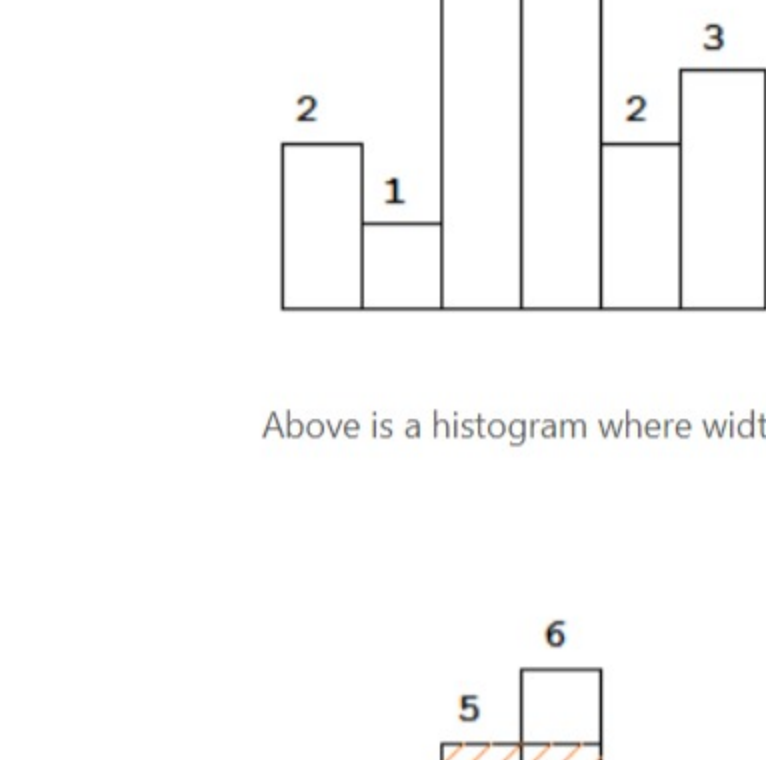


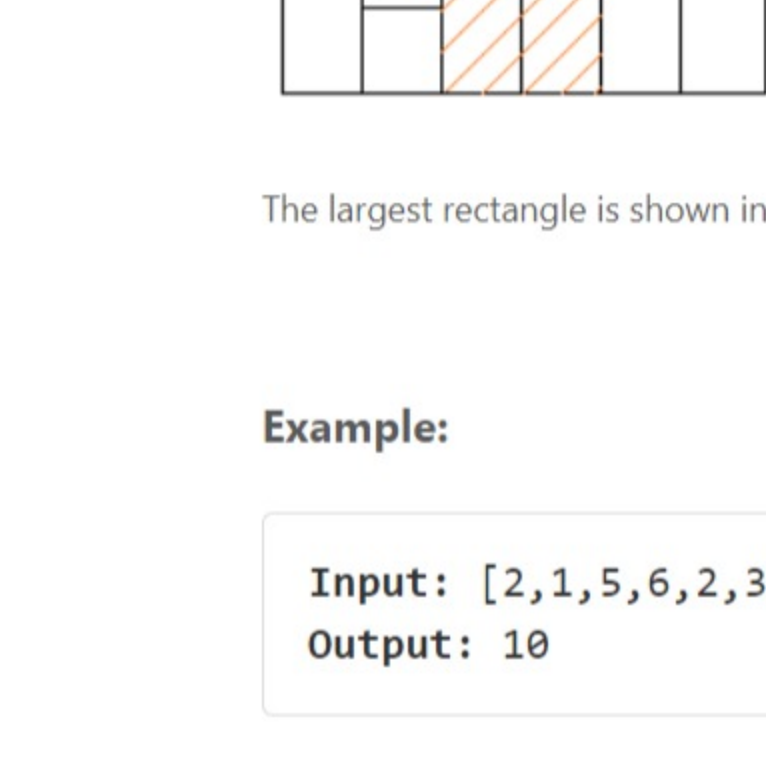
84. Largest Rectangle In Histogram

Dec. 7, 2016 | 61.9K views

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`

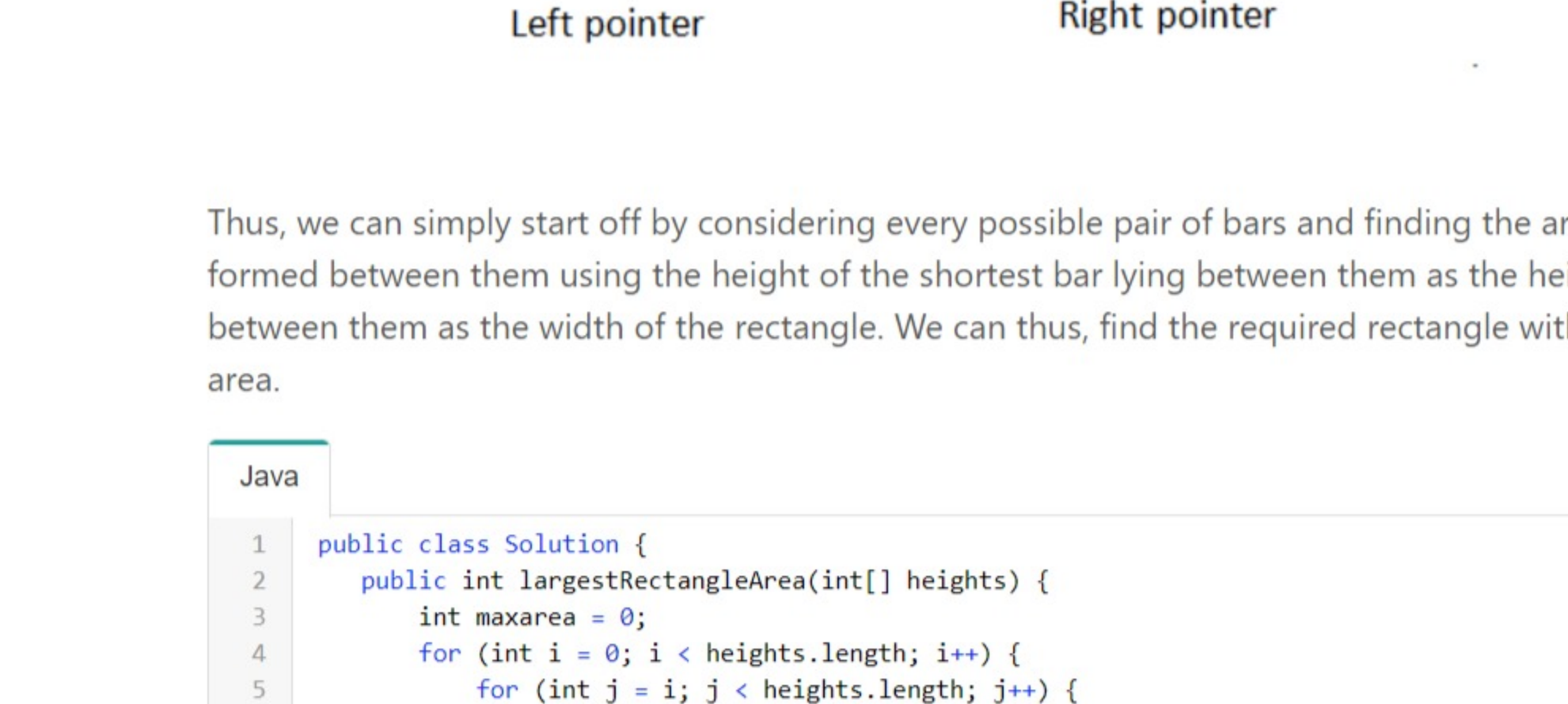
Summary

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

Solution

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.

JavaCopy

```
1 public class Solution {
2     public int largestRectangleArea(int[] heights) {
3         int maxarea = 0;
4         for (int i = 0; i < heights.length; i++) {
5             for (int j = i; j < heights.length; j++) {
6                 int minheight = Integer.MAX_VALUE;
7                 for (int k = i; k <= j; k++)
8                     minheight = Math.min(minheight, heights[k]);
9                 maxarea = Math.max(maxarea, minheight * (j - i + 1));
10            }
11        }
12        return maxarea;
13    }
14 }
15
```

Complexity Analysis

- Time complexity : $O(n^3)$. We have to find the minimum height bar $O(n)$ lying between every pair $O(n^2)$.
- Space complexity : $O(1)$. 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 height of the shortest bar 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.

In mathematical terms, $minheight = \min(minheight, heights(j))$, where $heights(j)$ refers to the height of the j th bar.

JavaCopy

```
1 public class Solution {
2     public int largestRectangleArea(int[] heights) {
3         int maxarea = 0;
4         for (int i = 0; i < heights.length; i++) {
5             int minheight = Integer.MAX_VALUE;
6             for (int j = i; j < heights.length; j++) {
7                 minheight = Math.min(minheight, heights[j]);
8                 maxarea = Math.max(maxarea, minheight * (j - i + 1));
9             }
10        }
11        return maxarea;
12    }
13 }
14
```

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

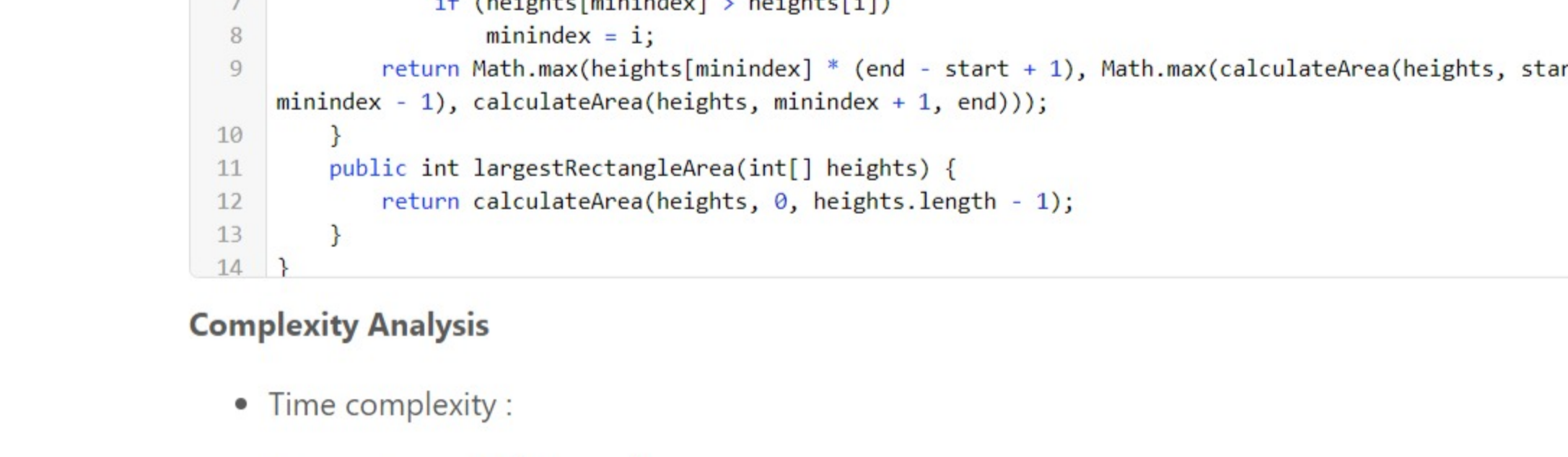
This approach relies on the observation that the rectangle with maximum area will be the maximum of:

- The widest possible rectangle with height equal to the height of the shortest bar.
- The largest rectangle confined to the left of the shortest bar(subproblem).
- 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 $2 \times 6 = 12$. 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 $4 \times 3 = 12$. Further, rectangles of area $6 \times 1 = 6$ and $5 \times 1 = 5$ exist in its left and right respectively. Similarly we find an area of $3 \times 3 = 9$, $4 \times 1 = 4$ and $9 \times 1 = 9$ to the left of 2. Thus, we get 16 as the correct maximum area. See the figure below for further clarification:



JavaCopy

```
1 public class Solution {
2     public int calculateArea(int[] heights, int start, int end) {
3         if (start > end)
4             return 0;
5         int minindex = start;
6         for (int i = start; i <= end; i++)
7             if (heights[minindex] > heights[i])
8                 minindex = i;
9         return Math.max(heights[minindex] * (end - start + 1), Math.max(calculateArea(heights, start, minindex - 1), calculateArea(heights, minindex + 1, end)));
10    }
11    public int largestRectangleArea(int[] heights) {
12        return calculateArea(heights, 0, heights.length - 1);
13    }
14 }
```

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 .

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 $O(n)$. Thus, the overall complexity becomes $O(n^2)$ 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 $O(\log n)$ time.

For implementation, click [here](#).

Complexity Analysis

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

Approach 5: Using Stack

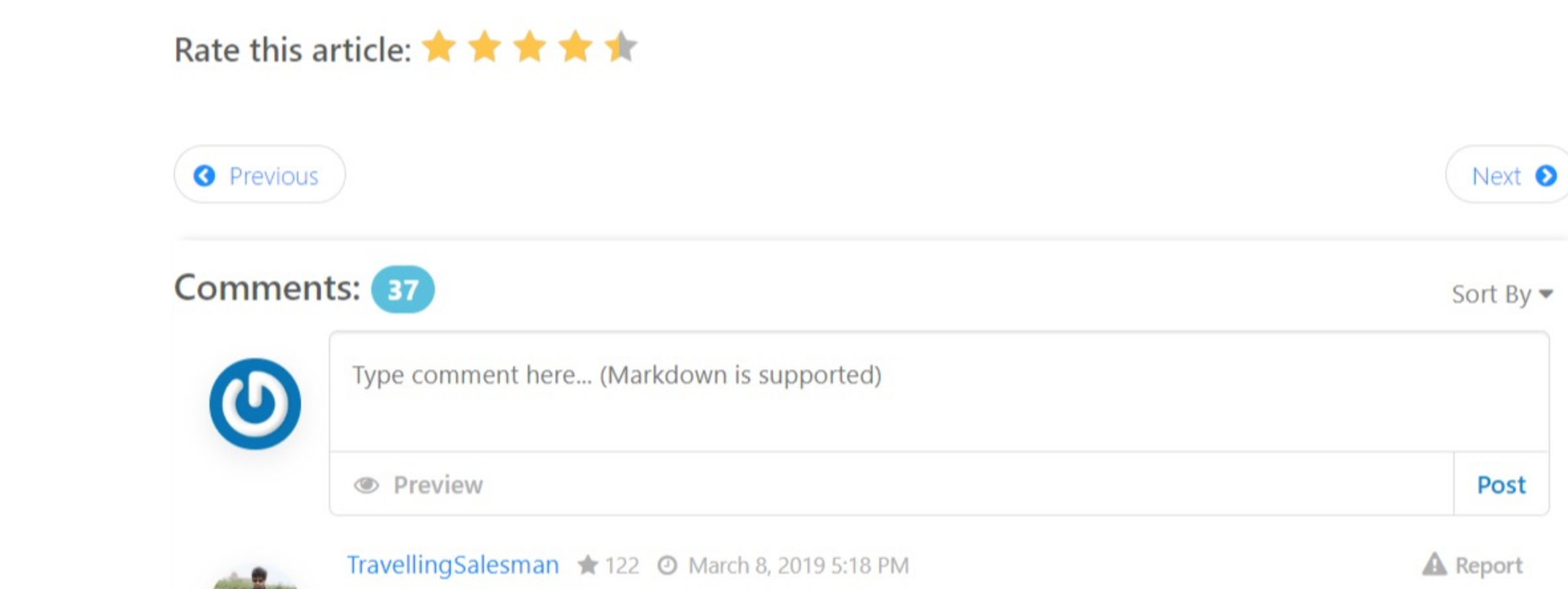
Algorithm

In this approach, we maintain a stack. Initially, we push a -1 onto the stack to mark the end. We start with the leftmost bar and keep pushing the current bar's index onto the stack until we get two successive numbers in descending order, i.e. until we get $a[i]$. Now, we start popping the numbers from the stack until we hit a number $stack[j]$ on the stack such that $a[stack[j]] \leq a[i]$. Every time we pop, we find out the area of rectangle formed using the current element as the height of the rectangle and the difference between the current element's index pointed to in the original array and the element $stack[top - 1] - 1$ as the width i.e. if we pop an element $stack[top]$ and i is the current index to which we are pointing in the original array, the current area of the rectangle will be considered as:

$$(i - stack[top - 1] - 1) \times a[stack[top]].$$

Further, if we reach the end of the array, we pop all the elements of the stack and at every pop, this time we use the following equation to find the area: $(stack[top] - stack[top - 1]) \times a[stack[top]]$, where $stack[top]$ refers to the element just popped. Thus, we can get the area of the largest rectangle by comparing the new area found everytime.

The following example will clarify the process further: `[6, 7, 5, 2, 4, 5, 9, 3]`



JavaCopy

```
1 public class Solution {
2     public int largestRectangleArea(int[] heights) {
3         Stack < Integer > stack = new Stack < > ();
4         stack.push(-1);
5         int maxarea = 0;
6         for (int i = 0; i < heights.length; ++i) {
7             while (stack.peek() != -1 && heights[stack.peek()] >= heights[i])
8                 maxarea = Math.max(maxarea, heights[stack.pop()] * (i - stack.peek() - 1));
9             stack.push(i);
10        }
11        while (stack.peek() != -1)
12            maxarea = Math.max(maxarea, heights[stack.pop()] * (heights.length - stack.peek() - 1));
13        return maxarea;
14    }
15 }
```

Complexity Analysis

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

Rate this article: ★★★★★

PreviousNext

Comments: 37Sort By

Type comment here... (Markdown is supported)

PreviewPost

TravellingSalesman★122March 8, 2019 5:18 PMReport

Let me try to give my thought process behind using stack, hope it helps:

- Idea is, we will consider every element $a[i]$ to be a candidate for the area calculation. That is, if $a[i]$ is the minimum element then what is the maximum area possible for all such rectangles? We can easily figure out that it's $a[i] * (R - L + 1 - 2)$ or $a[i] * (R - L - 1)$, where $a[R]$ is first subsequent

116ShareReply

SHOW 8 REPLIES

Kaiyubo★92November 14, 2018 2:52 AM

I think there is a typo in Divide and Conquer example. It should be $2 \times 7 = 14$.

70ShareReply

SHOW 1 REPLY

laaptu★42March 12, 2019 6:09 AMReport

This video <https://www.youtube.com/watch?v=RVlH0snn4Qc>, helped me understand the logic for solution of this problem using stack based approach.

Read More

33ShareReply

SHOW 3 REPLIES

go2ready★13February 22, 2018 1:43 PM

I guess there is a typo in the description of the algorithm. The second equation used to find area when for loop ends seems differ from the Java implementation down below. In the implementation, the formula used is: $(len(a) - stack[top - 1] - 1) * a[stack[top]]$. Can anyone else confirm this for me? Whether it is a typo? Or I have misunderstood sth.

13ShareReply

SHOW 3 REPLIES

dance-henry★22September 12, 2019 12:22 PMReport

There are 3 critical points to understand the stack approach.

- the stack is used to stored the index.
- when we pop the stack, the $heights[stack.pop()]$ is monotonically decreasing, and therefore we can treat the original index $(i - 1)$ as one anchor point to calculate the width.

8ShareReply

SHOW 1 REPLY

nrxn★70September 15, 2018 8:39 AM

What's the intuition behind the stack approach? Like why does this: $(i - stack[top - 1] - 1) * a[stack[top]]$ work? What I mean is what's the relationship between $a[stack[top]]$ and $stack[top - 1]$ as an index? I have seen several explanations of this solution from different sources but nobody has an intuitive explanation of why it works.

7ShareReply

SHOW 7 REPLIES

XiangkunYe★69January 13, 2019 12:20 PM

About YouTube I think there's a mistake that we should use $(len(a) - 1 - stack[top - 1]) * a[stack[top]]$ instead of $(stack[top] - stack[top - 1]) * a[stack[top]]$.

6ShareReply

SHOW 1 REPLY

lh19900702★94May 25, 2017 8:14 AM

The last step of animation is wrong. The final result 16 comes from (the minimum height 2 * the number of column 8). This is because, the last stop of i is 7. $2 * (7 - (-1)) = 16$ as well. The algorithm for ending is incorrect.

4ShareReply

SHOW 1 REPLY

boy2910231★98April 5, 2017 9:53 AMReport

for the last solution, we can add a 0 at the end of heights and "force pop" all the remaining elements in the stack. That way we can save the last while loop

4ShareReply

kevinhynes★286July 2, 2019 6:54 PMReport

Is Solution 3 (vanilla Divide and Conquer) supposed to pass? I am getting TLE in Python but if I copy and paste the Java solution it works..

```
def largestRectangleArea(self, heights: List[int]) -> int:
    max_area = 0
```

Read More

2ShareReply