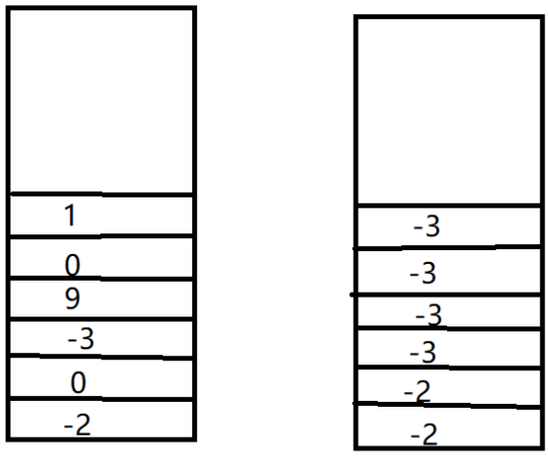
```
        else {
            if (st.empty() || st.pop() != c)
                return false;
        }
    }

    return st.empty();
}
```

155.最小栈

解题思路：为了在O(1)的时间内拿到栈中的最小元素，同时保持元素先进后出的特性，我们维护另外一个最小栈，对每个元素维护一个其对应的最小值。该题需要两个栈，一个数据栈存储所有元素，一个最小栈，存储每个元素对应的最小元素

[-2, 0, -3, 9, 0, 1]



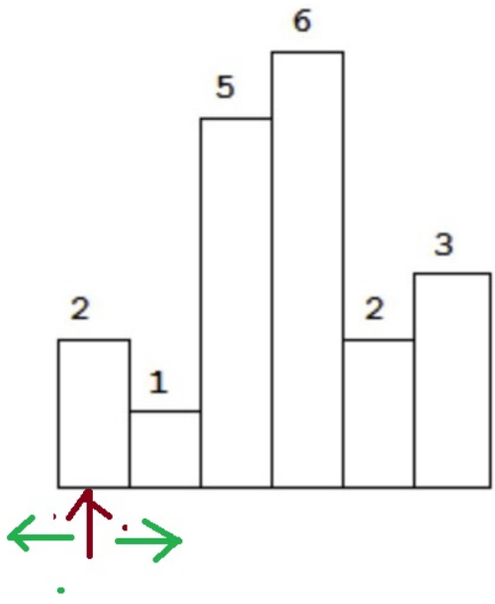
如图，在最小栈中维护每个元素对应的最小值。更新最小栈时，比较当前元素和最小栈顶元素。注意为了保证数据的先进先出特性，弹出操作时需要对数据栈和最小栈同时进行弹出，保证元素的对应关系

```
class MinStack {

    /** initialize your data structure here. */
    //需要维护当前的最小元素且满足先进后出的栈特性
    //使用两个栈，一个存放所有元素，一个存放当前栈中的最小元素
    private Stack<Integer> st = new Stack<Integer> ();
    private Stack<Integer> minst = new Stack<Integer> ();

    public MinStack() {

    }

    public void push(int x) {
        //st将元素入栈
        st.push(x);
```

```
        //同时维护最小栈信息
        if (minst.empty())
            minst.push(x);
        else{
            if (minst.peek() < x)
                minst.push(minst.peek());
            else
                minst.push(x);
        }
    }

    public void pop() {
        //同时弹出最小栈和数据栈的元素，保持两个栈同步
        st.pop();
        minst.pop();
    }

    public int top() {
        return st.peek();
    }

    public int getMin() {
        return minst.peek();
    }
}
```

84.柱状图中最大的矩形

题目：给定 *n* 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。求在该柱状图中，能够勾勒出来的矩形的最大面积



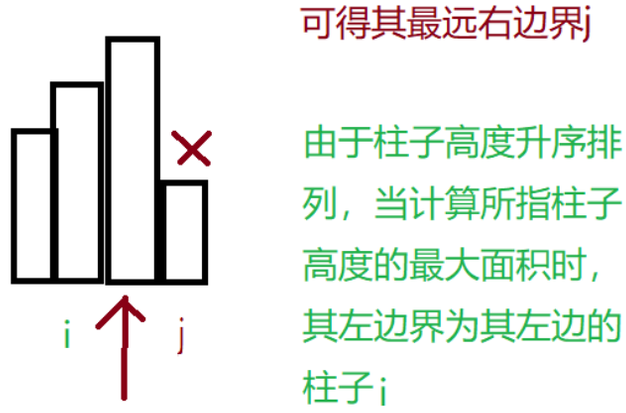
解题思路

case1：对于第一根柱子，思考以它为高的最大柱子面积是多少？其本质是**找出以其下标 i 为中心点，向左和向右可以扩展的最大宽度。**由此可以想到第一种解法，枚举每根柱子的高度，依次找其左边最远点和右边最远点，最后求其面积。时间复杂度为O(n^2)，空间复杂度O(1)

```
public int largestRectangleArea(int[] heights) {
    //枚举柱子高度，找其最远可以扩展的高度
    int maxAns = 0;
    for(int i = 0; i < heights.length; i++){

        //找其左边界
        int left = i - 1;
        while(left >= 0 && heights[left] >= heights[i]) left--;

        //找其右边界
        int right = i + 1;
        while(right <= heights.length -1 && heights[right] >= heights[i])
right++;

        //计算面积
        int area = (right - left -1) * heights[i];
        maxAns = Math.max(maxAns, area);
    }
    return maxAns;
}
```

case2：思考 case1 中的枚举过程，我们发现在枚举每一根柱子的高度时，找其左右边界过程中，存在某些左边的柱子和右边的柱子是被重复遍历的。我们是否可以使用辅助数据结构来记录这些已经遍历过的柱子，然后再遍历新柱子的高度时直接操作之前已经得到的信息，从而减少循环次数，降低时间复杂度?

我们发现柱子最远可以扩展的左右边界是遇到比自己矮的柱子时停止，那么在遍历柱子时，保持柱子高度的升序，当遇到矮柱子时，就需要立马计算以上一根柱子高度为高可得到的最大面积。对于之前的柱子，我们还不知道其最远可扩展右边界，因此不需要计算最大面积。**根据面积计算的先进后计算的性质，使用辅助数据结构栈，并维持栈中元素的非递减性。**此解法只需要遍历一次柱子高度，每个柱子的下标都只会进出栈一次，时间复杂度为O(n)




```
public int largestRectangleArea(int[] heights) {
    int maxAns = 0;
```

```
    //保证栈中存放遍历过的柱子的高度非递减性质
    //栈中存放柱子下边，用于计算最大可扩展宽度
    Stack<Integer> idx = new Stack<Integer>();

    int i = 0;
    while(i < heights.length){
        //若当前柱子i高度小于栈顶柱子高度，计算栈顶柱子高度，注意该步骤放在循环中，因为可能有多根柱子需要计算面积
        while (!idx.empty() && heights[idx.peek()] > heights[i]){
            //栈顶柱子下标
            int top = idx.pop();
            //右边界
            int width = i;

            //当栈顶柱子存在左边界时，计算最大宽度，需要考虑第一根柱子的特殊情况
            if (!idx.empty())
                width = width - idx.peek() -1;
            int area = width * heights[top];
            maxAns = Math.max(maxAns, area);
        }

        //i柱子进栈，继续遍历下一根柱子
        idx.push(i);
        i++;
    }

    //计算栈中剩余柱子面积
    while (!idx.empty()){
        int top = idx.pop();

        //注意剩余柱子的右边界
        int width = heights.length;
        if (!idx.empty())
            width = width - idx.peek() -1;
        int area = width * heights[top];
        maxAns = Math.max(maxAns, area);
    }
    return maxAns;
}
```