昵称：lichen782
园龄：1年2个月
粉丝：3
关注：0
+加关注

< 2014年1月 >

日	一	二	三	四	五	六
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1
2	3	4	5	6	7	8

搜索

找找看

谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

我的标签

algorithm(23)
leetcode(22)

随笔分类

LeetCode(22)

随笔档案

2013年7月 (23)

最新评论

1. Re:LeetCode 笔记系列 17 Largest Rectangle in Histogram
有一个问题，3322这种，这样是不是计算出来的是6啊。不是8，倒数第三张图里的棕色框明显不是可以取到的最大面积了，应该加上左边的1才是对吧
--TiegerSong

2. Re:LeetCode 笔记系列 17 Largest Rectangle in Histogram
亲，这个算法错了，你试试2 1 3 4 2 3
--sr2013

3. Re:LeetCode 笔记系列 19 Scramble String [合理使用递归]
超时了吧。。。
--无敌的佩恩

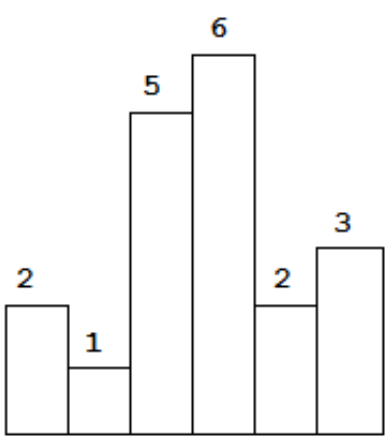
4. Re:LeetCode 笔记系列二 Container With Most Water
第二种方法的剪枝不错啊！
第三种就是短板效应的具体体现吧
--WingTsubasa

5. Re:LeetCode 笔记系列七 Substring with Concatenation of All Words
这题是压线通过，更好的优化应该用类似KMP的算法。请参考：

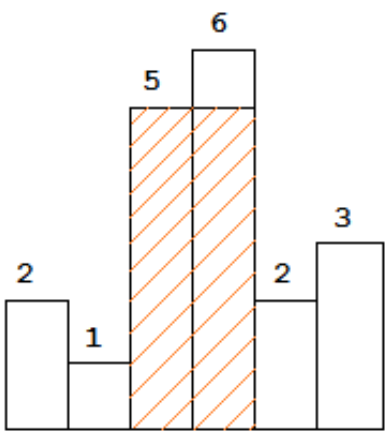
LeetCode 笔记系列 17 Largest Rectangle in Histogram

题目： **Largest Rectangle in Histogram**

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.

For example,
Given height = `[2,1,5,6,2,3]`,
return `10`.

解法一： 这样的题目一般都有**O(n)**的解法，因为**O(n^2)**的解法实在是太显而易见了。

可悲的是，本娃就只写出了后者。。。代码如下：

View Code

基本思想就是遍历所有**[i, j]**，并在过程中找出中间最矮的**bar**，得出从**i**到**j**的矩形面积。

不过我就知道，一定有大神用他们极简的代码来切题，下面就是一个。

解法二：

```
1 public int largestRectangleArea2(int[] height) {
2         Stack<Integer> stack = new Stack<Integer>();
3         int i = 0;
4         int maxArea = 0;
5         int[] h = new int[height.length + 1];
6         h = Arrays.copyOf(height, height.length + 1);
7         while(i < h.length){
8             if(stack.isEmpty() || h[stack.peek()] <= h[i]){
9                 stack.push(i++);
10            }else {
11                int t = stack.pop();
```

http://n00tc0d3r.blogspot.com/2013/06/substring-with-concatenation-of-all.html

--WingTsubasa

阅读排行榜

1. LeetCode 笔记系列 18 Maximal Rectangle [学以致用](1602)
2. LeetCode 笔记系列 17 Largest Rectangle in Histogram(1161)
3. LeetCode 笔记系列七 Substring with Concatenation of All Words(592)
4. LeetCode 笔记系列六 Reverse Nodes in k-Group [学习如何逆转一个单链表](495)
5. LeetCode 笔记系列 20 Interleaving String [动态规划的抽象](384)

评论排行榜

1. LeetCode 笔记系列 17 Largest Rectangle in Histogram(4)
2. LeetCode 笔记系列13 Jump Game II [去掉不必要的计算](1)
3. LeetCode 笔记系列二 Container With Most Water(1)
4. LeetCode 笔记系列七 Substring with Concatenation of All Words(1)
5. LeetCode 笔记系列 19 Scramble String [合理使用递归](1)

推荐排行榜

1. LeetCode 笔记系列二 Container With Most Water(1)
2. LeetCode 笔记系列16.3 Minimum Window Substring [从O(N*M)， O(NlogM)到O(N)，人生就是一场不停的战斗](1)

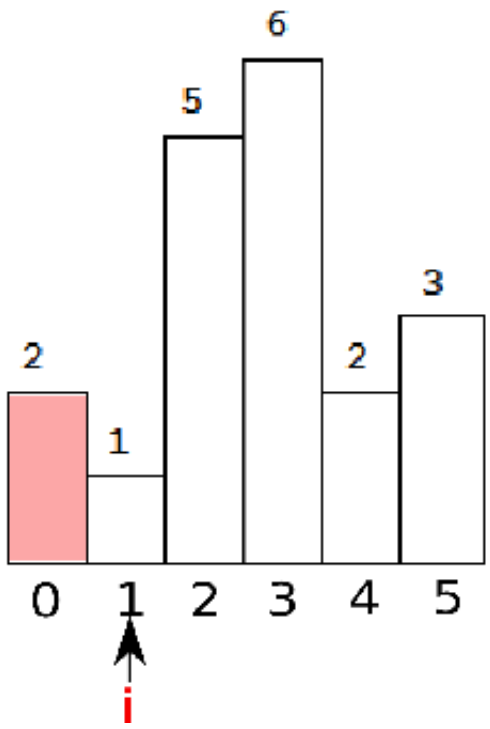
```
12                 maxArea = Math.max(maxArea, h[t] * (stack.isEmpty() ? i : i -
stack.peek() - 1));
13             }
14         }
15         return maxArea;
16     }
```

16行，给跪了。。。。

这个我不去**debug**下都特么不知道在干嘛。

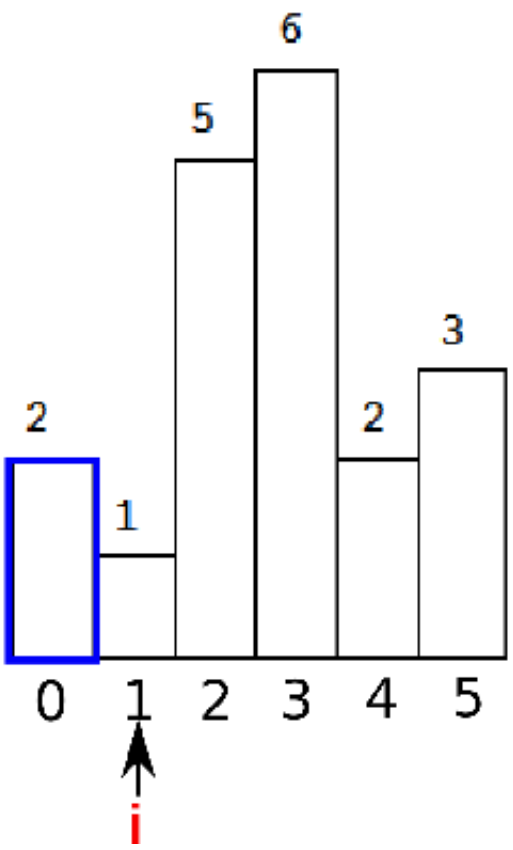
那要不就**debug**下看看这段代码在做神马。例子就用题目中的**[2,1,5,6,2,3]**吧。

首先，如果栈是空的，那么索引**i**入栈。那么第一个**i=0**就进去吧。注意栈内保存的是索引，不是高度。然后**i++**。

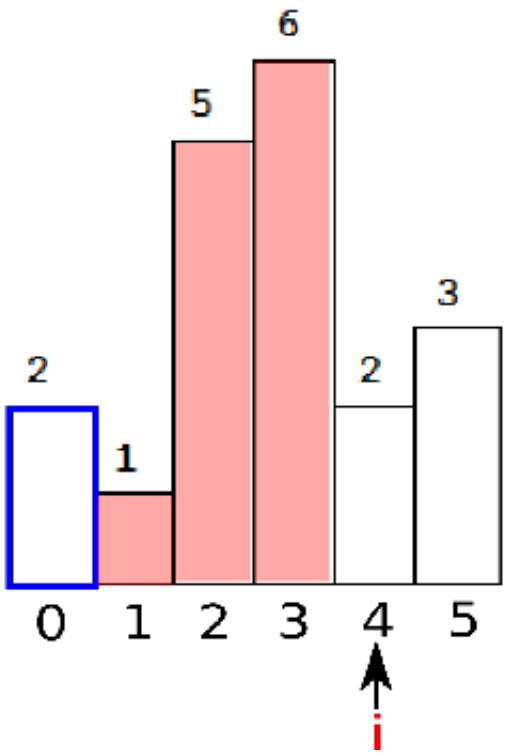


然后继续，当**i=1**的时候，发现**h[i]**小于了栈内的元素，于是出栈。（由此可以想到，哦，看来**stack**里面只存放**单调递增的索引**）

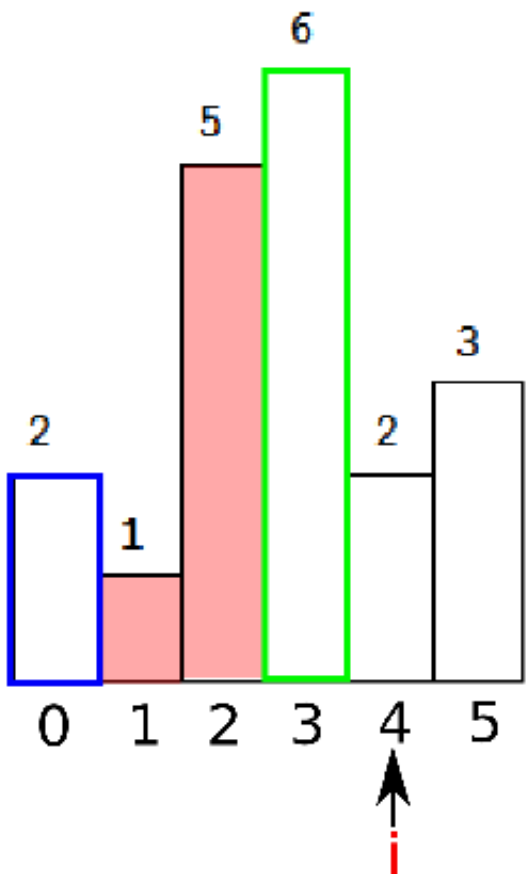
这时候**stack**为空，所以面积的计算是**h[t] * i.t**是刚刚弹出的**stack**顶元素。也就是蓝色部分的面积。



继续。这时候**stack**为空了，继续入栈。注意到只要是连续递增的序列，我们都要**keep pushing**，直到我们遇到了**i=4**，**h[i]=2**小于了栈顶的元素。

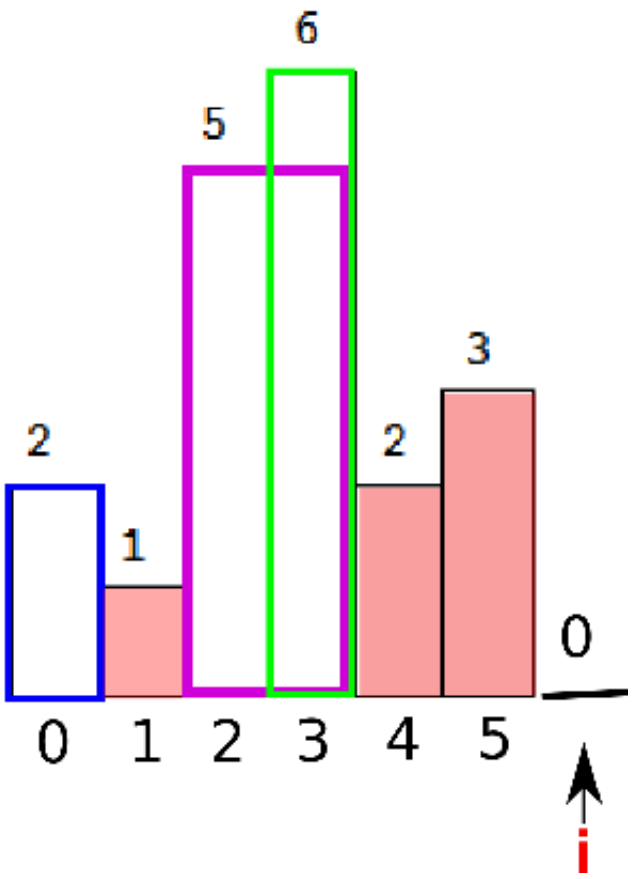


这时候开始计算矩形面积。首先弹出栈顶元素，**t =3**。即下图绿色部分。

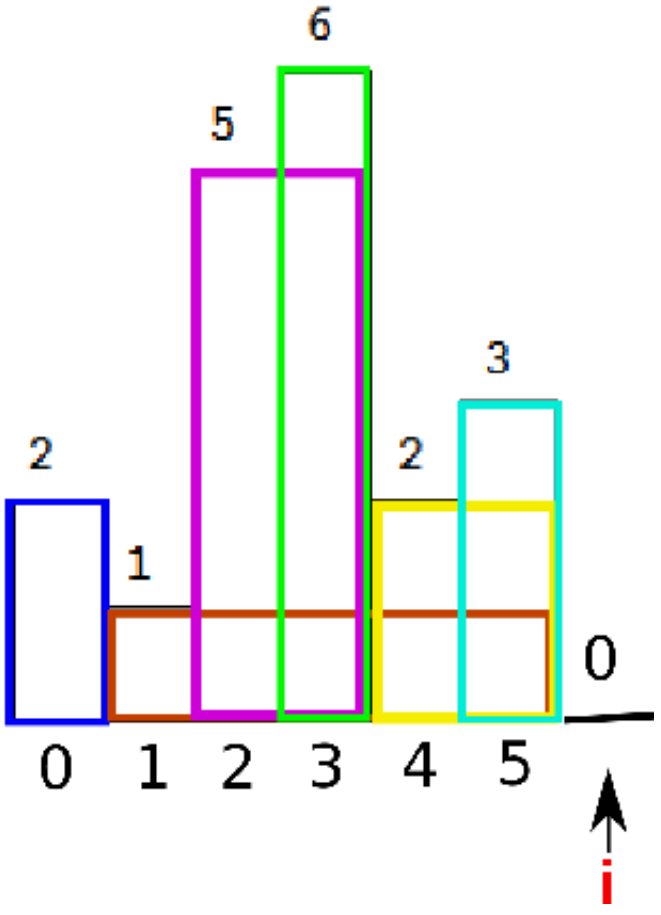


接下来注意到栈顶的（索引指向的）元素还是大于当前**i**指向的元素，于是出栈，并继续计算面积，桃红色部分。

最后，栈顶的（索引指向的）元素大于了当前**i**指向的元素，循环继续，入栈并推动**i**前进。直到我们再次遇到下降的元素，也就是我们最后人为添加的**dummy**元素**0.**



同理，我们计算栈内的面积。由于当前**i**是最小元素，所以所有的栈内元素都要被弹出并参与面积计算。



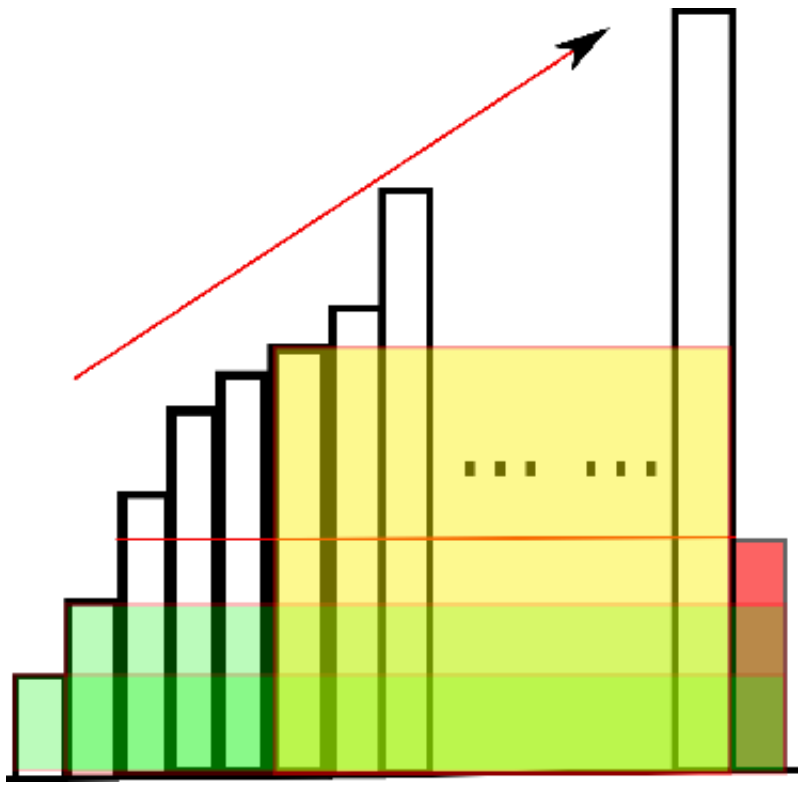
注意我们在计算面积的时候已经更新过了**maxArea**。

总结下，我们可以看到，**stack**中总是保持递增的元素的索引，然后当遇到较小的元素后，依次出栈并计算栈中**bar**能围成的面积，直到栈中元素小于当前元素。

可是为什么这个方法是正确的呢？ 我也没搞清楚。只是觉得不明觉厉了。

----------更新----------

可以这样理解这个算法，看下图。



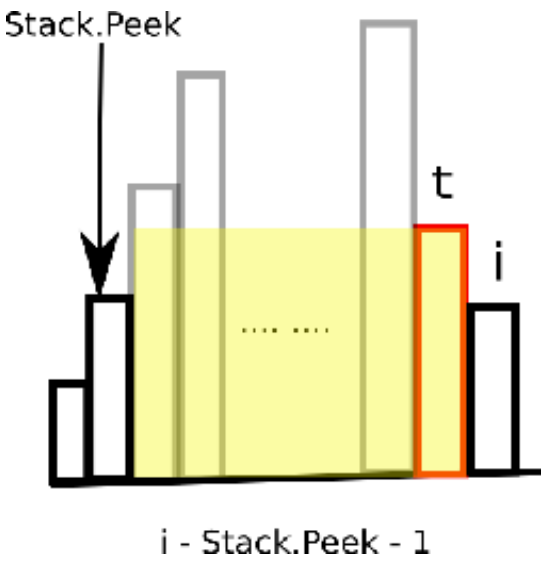
例如我们遇到最后遇到一个递减的**bar**（红色）。高度位于红线上方的（也就是算法中栈里面大于最右**bar**的）元素，他们是不可能和最右边的较小高度**bar**围成一个比大于在弹栈过程中的矩形面积了（黄色面积），因为红色的**bar**对他们来说是一个短板，和红色**bar**能围成的最大面积也就是红色的高度乘以这些"上流社会"所跨越的索引范围。但是"上流社会"的高度个个都比红色**bar**大，他们完全只计算彼此之间围成的面积就远远大于和红色**bar**围成的任意面积了。所以红色**bar**是不可能参与"上流社会"的**bar**的围城的（好悲哀）。

但是屌丝也不用泄气哦。因为虽然长度不占优势，但是团结的力量是无穷的。它还可以参与"比较远的"比它还要屌丝的**bar**的围城。他们的面积是有可能超过上流社会的面积的，因为距离啊！所以弹栈到比红色**bar**小就停止了。

另外一个细节需要注意的是，弹栈过程中面积的计算。

h[t] * (stack.isEmpty() ? i : i - stack.peek() - 1)

h[t]是刚刚弹出的栈顶端元素。此时的面积计算是**h[t]**和前面的"上流社会"能围成的最大面积。这时候要注意哦，栈内索引指向的元素都是比**h[t]**小的，如果**h[t]**是目前最小的，那么栈内就是空哦。而在目前栈顶元素和**h[t]**之间（不包括**h[t]**和栈顶元素），都是大于他们两者的。如下图所示：



那**h[t]**无疑就是**Stack.Peek**和**t**之间那些上流社会的短板啦，而它们的跨越就是**i - Stack.Peek - 1**。

所以说，这个弹栈的过程也是维持程序不变量的方法啊：**栈内元素一定是要比当前i指向的元素小的。**

---华丽---

我只想问算法的作者，他们到底是怎么想出来的，在这么短的时间内。是不是有一些类似的研究或者算法给他们以灵感？

太有画面感了有木有！

分类: **LeetCode**
标签: **leetcode**, **algorithm**

« 上一篇：**LeetCode** 笔记系列**16.3 Minimum Window Substring [**从**O(N*M)**， **O(NlogM)**到**O(N)**，人生就是一场不