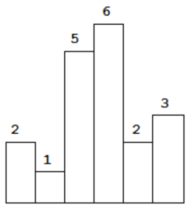
Leet Code 84. Largest Rectangle in Histogram — Graphically Explained Python3 Solution

- January 01, 2021

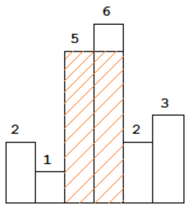
Photo by Pablo Hermoso on Unsplash

Problem Description

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
```

Solution

Obvious Solution

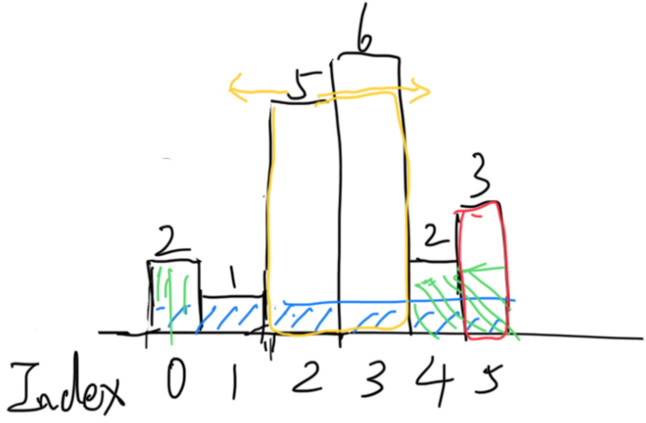
The obvious (and brute force) way is to use 2 pointers: left and right. Let left points to index 0 ~ N-1 and then right to left and N-1. For each left and right pair, I can calculate the area[left, right] using (right-left+1) * min(heights[left], heights[left+1],heights[left+2],...,heights[right]). Its time complexity is at least be $O(N^2)$.

Best Solution (to me)

[Credit] This is described by @TravellingSalesman in disucssion area (https://leetcode.com/problems/largest-rectangle-in-histogram/solution/) and here I just try to illustrate it further.

Take the example of heights array [2,1,5,6,2,3], each area must be in the height of any element within heights. In other words, the maximum area will either be an area with height=2, or an area with height=1, or an area with height=5 and etc. So instead of using 2 pointers like mentioned in above section, here I can use height central perspective: if I want to use heights[2] (=5) as the height of a candidate area, how to get its width? Not surprisingly, I need to look to its left to find the first (leftmost) height (its index referenced as **L** in following) that is smaller than 5 and rightmost height (its indexreferenced as **R** in following) that is smaller than 5: yellow arrows in below graph (Pic 1) illustrate the boundaries. With that the width=(R-L)-1=4–1–1=2 and hence the area = width * height = 2 * 5 = 10 (as shown in yellow rectangle).

What if a height is the samllest one so there is no valid L, R? For example, for heights[1]=1, obviously the area of heights[1]=1 will has the width=6 (the length of the heights array). Logically, I can assume L = -1 while R=size-of-heights. Similar logic applies to an element that seems has only a valid L or R: for example, red rectangle indicates an area with the heights[5] and it's L=4 and R=6 so its width=R-L-1=6–4–1=1 and its area = 1 * 3=3.



Pic 1

There is a smat way to implement above algorithm: using a stack that stores ascending elements. When I iterate the array, if the top of the stack (as referenced as stack[-1] in Python) is larger than current element, for **the one on the top of the stack**, we actually find its R; its L is actually the second to the top: with this, I can conclude the area with the height corresponding to the top of the stack; if the top of the stack (as referenced as stack[-1] in Python) is smaller or equal than current element, that means I still don't know what's the R for current top element in the stack, so I will just push current element to the stack.

Sounds dizzy? Let me walk through the algorithm with the example [2,1,5,6,2,3]. To simplify writing, I use Area[i] to denote the area whose height is height[i].

0, first I run into index=0 (height=2). It's the first one and I know its L = -1 but am not sure the value of R, so I just put its index (so that later I can get its index for width calculation and in the same time get its value for height through heights[index]) in the stack. Now stack=[0].

1, now I come to index=1(height=1). It's less than current top of stack's value since heights[1] < heights[stack[-1]]. So I find heights[stack[-1]] (aka heights[0])'s R= current index. So for the area of heights[0] (illustrated by green vertical lines in Pic 1) is (R-L-1) * heights[0] = (1-(-1)-1)*2=2. **So I have Area[0] calculated.** I will pop up index 0 from stack since its area is calcuated; but for index 1, its corresponding area is not yet calculated (I don't know its R although its L is -1), so I push it to stack and now stack=[1].

2, now I come to index=2(height=5). Since heights[2] > heights[stack[-1]], for stack[-1], I cannot decide its R, just push current index to stack and makes stack=[1,2].

3, now I come to index=3(height=6). Same as #2 and now stack=[1,2,3].

4,now I come to index=4(height=2). Now heights[4] <heights[stack[-1]], it means for index stack[-1] (aka index 3), I find its R and therefore **Area[3]= (R-L-1)*heights[3]=(4-heights[stack[-2])*heights[3]=(4–2–1)*6=6**. Now I pop up the top of the stack and the stack=[1,2]. Notice stack[-1]=2 and heights[2] >heights[4], it indicates heights[2]'s R=4 as well. So similarly I can have **Area[2]= (R-L-1)*heights[2]=(4-heights[stack[-2])*heights[3]=(4–1–1)*5=10** and pop up the stack to get stack=[1]. Now stack[-1]=1 and heights[1] < heights[4] so heights[1]'s R is not yet decided and hence it stays at the stack. Now stack=[1,4].

5,now I come to index=5(height=3) and similar to #3 now stack=[1,4,5].

6, now I go beyond the array. Remember there is a default R=size-of-array=6? So for everyone remaining in the stack, its R is the index after it (except for the last one, it's 6) and its L is the index before it(except for the first one, it's -1). Bear in mind that heights array is [2,1,5,6,2,3] and I will have

Area[5]= (6–4–1) * height[5] = 3

Area[4]=(6–1–1) * height[4] = 8

Area[1]=(6-(-1)-1)* height[1] = 6

7, looking at Area[x] where x in 0~5, the largest one will be the answer.

Source Code

Slightly different from above walk-through, I add a "-1" to the stack in the beginning (serving as default **L**). In the end, an extra checking (corresponding to #6 in above) is performed.

```python
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        size  = len(heights)
        stack = [-1]
        maxA  = 0
        for i in range(size):
            #for each h, find it's L and R
            while (stack[-1] != -1 and heights[i] <= heights[stack[-1]]):
                #h is R for all elements in stack(except the first one: -1)
                    preI = stack.pop()
                    area = heights[preI] * (i - stack[-1] - 1)
                    maxA = max(maxA, area)
            stack.append(i)
        while (stack[-1] != -1):
            #h is R for all elements in stack(except the first one: -1)
            preI = stack.pop()
            area = heights[preI] * (size - stack[-1] - 1)
            maxA = max(maxA, area)
        return maxA
```

lc84.py hosted with ❤ by GitHub view raw

Time & Space Complexity

As illustrated above, I only need to travse the heights array once and when visiting an element, although there is a need to do while loop against the stack, each element will just be inside the stack for once. Therefore, the time complexity is O(N) and so is space complexity.

Extended Reading

Python3 cheatsheet:

https://medium.com/@edward.zhou/python3-cheatsheet-continuously-updated-66d652115549

Algorithm Interview Questions LeetCode Hard Python3 programming

Popular posts from this blog

Leet Code 1060. Missing Element in Sorted Array — Explained Python3 Solution

- June 17, 2020

Problem Description Given a sorted array A of unique numbers, find the K -th missing number starting from the leftmost number of the array. Example 1: Input: A = [4,7,9,10] , K = 1 Output: 5 Explanation: Th …

READ MORE

Leet Code 490. The Maze—Graphical Explained Python3 Solution

- August 23, 2020

Photo by Susan Yin on Unsplash Problem Description There is a ball in a maze with empty spaces and walls. The ball can go through empty spaces by rolling up , down , left or right , but it won't stop rolling until …

READ MORE

Powered by Blogger

Theme images by Michael Elkan