← Back to Home

Two Pointers
Sliding Window
Intervals
Stack
- Overview
- Valid Parentheses
- Decode String
- Longest Valid Parentheses

Monotonic Stack
- Overview
- Daily Temperatures
- Largest Rectangle in Histog...

Linked List
Binary Search
Heap
Depth-First Search
Breadth-First Search
Backtracking
Graphs
Dynamic Programming
Greedy Algorithms

← Daily Temperatures

Linked List: Overview →

Largest Rectangle in Histogram

DESCRIPTION (credit Leetcode.com)

Given an integer array `heights` representing the heights of histogram bars, write a function to find the largest rectangular area possible in a histogram, where each bar's width is `1`.

EXAMPLES

Inputs:

```
heights = [2,8,5,6,2,3]
```

Output:

```
15
```

```python
class Solution:
    def largestRectangleArea(self, heights: list[int]):
        # Your code goes here
        pass
```

Results AI Feedback View Answer Run

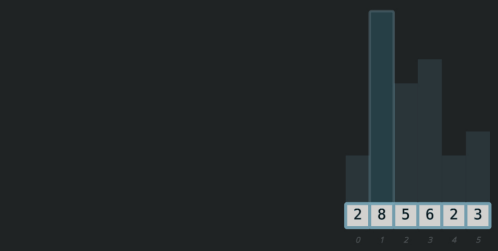
Run your code to see results here

Explanation

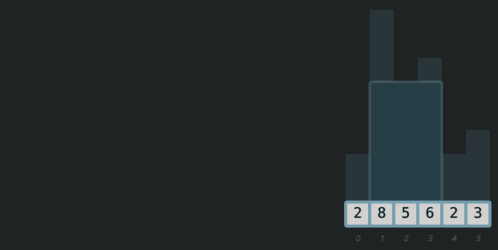
To solve this question, we calculate the largest rectangle that contains the bar at each index of the array, and return the largest of those rectangles at the end. By using a monotonically increasing stack, we can solve this question in $O(n)$ time complexity, compared to the brute-force solution which takes $O(n^2)$ time.

Largest Rectangle At Each Index

To calculate the largest rectangle at each index, we need to know the index of the first shorter bar to both the left and the right of the current bar. The width of the rectangle is the difference between the two indices - 1, and the height is the height of the current bar.



Index of first shorter bar to left: 0. Index of first shorter bar to right: 2. Total area: 8 * (2 - 0 - 1) = 8



Index of first shorter bar to left: 0. Index of first shorter bar to right: 4. Total area: 5 * (4 - 0 - 1) = 15

Brute-Force Solution

The brute-force solution iterates over each bar in the array, and for each bar, finds the first shortest bar to the left and the right of the bar using two while loops, for a time complexity of $O(n^2)$.

```
def largestRectangleArea(heights):
    max_area = 0
    n = len(heights)

    for i in range(n):
        left = i - 1
        while left >= 0 and heights[left] >= heights[i]:
            left -= 1

        right = i + 1
        while right < n and heights[right] >= heights[i]:
            right += 1

        max_area = max(max_area, (right - left - 1) * heights[i])

    return max_area
```

Monotonically Increasing Stack

By using a monotonically increasing stack, we can find the first shortest bar to the left and right of each bar in $O(n)$ time complexity.

We first initialize our stack, as well as a variable to store the maximum area and a pointer `i` to iterate over the array. The stack contains the indices of the bars in increasing order of their heights. When an index is placed on the stack, it means we are waiting to find a shorter bar to the right of that current index.

```
def largest_rectangle_area(heights):
  stack = []
  max_area = 0
  i = 0
  while i < len(heights):
    if not stack or heights[i] >= heights[stack[-1]]:
      stack.append(i)
      i += 1
    else:
      top = stack.pop()
      right = i - 1
      left = stack[-1] if stack else -1
      area = heights[top] * (right - left)
      max_area = max(max_area, area)

  while stack:
    top = stack.pop()
    width = i - stack[-1] - 1 if stack else i
    area = heights[top] * width
    max_area = max(max_area, area)
  return max_area
```

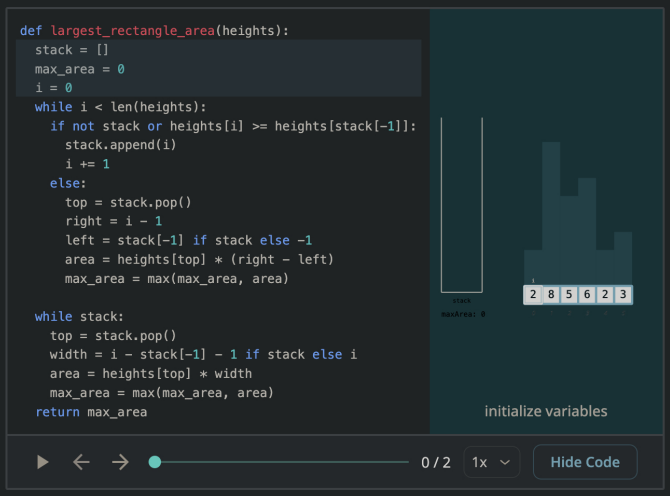
largest rectangle in histogram

0 / 1 1x Hide Code

We then iterate over the array. If `height[i]` is greater than the height of the index at the top of the stack, we push `i` onto the stack. If `i` is less than the height of index at the top of the stack, we pop the index at the top of the stack and calculate the area of the rectangle containing that index.

Pushing To The Stack

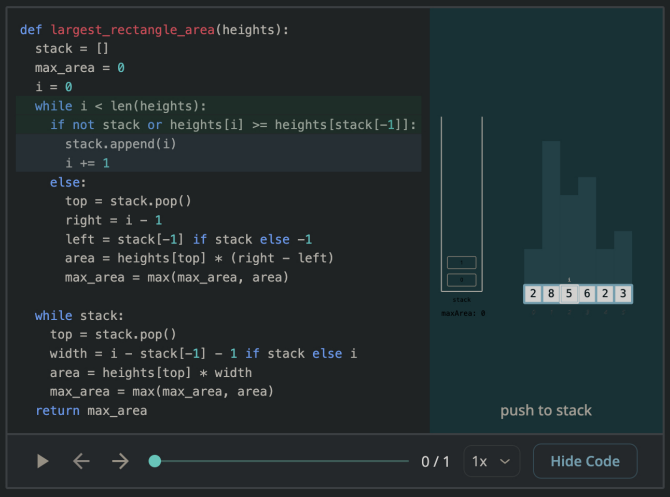
If `height[i]` is greater than the height of the index at the top of the stack (or if the stack is currently empty), we push `i` onto the stack. This means we are waiting to find a shorter bar to the right of `i`, and we increment `i`.



Pushing `i = 0` and `i = 1` to the stack

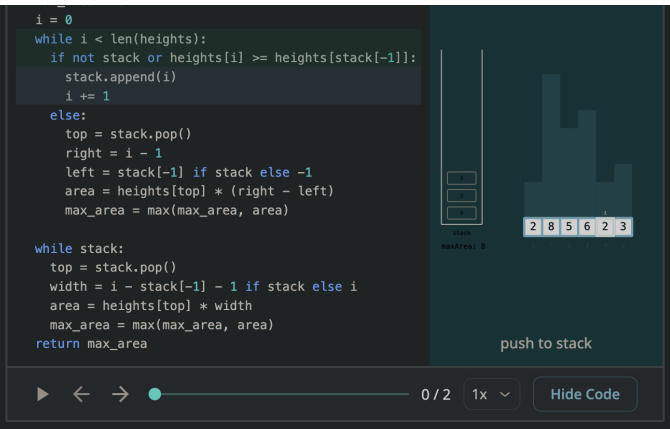
Popping From The Stack

If `height[i]` is less than the height of the index at the top of the stack, we have enough information to calculate the area of the rectangle containing the index at the top of the stack. `i` is the first shorter bar to the right of the index at the top of the stack. Because the stack is monotonically increasing, the index beneath the index at the top is the first shorter bar to the left of it (or -1 if the stack is empty).



At this point, `i = 2`, and the top of the stack is 1. `heights[2] < heights[1]` (5 < 8), so `i` is the right boundary for the rectangle at index 1. The left boundary is the index at the top of the stack after popping 1 from the stack (in this case, 0).

`i` still has the potential to be the right boundary for the new index at the top of the stack after the previous index was popped, so we don't increment `i`, and instead, move to the next iteration.

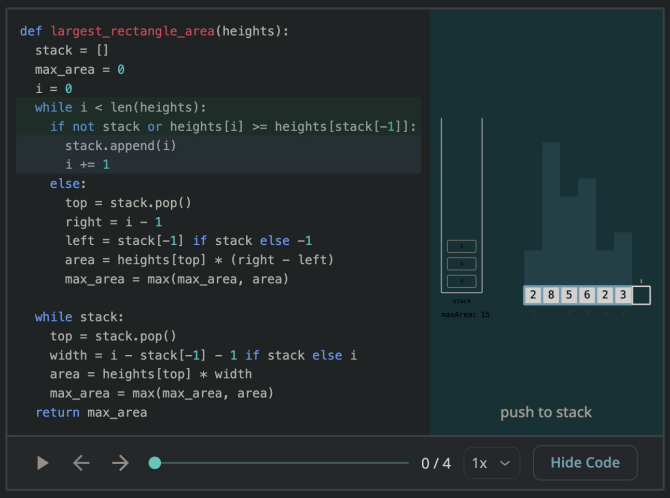


Another example. `i = 4`, and is the right boundary for both indexes 3 and 2.

Emptying The Stack

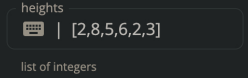
When `i` has iterated over the entire array, we still need to process the remaining indexes on the stack, which are the indexes that have not yet found a shorter bar to the right. So to calculate the area of the rectangle for these indexes, we use the end of the array as the right boundary, while the calculation for the left boundary remains the same.

When the stack is empty, we have processed all the indexes, and we return the maximum area.



Emptying the stack, and returning `maxArea`.

Solution



```
return max_area
```

histogram

0 / 14 1x Hide Code

Complexity Analysis

Time Complexity: `O(n)` . Each item is pushed and popped from the stack at most once.
Space Complexity: `O(n)` where `n` is the length of the input array for the size of the stack.

Next: Overview →

Add a comment...

Anonymous Post As Tariq Williams

XenophobicTomatoAnteater112
Created at: 7/31/2024, 03:32 PM
Please update the image caption under heading "Largest Rectangle at each Index" to Index of first "shorter" bar to left
Reply

LonelySalmonCanid112
Created at: 9/5/2024, 10:26 AM
How is this different from container with most water problem?
Reply

Jimmy Zhang
Created at: 9/5/2024, 01:20 PM
its very different.

in container with most water water can flow "above" the rectangles in between the two largest ones. they can't here.

try something like [6, 1, 1, 6] and compare the outputs

Reply