【第十八课】单调栈及经典问题（上）

155. 最小栈

1.我们只需要设计一个数据结构，使得每个元素 a 与其相应的最小值 m 时刻保持一一对应。因此我们可以使用一个辅助栈，与元素栈同步插入与删除，用于存储与每个元素对应的最小值。

2.当一个元素要入栈时，我们取当前辅助栈的栈顶存储的最小值，与当前元素比较得出最小值，将这个最小值插入辅助栈中；

3.当一个元素要出栈时，我们把辅助栈的栈顶元素也一并弹出；

```
var MinStack = function() {
    this.x_stack = [];
    this.min_stack = [Infinity];
};

MinStack.prototype.push = function(x) {
    this.x_stack.push(x);
    this.min_stack.push(Math.min(this.min_stack[this.min_stack.length - 1], x));
};

MinStack.prototype.pop = function() {
    this.x_stack.pop();
    this.min_stack.pop();
};

MinStack.prototype.top = function() {
    return this.x_stack[this.x_stack.length - 1];
};

MinStack.prototype.getMin = function() {
    return this.min_stack[this.min_stack.length - 1];
};
```

496. 下一个更大元素 I

1、创建一个临时栈，一个哈希表，然后遍历 nums2
2、若当前栈无数据，则当前数字入栈备用。
3、若当前栈有数据，则用当前数字与栈顶比较：
3、 当前数字 > 栈顶，代表栈顶对应下一个更大的数字就是当前数字，则将该组数字对应关系，记录到哈希表。
4、 当前数字 < 栈顶，当前数字压入栈，供后续数字判断使用。
5、这样，我们就可以看到哈希表中存在部分 nums2 数字的对应关系了，而栈中留下的数字，代表无下一个更大的数字，我们全部赋值为 -1，然后存入哈希表即可。
6、遍历 nums1，直接询问哈希表拿对应关系即可。

```
let nextGreaterElement = function(nums1, nums2) {
    let map = new Map(), stack = [], ans = [];
    nums2.forEach(item => {
        while(stack.length && item > stack[stack.length-1]){
            map.set(stack.pop(), item)
        };
        stack.push(item);
    });
    stack.forEach(item => map.set(item, -1));
    nums1.forEach(item => ans.push(map.get(item)));
    return ans;
};
```

503. 下一个更大元素 II

单调栈

1、我们可以使用单调栈解决本题。单调栈中保存的是下标，从栈底到栈顶的下标在数组 nums 中对应的值是单调不升的。

2、每次我们移动到数组中的一个新的位置 ii，我们就将当前单调栈中所有对应值小于 nums[i] 的下标弹出单调栈，这些值的下一个更大元素即为 nums[i]（证明很简单：如果有更靠前的更大元素，那么这些位置将被提前弹出栈）。随后我们将位置 ii 入栈。

3、但是注意到只遍历一次序列是不够的，例如序列 [2,3,1][2,3,1]，最后单调栈中将剩余 [3,1][3,1]，其中元素 [1][1] 的下一个更大元素还是不知道的。

4、 复制该序列的前 n-1个元素拼接在原序列的后面。这样我们就可以将这个新序列当作普通序列。

5、遍历两次数组在处理时对下标取模即可，遍历的时候处理两次，就是把原数组扩大了一倍

```
var nextGreaterElements = function(nums) {
    const n = nums.length;
    const ret = new Array(n).fill(-1);
    const stk = [];
    for (let i = 0; i < n * 2 - 1; i++) {
        while (stk.length && nums[stk[stk.length - 1]] < nums[i % n]) {
            ret[stk[stk.length - 1]] = nums[i % n];
            stk.pop();
        }
        stk.push(i % n);
    }
    return ret;
};
```

901. 股票价格跨度

1.用一个count属性来记录next操作的次数。

2.单调栈stack维护一个递减栈，递减栈中存储的 是一个包括它的值（value）和它是多少次next操作的值(count)。

3.每一次next操作，如果元素比栈顶元素的value值要大，说明他的递增序列还需要继续判断，对栈内元素出栈，直到栈顶元素的值大于当前next操作的当前值。

4.当栈顶元素大于next操作的当前值时，说明递增序列是从栈顶元素的下一next操作，到当前next操作，所以他们count的差就是递增的天数

```
 var StockSpanner = function() {
    this.stack = [];
    this.count = 0;
};

/**
 * @param {number} price
 * @return {number}
 */
StockSpanner.prototype.next = function(price) {
    while (this.stack.length && price >=  this.stack[this.stack.length -
1].value) {
        this.stack.pop();
    }
    let tmp = this.stack.length ? this.stack[this.stack.length - 1].index : 0;
    this.count++;
    this.stack.push({
        index: this.count,
        value: price
    });
    return this.count - tmp;
};
```

739. 每日温度

递减单调栈思路，剔除波谷，留下波峰，栈内逐一向前对比，根据下标完成对应结果获取。

```
 var dailyTemperatures = function(T) {

    let stack = []
    let res = Array(T.length).fill(0)
    for (let i = 0;i < T.length;i++) {
        while (stack.length && T[i] > T[stack[stack.length - 1]]) {
            let len = stack.length
            if (T[i] > T[stack[len - 1]]) {
                res[stack[len - 1]] = i - stack[len - 1]
                stack.pop()
            }
        }
        stack.push(i)
    }
    return res
};
```