {  int minheight = Integer.MAX\_VALUE;  for (int k = i; k <= j; k++)  minheight = Math.min(minheight, heights[k]);  maxarea = Math.max(maxarea, minheight \* (j - i + 1));  }  }  return maxarea;  }  } |

**Complexity Analysis**

* Time complexity : O(n^3)*O*(*n*3). We have to find the minimum height bar O(n)*O*(*n*) lying between every pair O(n^2)*O*(*n*2).
* Space complexity : O(1)*O*(1). Constant space is used.

#### **Approach 2: Better Brute Force**

**Algorithm**

We can do one slight modification in the previous approach to optimize it to some extent. Instead of taking every possible pair and then finding the bar of minimum height lying between them everytime, we can find the bar of minimum height for current pair by using the minimum height bar of the previous pair.



|  |
| --- |
| public class Solution {  public int largestRectangleArea(int[] heights) {  int maxArea = 0;  int length = heights.length;  for (int i = 0; i < length; i++) {  int minHeight = Integer.MAX\_VALUE;  for (int j = i; j < length; j++) {  minHeight = Math.min(minHeight, heights[j]);  maxArea = Math.max(maxArea, minHeight \* (j - i + 1));  }  }  return maxArea;  }  } |

**Complexity Analysis**

* Time complexity : O(n^2). Every possible pair is considered
* Space complexity : O(1). No extra space is used.

#### **Approach 3: Divide and Conquer Approach**

**Algorithm**

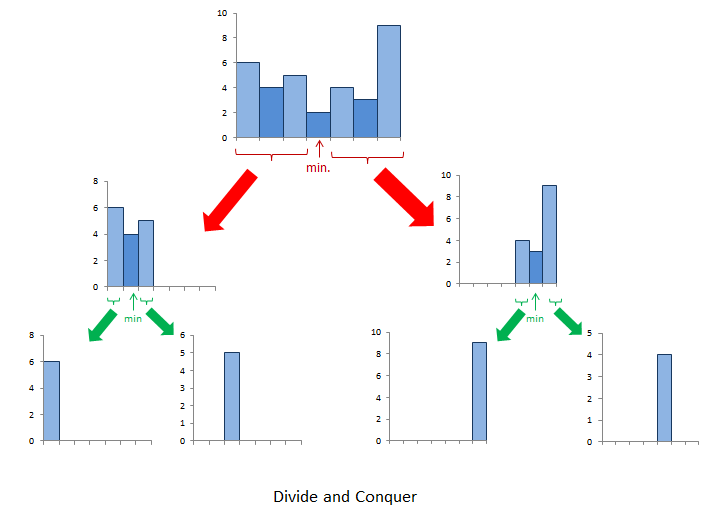
This approach relies on the observation that the rectangle with maximum area will be the maximum of:

1. The widest possible rectangle with height equal to the height of the shortest bar.
2. The largest rectangle confined to the left of the shortest bar(subproblem).
3. The largest rectangle confined to the right of the shortest bar(subproblem).

Let's take an example:

[6, 4, 5, 2, 4, 3, 9]

Here, the shortest bar is of height 2. The area of the widest rectangle using this bar as height is 2x8=16. Now, we need to look for cases 2 and 3 mentioned above. Thus, we repeat the same process to the left and right of 2. In the left of 2, 4 is the minimum, forming an area of rectangle 4x3=12. Further, rectangles of area 6x1=6 and 5x1=5 exist in its left and right respectively. Similarly we find an area of 3x3=9, 4x1=4 and 9x1=9 to the left of 2. Thus, we get 16 as the correct maximum area. See the figure below for further clarification:



|  |
| --- |
| **public class Solution {**  **public int calculateArea(int[] heights, int start, int end) {**  **if (start > end)**  **return 0;**  **int minindex = start;**  **for (int i = start; i <= end; i++)**  **if (heights[minindex] > heights[i])**  **minindex = i;**  **return Math.max(heights[minindex] \* (end - start + 1),**  **Math.max(calculateArea(heights, start, minindex - 1),**  **calculateArea(heights, minindex + 1, end))**  **);**  **}**  **public int largestRectangleArea(int[] heights) {**  **return calculateArea(heights, 0, heights.length - 1);**  **}**  **}** |

**Complexity Analysis**

* Time complexity :

Average Case: *O*(*n*log*n*).

Worst Case: O(n^2). If the numbers in the array are sorted, we don't gain the advantage of divide and conquer.

* Space complexity : *O*(*n*). Recursion with worst case depth n*n*.

#### **Approach 4: Better Divide and Conquer**

**Algorithm**

You can observe that in the Divide and Conquer Approach, we gain the advantage, since the large problem is divided into substantially smaller subproblems. But, we won't gain much advantage with that approach if the array happens to be sorted in either ascending or descending order, since every time we need to find the minimum number in a large subarray O(n)*O*(*n*). Thus, the overall complexity becomes O(n^2)*O*(*n*2) in the worst case. We can reduce the time complexity by using a Segment Tree to find the minimum every time which can be done in O\big(\log n\big)*O*(log*n*) time.

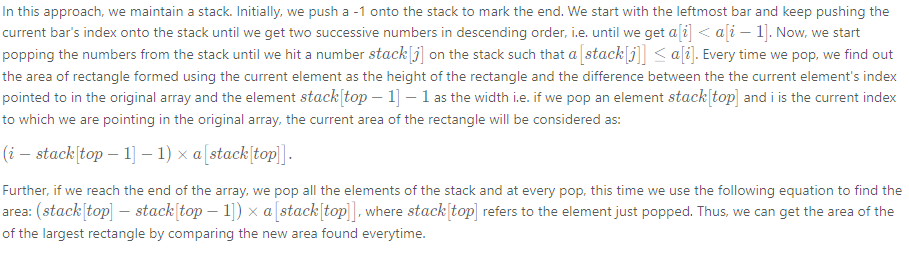
For implementation, click [here](https://leetcode.com/problems/largest-rectangle-in-histogram/discuss/28941/segment-tree-solution-just-another-idea-onlogn-solution).

**Complexity Analysis**

* Time complexity : O\big(n\log n\big)*O*(*n*log*n*). Segment tree takes \log nlog*n* for a total of n*n* times.
* Space complexity : O(n)*O*(*n*). Space required for Segment Tree.

#### **Approach 5: Using Stack**

**Algorithm**



The following example will clarify the process further:

[6, 7, 5, 2, 4, 5, 9, 3]

1 / 18

|  |
| --- |
| **public class Solution {**  **public int largestRectangleArea(int[] heights) {**  **Deque<Integer> stack = new ArrayDeque<>();**  **stack.push(-1);**  **int length = heights.length;**  **int maxArea = 0;**  **for (int i = 0; i < length; i++) {**  **while ((stack.peek() != -1)**  **&& (heights[stack.peek()] >= heights[i])) {**  **int currentHeight = heights[stack.pop()];**  **int currentWidth = i - stack.peek() - 1;**  **maxArea = Math.max(maxArea, currentHeight \* currentWidth);**  **}**  **stack.push(i);**  **}**  **while (stack.peek() != -1) {**  **int currentHeight = heights[stack.pop()];**  **int currentWidth = length - stack.peek() - 1;**  **maxArea = Math.max(maxArea, currentHeight \* currentWidth);**  **}**  **return maxArea;**  **}**  **}** |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). n*n* numbers are pushed and popped.
* Space complexity : O(n)*O*(*n*). Stack is used.