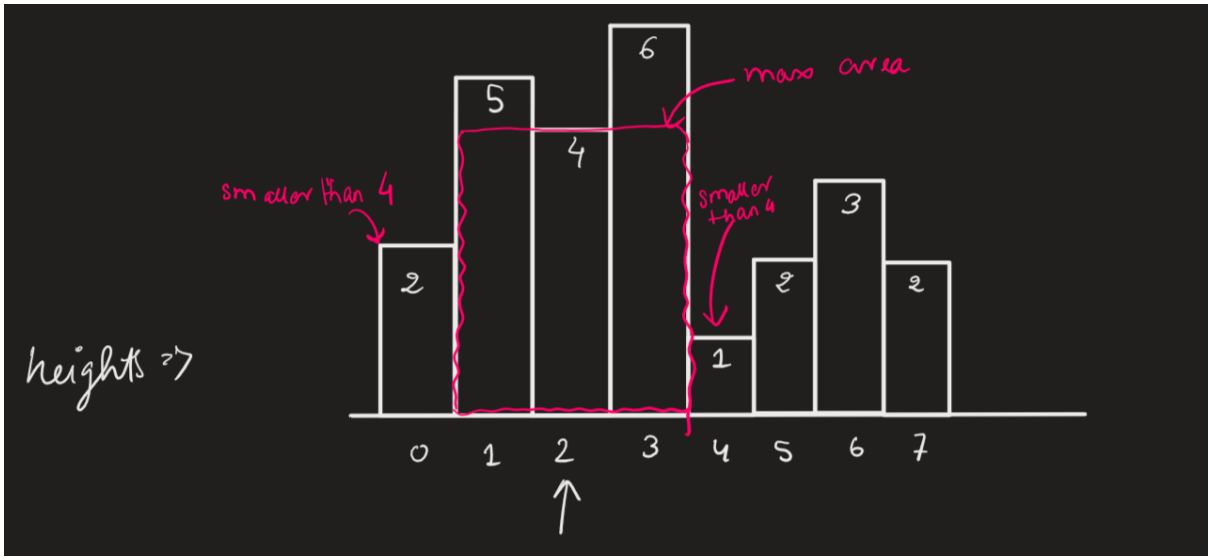
LC84-Largest Rectangle in Histogram

Question Link: https://leetcode.com/problems/largest-rectangle-in-histogram/

We are given an array of heights of blocks kept adjacent to each other, and we need to find the maximum area that can be obtained by considering multiple blocks.

Intuition



Say we consider including the height at index 2, i.e. heights[2] = 4. Now the **maximum area that we can obtain by including this block height and multiple blocks around it can be limited to the region between: the height on its left that is smaller than itself, and the height on its right that is smaller than itself.** You can test and check this inference by yourself.

So our target will be to find for every block height in given array, the height smaller than itself on its left and the height smaller than itself on its right.

Algorithm & Code

There can be multiple approaches to solve this problem➖

(i) Brute Force Approach - Nested loop

For every heights[i], we can traverse on its left and right to find the nearest height smaller than itself. And then we can use the width between them multiplied by heights[i] to get the area. We will keep on updating the max area, as we find the area including every block.

CODE

```
// WRITE THIS ONE YOURSELF BUDDY :)
```

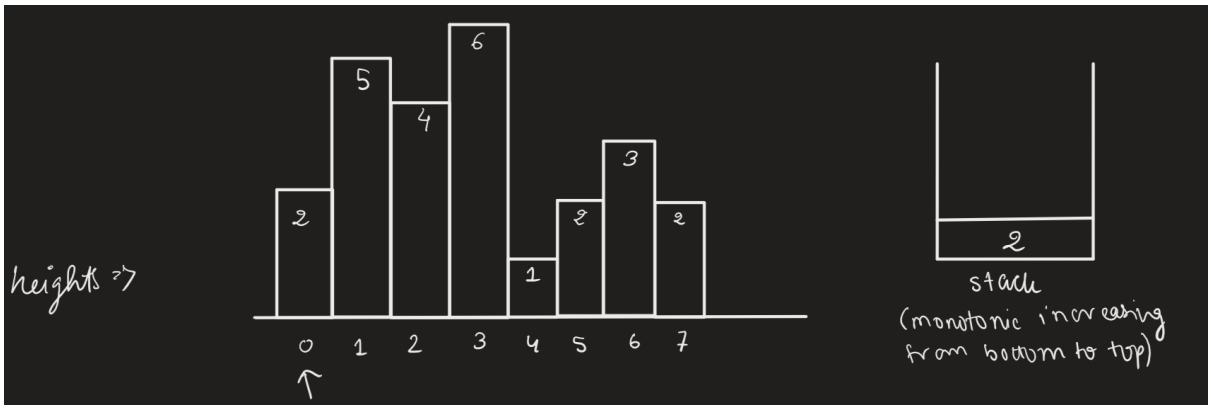
Time Complexity: O(N^2)

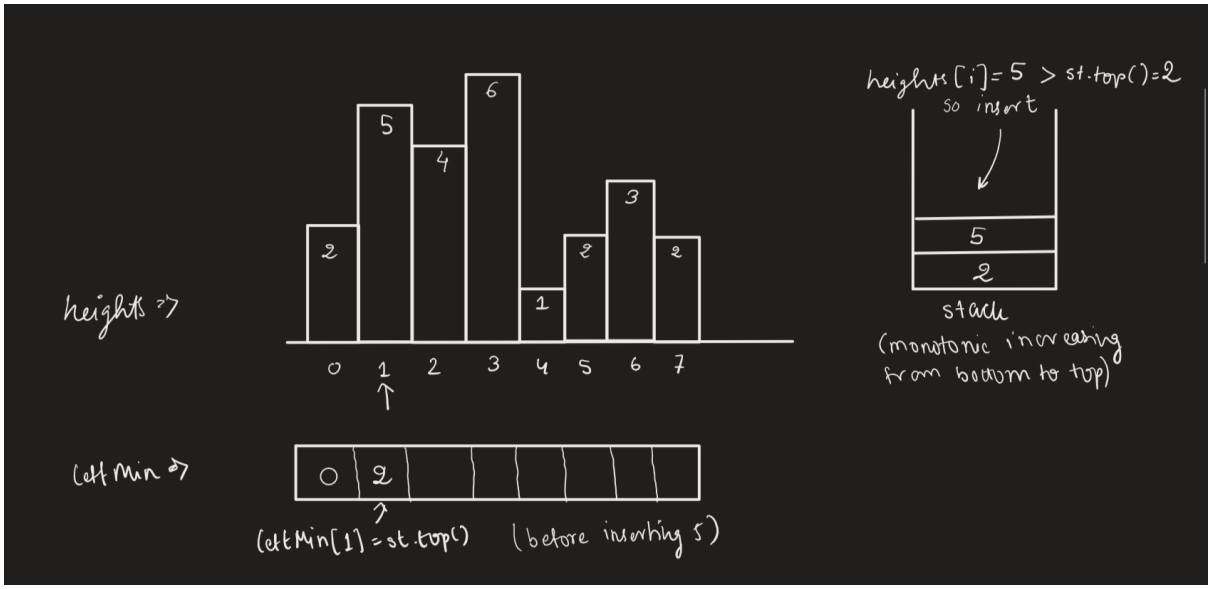
Space Complexity: O(1)

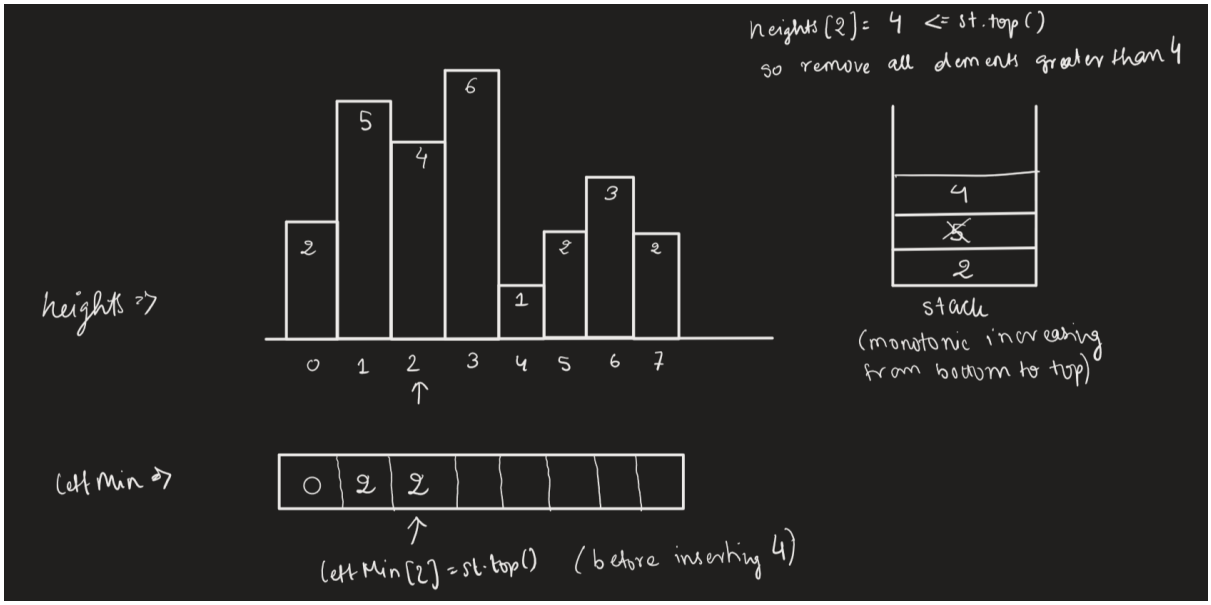
[LeetCode shows TLE for this solution on submitting! But you can test it on custom inputs]

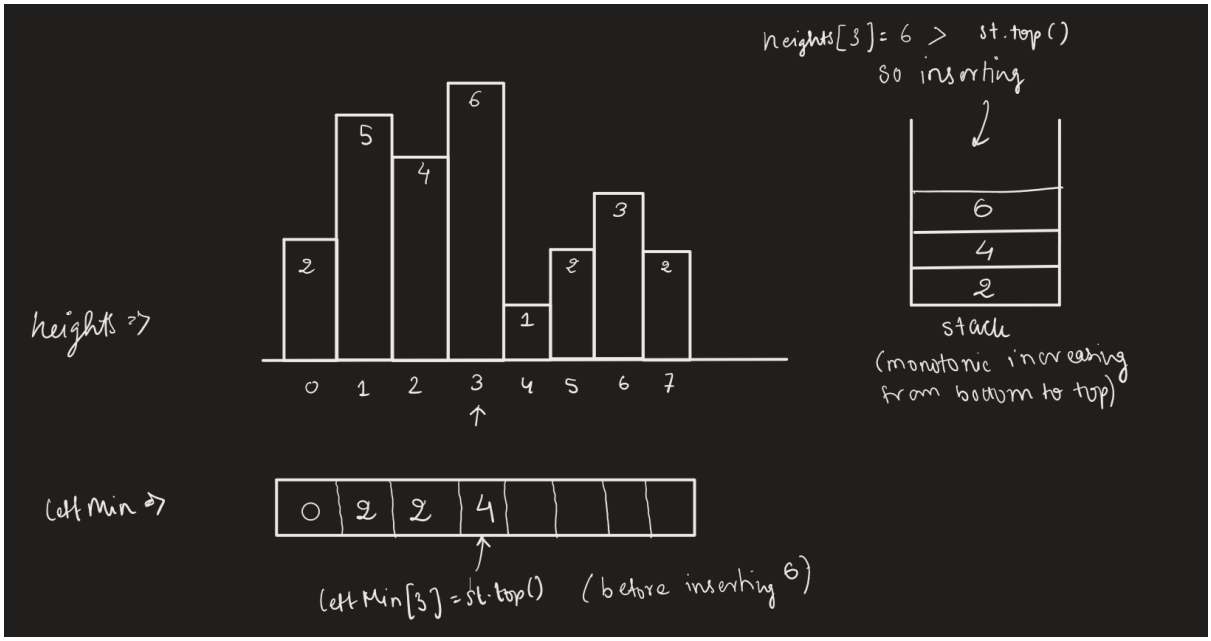
(ii) Optimal Approach - Stack with leftMin and rightMin

We can use a monotonic stack in order to store the left smaller and right smaller element for every heights[i] in two additional arrays leftMin and rightMin.









→ If we keep on following this process, we will get the complete leftMin array (array containing smaller height on the left/before, for height at every current index).

→ In a similar way, we can traverse from the right end (back) of the array and maintain a monotonic increasing stack (from bottom to top), in order to obtain the rightMin array (array containing smaller height on the right/after, for height at every current index).

NOTE: Here I have taken the heights value in the stack for easier understanding, but actually we will have to store the indices in the stack and the leftMin and rightMin arrays.

→ Finally we can traverse in the heights array and for every index the max area including the height at that index will be = **(leftMin[i] - rightMin[i] - 1) * heights[i]**.

Here, leftMin[i] = index of the height smaller than heights[i] on the left

rightMin[i] = index of the height smaller than heights[i] on its right

→ Now at every index, we have to update our maximum area (say **maxArea**) with the area calculated above utilizing the current heights[i]. maxArea value at the last is our final answer.

CODE

```
class Solution {
public:
    int largestRectangleArea(vector<int>& heights) {
        stack<int> st;
        vector<int> leftMin(heights.size(),-1), rightMin(heights.size(), heights.size());
        //filling leftMin by traversing from left end of array:-
        leftMin[0] = -1;
        st.push(0);
        for(int i = 1; i<heights.size(); ++i)
        {
            while(!st.empty() && heights[st.top()]>=heights[i]) st.pop();
            if(!st.empty()) leftMin[i] = st.top();
            st.push(i);
        }
        while(!st.empty()) st.pop();//emptying stack for reusing

        //filling rightMin by traversing from right end of array:-
        rightMin[heights.size()-1] = heights.size();
        st.push(heights.size()-1);
        for(int i = heights.size()-1; i>=0; --i)
        {
            while(!st.empty() && heights[st.top()]>=heights[i]) st.pop();
            if(!st.empty()) rightMin[i] = st.top();
            st.push(i);
        }
        //final traversal for maxArea calculation and updation:-
        int maxArea = 0;
        for(int i = 0; i<heights.size(); ++i)
        {
            cout<<heights[i]<<"-> "<<leftMin[i]<<" and "<<rightMin[i]<<endl;
            maxArea = max(maxArea, heights[i]*(rightMin[i]-leftMin[i]-1));
        }
        return maxArea;
    }
};
```

Time Complexity: O(N) for leftMin + O(N) for rightMin + O(N) for traversal and area updation

Space Complexity: O(N) for stack + O(N) for leftMin + O(N) for rightMIn

[LeetCode shows TLE for this solution on submitting! But you can test it on custom inputs]

(iii) Most Optimal Approach - Stack with Two Pointers

→ If you noticed a pattern in the (ii) approach, for every height at an index, the index in st.top() would always finally be containing the left smaller height only for any height[i].

→ Now, let's say we encounter a heights[i] that is smaller than heights[st.top()]. This would mean that for heights[st.top()], this heights[i] is the right smaller height. So we have obtained info about both left smaller height and right smaller height indices. Let's calculate the area using the formula mentioned above in (ii) approach, and keep updating the maxArea.

Edge case 1: What if there is no left smaller height? **Soln:** We consider the left boundary of array (imaginary) as the left smaller height. So we consider the width from current index to left/starting end (0 index) of the array.

CODE

```
class Solution {
public:
```

```
    //INTUITION: For any hegight in the array, the max area that can it can cause
//will be between its right smaller height and left smaller height...THINK!
    int largestRectangleArea(vector<int>& heights) {
        //A less optimal solution will be if we create two different arrays leftSmaller...
//...(where we store left smaller boundary for every index) and rightSmaller...
//...(where we store right smaller boundary for evry index)

        //ALGO: Monotonic stack (maintaining increasing order from bottom to top)

        //MOST OPTIMAL TC:O(N)+O(N) SC:O(N) Single Pass Approach:-
        stack<int> st;
        int maxArea = 0;

        for(int i = 0; i<=heights.size(); i++)
        {
            while(!st.empty() && ( i==heights.size() || heights[st.top()]>=heights[i] )) //the i==heights.size() condition
//is for the edge case when the input array is like [1,2,3,4,5]
            {
                int height = heights[st.top()]; //for this height at index = st.top(), heights[i]...
//...at ith index(i.e current element) will act as the right smaller(right boundary),...
//...and the next st.top() will act as the left boundary for the area
                st.pop();
                int width;
                if(st.empty()) width = i - 0; //from starting of array, if stack is empty left boundary...
//...would be starting of the array
                else width = i-st.top()-1; //the width between the left boundary and right boundary

                maxArea = max(maxArea, height*width); //updating area
            }
            st.push(i);
        }
        return maxArea;
    }
};
```

Time Complexity: O(N) for traversal (i.e. it is a one-pass algo)

Space Complexity: O(N) for stack

THAT’S ALL, ENJOY :)