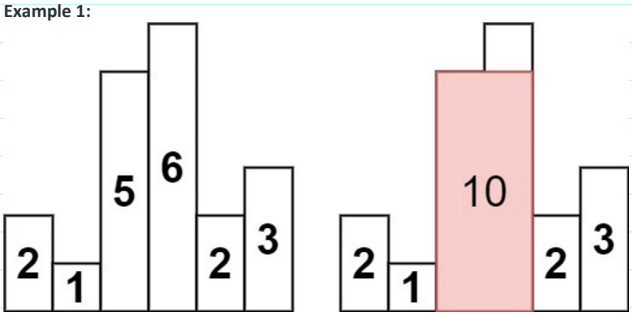
SQP10 Largest Rectangle in Histogram

Saturday, December 3, 2022 12:00 PM

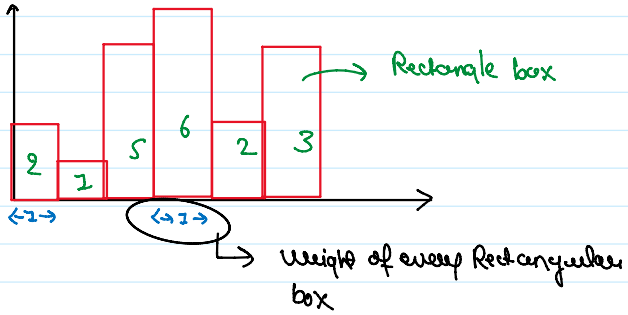
Given an array of integers heights representing the histogram's bar height where the width of each bar is 1, return *the area of the largest rectangle in the histogram*.

Example 1:



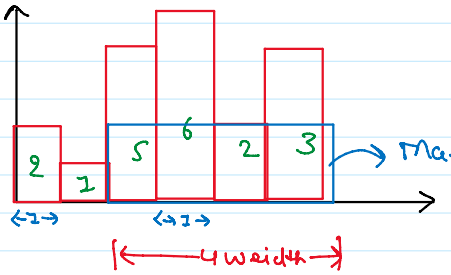
Input: heights = [2,1,5,6,2,3]
Output: 10
Explanation: The above is a histogram where width of each bar is 1.
The largest rectangle is shown in the red area, which has an area = 10 units.

Example



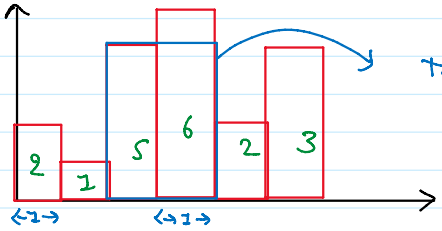
Maximum Combination case possible we finding the Maximum answer

Rectangle 1



2x4 = (8) ↑↑ as Much as Max

Rectangle 2



5x2 = (10) ✓

↳ that is the My Maximum answer of this problem

Many More Combination of Rectangle possible but we finding the Maximum One

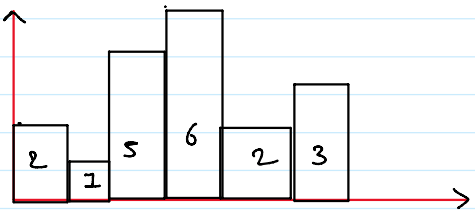
Example 2

That is the Maximum area of this problem = $2 \times 3 = 6$ Maximum

Thought Process

Always in Interview Start with Brute Force Approach and then go to the Optimal Approach

→ First we finding the seperately Area for every particular Rectangular box that are positive for left and right hand side



Now finding the Area for every Particular Rectangular box

2 → 2 covers only one box because next box is Smaller than 2 So My Area is

Area = $2 \times 1 = 2$

1 → 1 Covers all the Rectangular box because (1) is Smallest unit in this Histogram so Area is

Area = $1 \times 6 = 6$

5 → 5 Covers only two Rectangular box → My Maximum Answer

Area = $2 \times 5 = 10$

6 → 6 Covers only 1 Rectangular box because that is biggest one in Histogram

Area = $6 \times 1 = 6$

2 → 2 covers 4 Rectangular box So the area is

Area = $2 \times 4 = 8$

3 → 3 Covers only 1 Rectangular box

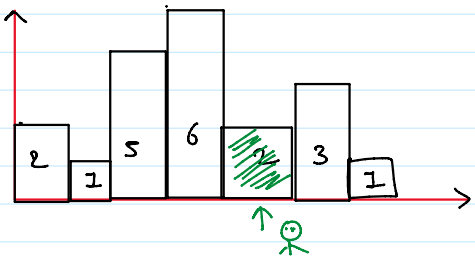
Area = $3 \times 1 = 3$

Area = 3x1 = (3)

That is the way how to finding the every particular Rectangular box Area

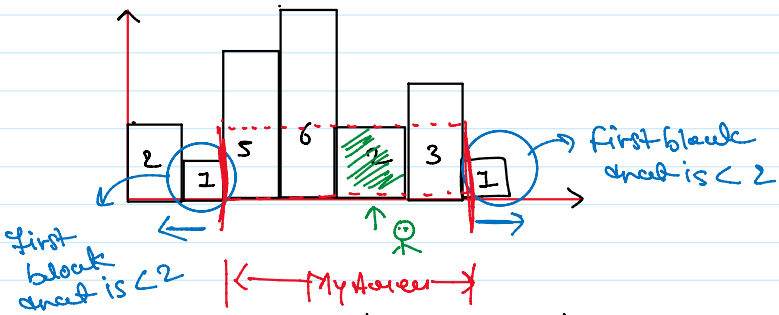
Maximum Ans = (10)

→ Now the Question is How to find the left and Right all the Rectangular box that Comes under the particular Rectangular box



→ Assume I am standing on box = 2 So How to finding the All the box that area Comes Under that Range

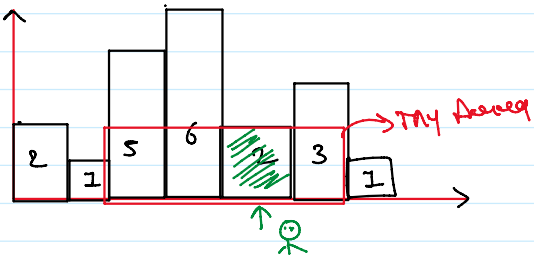
- → for Right hand side
  - → finding the First block that is < Standing block = 2
- → for left hand side
  - → same Approach finding the first block that is < Standing block = 2



↳ And that is My Width of that particular Rectangular Box

2x4 = (8)

finding the Area Using Index



Index = 0 1 (2) 3 4 (5) 6

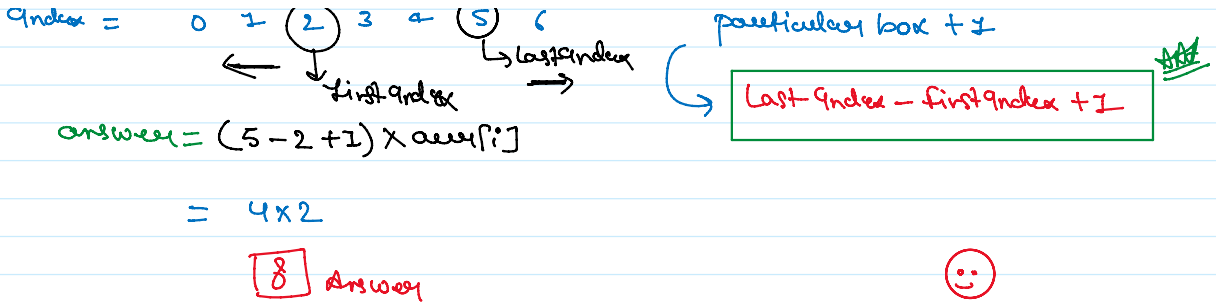
↳ last index

↳ first index

formula's –

last index and first index of first Smaller box for particular box + 1

(last index – first index + 1



→ That is the way for finding every arr[i] finding the Area

for(0,n) → that for loop for linear iteration for every Rectangular box

for(i,n)

for finding the Right hand side first < box index →

for(0,i)

→ for finding the left hand side first < box index ←

Time Complexity = O(N²)

→ Because we used Nested loop

Implementation

```
class Solution:
    def largestRectangleArea(self, h):

        n = len(h)
        ans = 0

        for i in range(n):

            right = 0
            left = 0

            for j in range(i, n):
                if h[j] >= h[i]:
                    right = j
                else:
                    break

            for k in range(i, -1, -1):
                if h[k] >= h[i]:
                    left = k
                else:
                    break

            ans = max(ans, (right - left + 1) * h[i])

        return ans
```

Optimized Approach

During the Bruteforce Approach we finding the formula that is

(lastIndex – firstIndex) × arr[i] + 1

During the Brute force Approach we finding the formula that is

(last index − first index) × arr[i] + 1

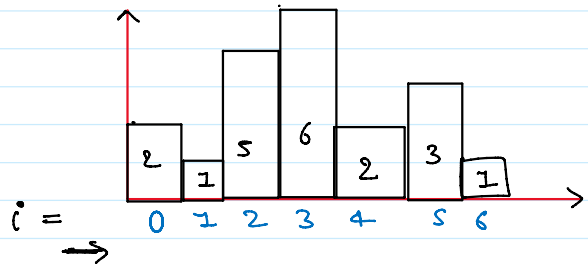
→ So how can I finding the Right Smaller index and Left Smaller index
→ Here we used the concept that we learn previously that is Next Greatest element and on the bases of this concept we finding the

- Next Smaller element
- Previously Smaller element

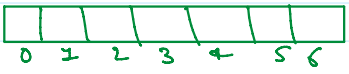
→ We creating and Left Small index and right Small index array of the size of histogram

→ Maintain the Stack data structure here carrying the index of the Rectangular box

#Lets Understand



→ Initialized the Array

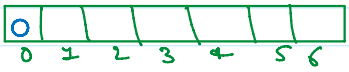
Left Small index = 

Stack = 

Start the iteration from the left hand side →

i = 0
Look into the stack and check if any greater element that is ≥ then arr[i] ✗
(No Greater element)

- So put 0 into the Left index array
- and put that index into the stack

Left Small index = 

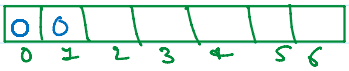
Stack =

i = 1

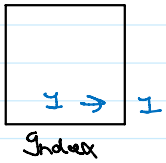
Look into the stack and check if any greater element that is ⩾ than arr[i] ✓
(Yes present)

→ So Remove that Index from the stack
→ and assign the previous value into the Left Index Array

LeftSmall Index =



Stack =

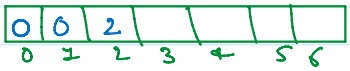


i = 2

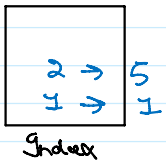
Look into the stack and check if any greater element that is ⩾ than arr[i] ✗
(No Greater element)

→ In this case what ever Index at the top of Stack + 1 push into the LeftIndex array
→ and push i into the Stack

LeftSmall Index =



Stack =

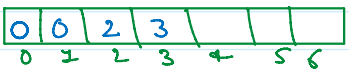


i = 3

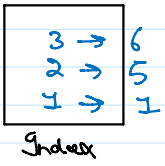
Look into the stack and check if any greater element that is ⩾ than arr[i] ✗
(No Greater element)

→ In this case what ever Index at the top of Stack + 1 push into the LeftIndex array
→ and push i into the Stack

LeftSmall Index =



Stack =



→ if we observed that we push that Index into the increasing order

Index

→ If we observed then we put that index into the increasing order

$i = 4$

Look into the stack and check if any greater element that is > then arr[i] ✓

(Yes present)

→ So Removed all the values of index that is ≥ i from the stack

→ and assign the Stack.pop() + 1 into the left index array and put 4 into the stack

Left Small Index =

0	1	2	3	4	5	6
0	0	2	3	2		

Stack =

3 →	6 → Removed
2 →	5 → Removed
1 →	1

Index

Stack =

4 →	2
1 →	1

$i = 5$

Look into the stack and check if any greater element that is ≥ then arr[i] ✗

(No Greater element)

→ In this case what ever index at the top of stack + 1 put into the left index array

→ and put i into the stack

Left Small Index =

0	1	2	3	4	5	6
0	0	2	3	2	5	

Stack =

5 →	3
4 →	2
1 →	1

Index

$i = 4$

Look into the stack and check if any greater element that is > then arr[i] ✓

(Yes present)

→ So Removed all the values of index that is ≥ i from the stack

→ and assign the Stack.pop() + 1 into the left index array and put 4 into the stack

Left Small Index =

0	1	2	3	4	5	6
0	0	2	3	2	5	

Stack =

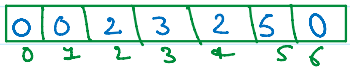
5 →	3 Removed
4 →	2 Removed
1 →	1 Removed

Index

Index

Stack = 

→ If stack becomes Empty then simply push the 0 into the Left index array

Left Small Index = 

→ That is the way How to finding the Left index of all Rectangular box

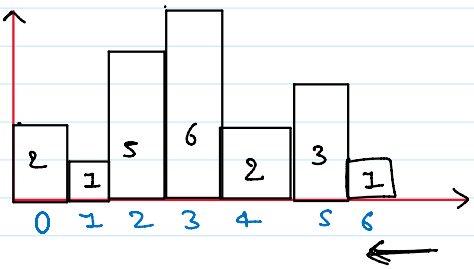
→ Similary process follow for finding the Right Small Index only Some changes Apply

→ Start Iteration from right side ←

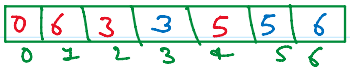
→ On Left Small Index we push stack.pop()+1 into the array

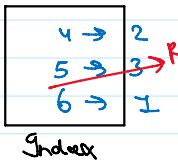
→ For Right Small Index we push stack.pop() −1 into the array

→ Now Remaining Concept and logic will be same that we follow into the Left Small Array

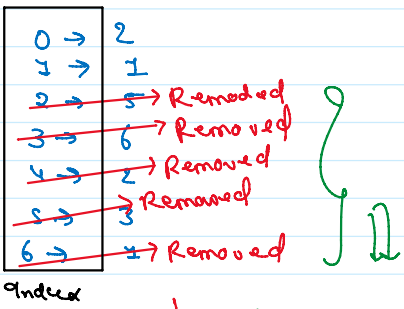
i = 

→ Initialized the Array

Right Small Index = 

Stack = 

Start the iteration from the Right hand side ←

Stack 

→ If stack if empty so push last index that is present into the stack that is = 6 and push into the Right Index array and push 9 into

next is != 6 and push into the Right-index array and pop 9 into stack

That is the way to finding the Left and Right small index array

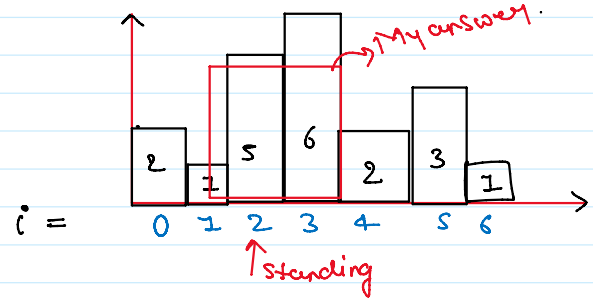
Left small index = | 0 | 0 | 2 | 3 | 2 | 5 | 0 |
0 1 2 3 4 5 6

Right small index = | 0 | 6 | 3 | 3 | 5 | 5 | 6 |
0 1 2 3 4 5 6

after finding the Left and right small index array we traverse linearly in our histogram and apply formula

(Right − left index) + 1 × arr[i]

→ and finding the our Maximum answer



Left small index = | 0 | 0 | 2 | 3 | 2 | 5 | 0 |
0 1 2 3 4 5 6

Right small index = | 0 | 6 | 3 | 3 | 5 | 5 | 6 |
0 1 2 3 4 5 6

example

(Right index − Left index + 1) × arr[i]

(3 − 2 + 1) × arr[2]

3 × 5 = 10

✓ 10 that is my Maximum Answer

Time complexity = O(N) + O(N) + O(N) ≈ O(N)
→ for left index Array
→ for Right index Array
→ for finding the Maximum answer

Space complexity = O(N) + O(N) + O(N) ≈ O(3N)
→ for left index Array
→ for Right index array
→ for Stack

#Implementation Python

```
# lc: https://leetcode.com/problems/largest-rectangle-in-histogram/
# gfg: https://practice.geeksforgeeks.org/problems/maximum-rectangular-
```

#Implementation Java

```
import java.util.ArrayList;
import java.util.Stack;
```

```
# lc: https://leetcode.com/problems/largest-rectangle-in-histogram/
# gfg: https://practice.geeksforgeeks.org/problems/maximum-rectangular-area-in-a-histogram-1587115620/1

class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:

        n = len(heights)
        leftSmallIndexArray = [-1] * n
        rightSmallIndexArray = [-1] * n

        stack = []

        for i in range(n):
            while len(stack) != 0 and heights[stack[-1]] >= heights[i]:
                stack.pop()

            if len(stack) == 0:
                leftSmallIndexArray[i] = 0
            else:
                leftSmallIndexArray[i] = stack[-1] + 1
            stack.append(i)

        stack.clear()

        for i in range(n - 1, -1, -1):
            while len(stack) != 0 and heights[stack[-1]] >= heights[i]:
                stack.pop()

            if len(stack) == 0:
                rightSmallIndexArray[i] = n - 1
            else:
                rightSmallIndexArray[i] = stack[-1] - 1
            stack.append(i)

        MaxArea = 0

        for i in range(n):
            MaxArea = max((rightSmallIndexArray[i] - leftSmallIndexArray[i] + 1) * heights[i], MaxArea)

        return MaxArea
```

More Optimized Approach

As we previously learned a formula for finding area of every particular Rectangular box

(Right - left) + 1 X arr[i]

↳ this formula take (2) passes one for RightSmall index and one for leftSmall index ☺

→ Now in this Approach every height is Covered Convert 2 pass to 1 pass

Thought Process

```
import java.util.ArrayList;
import java.util.Stack;

public class L9_Largest_Rectangle_in_Histogram {
    public static void main(String[] args) {

        System.out.println("L9
_Largest_Rectangle_in_Histogram");
    }
}

class Solution {
    public int largestRectangleArea(int[] heights) {

        int n = heights.length;
        int[] leftSmallIndexArray = new int[n];
        int[] rightSmallIndexArray = new int[n];


        Stack<Integer> stack = new Stack();

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

            while(!stack.isEmpty() && heights[stack.peek()] >=
heights[i]){
                stack.pop();
            }
            if(stack.isEmpty()){
                leftSmallIndexArray[i] = 0;
            }
            else{
                leftSmallIndexArray[i] = stack.peek()+1;
            }
            stack.push(i);

        }
        while(!stack.isEmpty()){
            stack.pop();
        }

        for (int i = n-1; i >= 0; i--) {

            while(!stack.isEmpty() && heights[stack.peek()] >=
heights[i]){
                stack.pop();
            }
            if(stack.isEmpty()){
                rightSmallIndexArray[i] = n-1;
            }
            else{
                rightSmallIndexArray[i] = stack.peek()-1;
            }
            stack.push(i);

        }

        int MaxArea = 0;
        for (int i = 0; i < n; i++) {
            MaxArea =
Math.max(MaxArea,(rightSmallIndexArray[i] -
leftSmallIndexArray[i] + 1) * heights[i]);

        }

        return MaxArea;
```

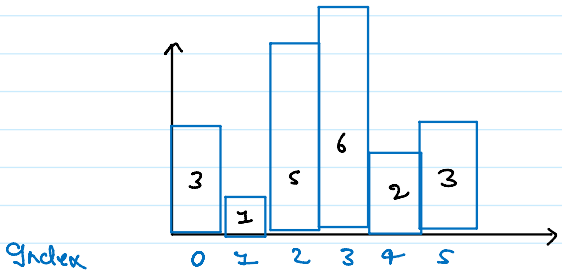
```
        }

        return MaxArea;

    }
}
```

Thought Process

→ Previously we learned to finding the Left and Right Index and store the Index into Stack

→ In this Approach we Not finding the hole left and Right every left and right we finding the Area as well as that is my intuitions

Let's Dry Run and Understand



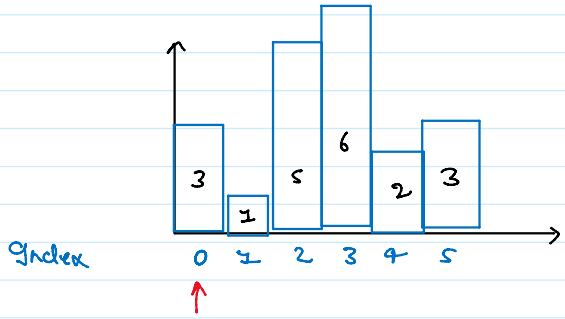
Stack =

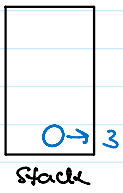
MaxArea = 0

→ Now to finding the left and Right Index at single pass

→ We Create the MaxArea

→ Start Iteration from 0 Index





i=0 Our Stack is Empty so Simply put the Index into the Stack

i=1

Now Check if any element that is > then arr[i] into the Stack ✓

(Yes exist)

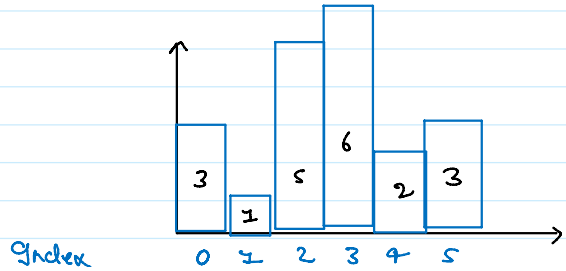
→ Removed that element and i is the Right Index

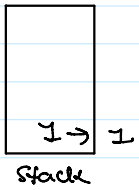
→ And if our stack if empty that means Left Index is = 0

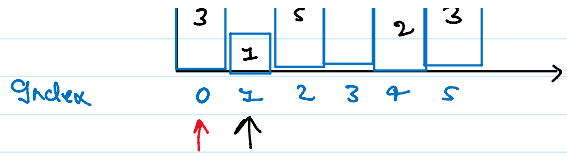
→ In this case My Overall width is = 1

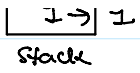
MaxArea = 3 × 1 = (3) → update the max area and put i into the stack

MaxArea = ~~0~~ 3





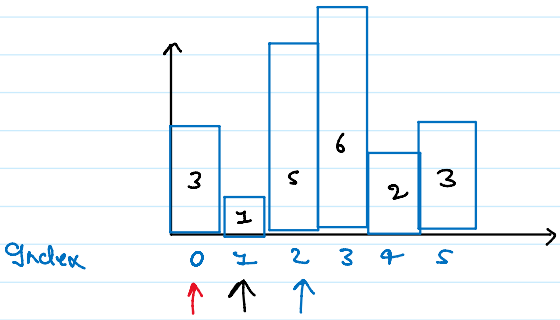


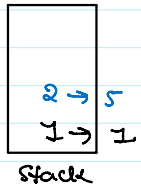


i = 2

Now Again Check if element next is > then arr[i] So again same process follow other wise put next i into the stack

MaxArea = ~~0~~ 3

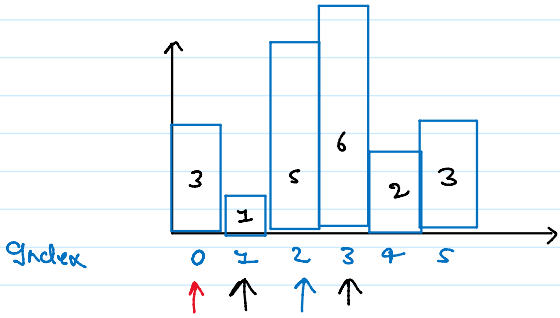


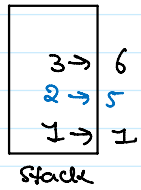


i = 3

Now Again Check if element next is > then arr[i] So again same process follow other wise put next i into the stack

MaxArea = ~~0~~ 3





i = 4

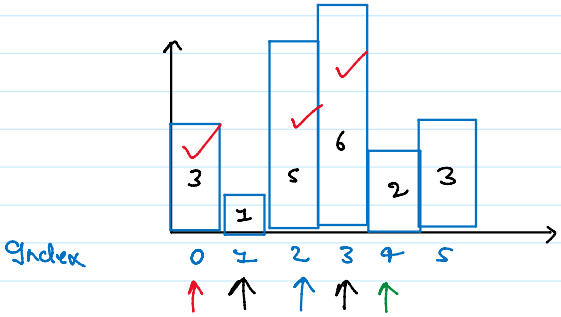
Now again check if any element present next is > arr[i] into the stack (Yes exist)

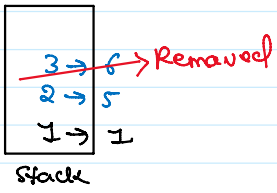
✓

→ Now compute every blocks of width

→ and removed from the stack

MaxArea = ~~0~~ ~~3~~ 6





Right Index = 4

Left Index = previous of the top of the Stack

= 2

MaxArea = Right − left − 1 × arr[i]

= 4 − 2 − 1 × 6

= (6)

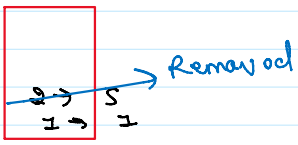
→ Again Check if any element next is > then arr[i] So again same

MaxArea = ~~0~~ ~~3~~ ~~6~~ 10

= (6)

→ Again check if any element next is > than arr[i] so again same process follow

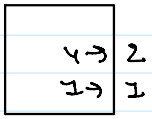
MaxArea = ~~0~~ ~~3~~ ~~6~~ 10

Stack = 

Right index = 4
left index = 1

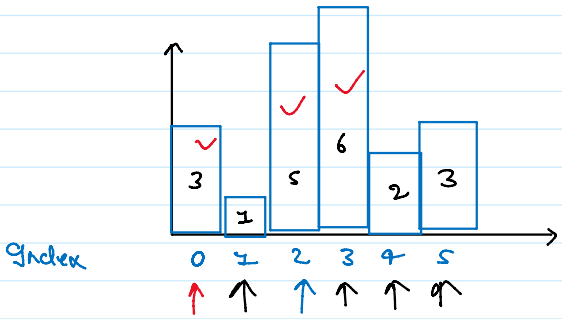
MaxArea = Right − left − 1 × arr[i]
MaxArea = 4 − 1 − 1 × 5
= (10)

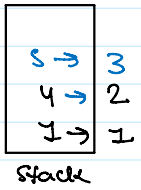
→ Again check if any element next is > than arr[i] into the stack ✗
↳ Not any element exist next is > than arr[i]
So simply push i into the stack

Stack = 

i = 5
Now Again check if element next is > than arr[i] so again same process follow other wise push next i into the stack

MaxArea = ~~0~~ ~~3~~ ~~6~~ 10

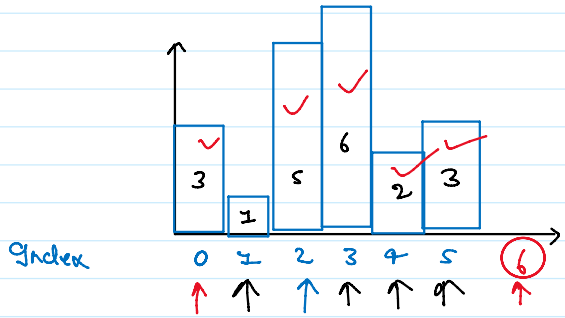


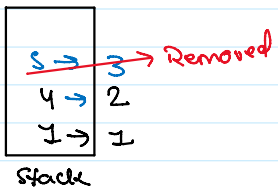


→ Now hence My operation will be end but my (3) block code Not calculated what Can I do

↳ Thus we Moved one More operation i = 6
and again same process follow and keep to compute all the Remaing box Areas.

MaxArea = ~~0~~ ~~3~~ ~~6~~ 10

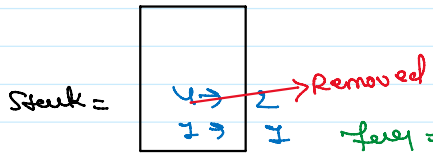




Now 3
Right index = 6
left index = 4

→ Again same process till we see less Removing element into the Stack

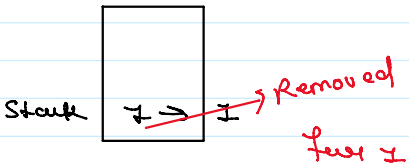
$MaxArea = (6-4-1) \times arr(i)$
$= 1 \times 3$
$= 3$ ☺



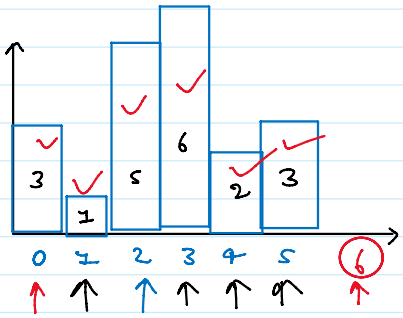
less = 2
RightIndex = 6
LeftIndex = 1

→ Again same process till we see less Removing element into the Stack

$MaxArea = (6-1-1) \times arr(i)$
$= 4 \times 2$
$= 8$



less 1
RightIndex = 6
leftIndex = 0



$MaxArea = (6-0) \times arr(i)$
$= 5 \times 1$
$= 6$

↳ if only Remaining element is 1 into to the stack or 0 then this case we can not -1 ☺

Time Complexity = $O(N) + O(N)$
→ less linearly iterate
→ less maintaining the Stack
→ every time we not Removed the every element from the Stack

Space Complexity = $O(N)$
→ Only less Stack No Need to acquired another data structures ☺

Implementation Python

Implementation Java

```python
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:

        stack = []
        MaxAns = 0
        n = len(heights)

        for i in range(n+1):

            while len(stack) != 0 and (i == n or heights[stack[-1]] >= heights[i]):
                height = heights[stack[-1]]
                stack.pop()
                width = 0

                if len(stack) == 0:
                    width = i
                else:
                    width = i - stack[-1]-1

                MaxAns = max(MaxAns,width*height)

            stack.append(i)

        return MaxAns
```

```java
class Solution {
    public int largestRectangleArea(int[] heights) {

        int MaxAns = 0;
        int n = heights.length;

        Stack<Integer> stack = new Stack();

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

            while(!stack.isEmpty() && (i == n || heights[stack.peek()] >= heights[i])){
                int height = heights[stack.peek()];
                stack.pop();
                int weidth;

                if(stack.size() == 0){
                    weidth = i;
                }
                else{
                    weidth = i - stack.peek()-1;
                }

                MaxAns = Math.max(MaxAns,weidth * height);
            }
            stack.push(i);
        }


        return MaxAns;
    }
}
```