【第十八课】单调栈及经典问题（下）

84. 柱状图中最大的矩形

1.求切出最大地矩形面积。
2.矩形是由高和宽决定了面积 如何计算最大地矩形面积。
3.向前找比当前元素小地位置，，，往后找比当前元素小地位置， 高度位5地木板前面是木板1，往后是木板2。
4.高度为5 宽度为2 面积 = 5*2 = 10。
5.需要在第一个位置 加一个虚拟木板 下标是-1，在最后一个位置 加一个虚拟木板 下标是数组的长度。
整体思路还是单调栈的做法。

```
const largestRectangleArea = (heights) => {
    let stack = [];
    let l =  new Array(heights.length),r =  new Array(heights.length);
    let n = heights.length;
    for(let i = 0;i < n; i++) l[i] = -1,r[i] = n;
    for(let i = 0;i < n; i++) {
        while(stack.length && heights[i] <= heights[stack[stack.length - 1]] ) {
            r[stack[stack.length - 1]] = i;
            stack.pop();
        }
        if(stack.length) l[i] = stack[stack.length - 1];
        stack.push(i)
    }
    let ans = 0;
    for(let i = 0; i < n ;i++){
        ans = Math.max(ans, heights[i] * (r[i]-l[i]-1));
    }
    return ans;

}
```

1856. 子数组最小乘积的最大值

1.最小值 = 数组的最小值 * 数组所有元素的和值。
2.假设当前值 就是 子数组的 最小值，其实就是求 以当前值为最小值的子数组 最长可以切多长。
3.向左找到一个小于当前值的位置，向右找到一个小于当前值的位置。中间这段数组的和值，就是算最小乘积的第二个值，再乘上当前值就是结果。
4.以每个数字作为子数组的最小值，求一个最大乘积， 在所有乘积里面找一个最大值，这就是这道题要求的。

5.编程技巧：如何快速求一段区间和和值= 前缀和。
6.第一步：先求每一个元素的最大跨度，确定下标。
7.第二步，需要一个前缀和数组 算一下区间和，注意 前缀和数组的下标是从1开始，原数组下标是从0开始。
8.其实就是上一道变种，84 上一道求得是跨度，这道乘的是跨度的和值。

```
var maxSumMinProduct = function(nums) {
  // 前缀和
  const sum = [0];
  const mod = 1e9 + 7;
  for(let i = 1; i <= nums.length; i++) {
    sum[i] = sum[i - 1] + nums[i - 1];
  }
  // 使用单调栈求解出左侧第个严格小于该数的元素位置，和右侧第一个严格小于该数的元素的位置
  let stack = [];
  const len = nums.length;
  const right = new Array(len).fill(len);
  for(let i = 0; i < len; i++) {
    while(stack.length && nums[stack[stack.length - 1]] > nums[i]) {
      right[stack[stack.length - 1]] = i;
      stack.pop();
    }
    stack.push(i)
  }
  stack = [];
  const left = new Array(len).fill(-1);
  for(let j = len - 1; j >= 0; j--) {
    while(stack.length && nums[stack[stack.length - 1]] > nums[j]) {
      left[stack[stack.length - 1]] = j;
      stack.pop();
    }
    stack.push(j);
  }
  // 根据前缀和和left，  right数组进行枚举求解
  let max = BigInt(0);
  for(let k = 0; k < len; k++) {
    const total = BigInt(sum[right[k]] - sum[left[k] + 1]) * BigInt(nums[k]);
    if (max < total) {
      max = total;
    }
  }
  return max % BigInt(mod);
};
```

907. 子数组的最小值之和

1.题目的关键在于找到A[i]属于的区间, 满足A[i]为该区间内的最小值。

2.这里使用单调递增栈巧妙地找到该区间的左右边界：

- 单调栈的前一个元素为小于该元素的第一个索引, 即为区间的左边界
- 单调栈的元素出栈时, 说明其后面出现了比该元素更小的值, 即为区间的右边界

3.就是将第i个元素入队列，求出前面区间的跨度，就是第I个元素贡献的和值，再加上sum3（假设前面有3个值，前三个和值）。

```
 /**
 * @param {number[]} arr
 * @return {number}
 */
var sumSubarrayMins = function (arr) {
    let stack = [];
    let mod_num =  1e9 + 7;
    let ans = 0;
    let sum = new Array(arr.length + 1);
    sum[0] = 0;
    for(let i  = 0;i < arr.length;i++){
        while(stack.length && arr[i] <= arr[stack[stack.length - 1]] )
stack.pop();
        let ind = stack.length ? stack[stack.length - 1] : -1;
        stack.push(i);
        sum[stack.length] = (sum[stack.length - 1] + arr[i] *(i - ind)) %
mod_num;
        ans += sum[stack.length];
        ans %= mod_num;
    }
    return ans;

}
```

456. 132 模式

【助教代码思路】

1.所谓「单调栈」就是栈中的元素都是依次递增或者递减的 如 [4, 3, 2, 1]
2.【132】模式至少存在三个数字 分别为 min， max， med
3.min， max， med 顺序不可调换
4.遍历数组找 med 的位置
5.如果当前遍历大于栈顶元素 则栈的单调性被破坏，清空栈 将当前值压入栈底 将栈底元素作为med
6.如果 med max 都存在 遍历到 比med小的值 可放入min位置 返回true7.本题用单调栈存储 max 位置的值
8.med 初始值设置为理论最小值 即可不用判断 med max 存在

【船长思路】

1.【132 】模式 是指第一个值最小，第二个值最大，第三个值落在 第一个值和第二个值中间的，并且排在最后一位：比如 3 12 9。
2.首先，把每一个元素，当成是中间的最大值。 把当前元素入一个单调递减栈。
3.然后把小于我当前元素的值都弹出去，我弹出去的最后一个元素就是 中间这段区间的最大值。
4.接着，中间这段区间的最大值 那就是当前区间内部的 小于我当前元素的 最大值，我们判断 最后弹出去的这个元素，是否落在 最大值和最小值中间即可

5.那么这个算法过程中，有个地方需要证明，落在当前区间内部的最大值，其实并不是我当前元素后面所有元素中小于他的那个最大值

```
var find132pattern = function(nums) {
    let stack = [], med = -Infinity
    for (let i = nums.length - 1; i >= 0; i--) {
        if (nums[i] < med) return true
        while (stack.length > 0 && stack[stack.length - 1] < nums[i]) {
            med = stack.pop()
        }
        stack.push(nums[i])
    }
    return false
};
```

42. 接雨水

接到的雨水 就是每个柱子找到左边第一个比他大的柱子和右边第一个比他大的柱子，通过两边的高柱子之间的距离和相对低的柱子和当前柱子的差来计算能接到的雨水，然后将所有接到的水相加

```
var trap = function(height) {
    let ans = 0;
    const stack = [];
    const n = height.length;
    for (let i = 0; i < n; ++i) {
        while (stack.length && height[i] > height[stack[stack.length - 1]]) {
            const top = stack.pop();
            if (!stack.length) {
                break;
            }
            const left = stack[stack.length - 1];
            const currWidth = i - left - 1;
            const currHeight = Math.min(height[left], height[i]) - height[top];
            ans += currWidth * currHeight;
        }
        stack.push(i);
    }
    return ans;
};
```