单调栈

每日温度

例子

示例 1:

输入：`temperatures` = [73,74,75,71,69,72,76,73]
输出：[1,1,4,2,1,1,0,0]

示例 2:

输入：temperatures = [30,40,50,60]
输出：[1,1,1,0]

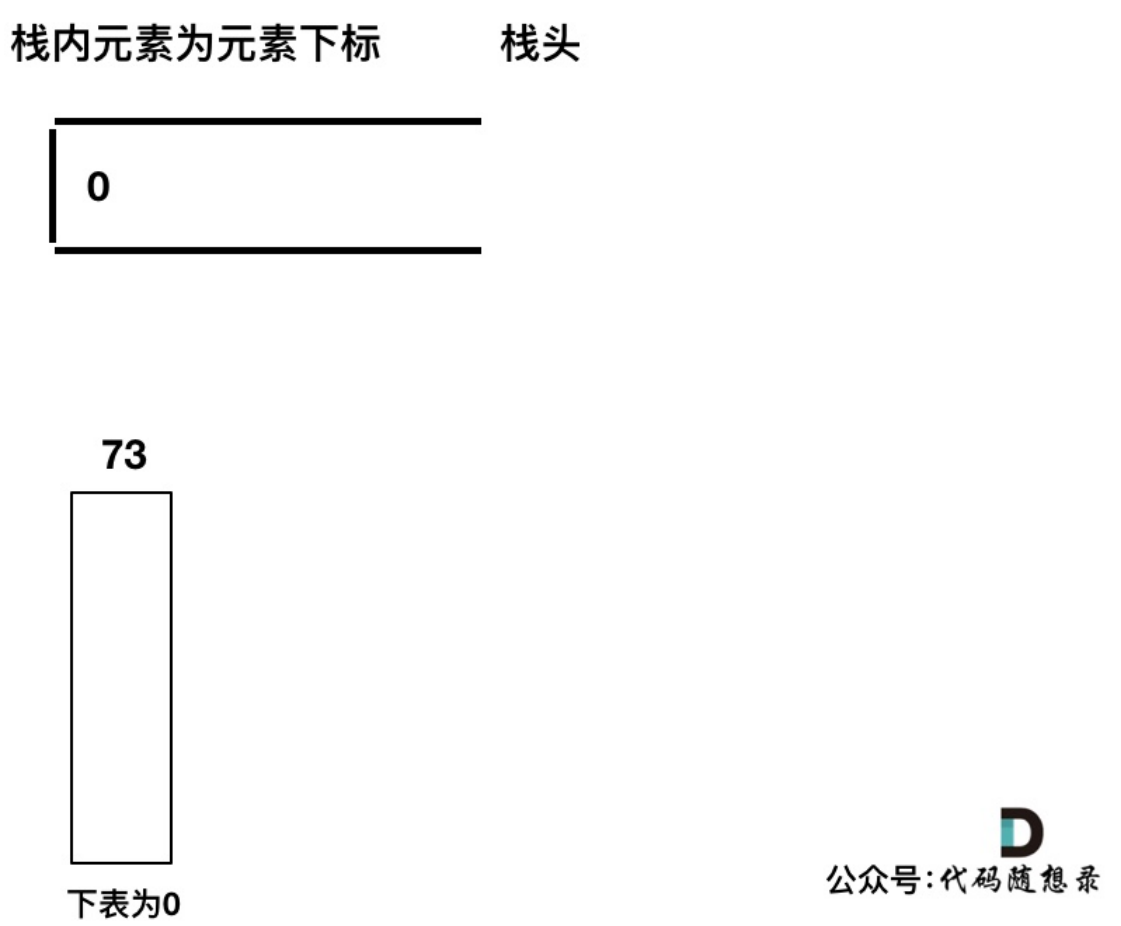
示例 3:

输入：temperatures = [30,60,90]
输出：[1,1,0]

解析

栈s保存下标位置。当天温度小于等于栈顶位置的温度时，则插入栈顶；如果当天温度大于栈顶位置温度，则计算栈顶位置的比栈顶位置温度大的元素的个数（i-s.top()）

样例：temperatures = [73, 74, 75, 71, 69, 72, 76, 73]




```
class Solution {
```

```
public:
    vector<int> dailyTemperatures(vector<int>& temperatures) {
        stack<int> s;//保存下标
        vector<int> result(temperatures.size(), 0);
        s.push(0);
        for (int i = 1;i < temperatures.size();i++) {
            if (temperatures[i] <= temperatures[s.top()]) {//栈顶维持当前温度最低的下
标，从栈顶到栈底温度依次递增
                s.push(i);
            }
            else {
                while (!s.empty() && temperatures[i] > temperatures[s.top()]) {
                    result[s.top()] = i - s.top();//收获结果
                    s.pop();
                }
                s.push(i);
            }
        }

        return result;
    }
};
```

下一个更大的元素

下一个更大的元素I

```
输入：nums1 = [4,1,2], nums2 = [1,3,4,2].
输出：[-1,3,-1]
解释：nums1 中每个值的下一个更大元素如下所述：
4 ，nums2 = [1,3,4,2]。不存在下一个更大元素，所以答案是 -1 。
1 ，nums2 = [1,3,4,2]。下一个更大元素是 3 。
2 ，nums2 = [1,3,4,2]。不存在下一个更大元素，所以答案是 -1 。
```

单调栈

```
class Solution {
public:
    vector<int> nextGreaterElement(vector<int>& nums1, vector<int>& nums2) {
        stack<int> st;//记录nums2的下标位置
        vector<int> result(nums1.size(), -1);//如果没有比其大的元素，则为-1。所以初始化
```

```
为-1
        unordered_map<int, int> unmap;
        //下标元素与下标

        for (int i = 0;i < nums1.size();i++) {
            unmap[nums1[i]] = i;
        }

        st.push(0);
        for (int i = 1;i < nums2.size();i++) {
            if (nums2[i] <= nums2[st.top()]) {
                st.push(i);
            }
            else {
                while (!st.empty() && nums2[i] > nums2[st.top()]) {
                    if (unmap.count(nums2[st.top()])) {//如果在nums1中出现过
                        int index = unmap[nums2[st.top()]];//这个元素的下标
                        result[index] = nums2[i];
                    }
                    st.pop();
                }
                st.push(i);
            }
        }

        return result;
    }
};
```

暴力解法

```
class Solution {
public:
    vector<int> nextGreaterElement(vector<int>& nums1, vector<int>& nums2) {
        vector<int> result(nums1.size(), -1);

        for (int i = 0; i < nums1.size(); i++) {
            bool found = false;  // 用于标记是否找到了下一个更大的元素
            for (int j = 0; j < nums2.size(); j++) {
                if (nums1[i] == nums2[j]) {
                    found = true;
                }
                if (found && nums2[j] > nums1[i]) {
                    result[i] = nums2[j];
                    break;  // 找到下一个更大的元素后，结束循环
```

```
                }
            }
        }

        return result;
    }
};
```

下一个更大的元素II

例子

```
输入: nums = [1,2,1]
输出: [2,-1,2]
解释: 第一个 1 的下一个更大的数是 2；
数字 2 找不到下一个更大的数；
第二个 1 的下一个最大的数需要循环搜索，结果也是 2。
```

单调栈

本质上与每日温度的习题是一样的，特殊之处是：

1.本体要求是判断一个环的大小情况，所以采用取余的方式来减少空间复杂度

2.本题求的是下一个比当前元素大的元素的数值，而不是求两者距离

```
class Solution {
public:
    vector<int> nextGreaterElements(vector<int>& nums) {

        if (nums.size() == 1) {
            vector<int> vec;
            vec.push_back(-1);
            return vec;
        }
        stack<int> st;//栈中保存下标
        vector<int> result(nums.size(), -1);

        st.push(0);
        for (int i = 1;i < nums.size() * 2;i++) {
            //i %= nums.size();//这个地方直接取余会导致i的遍历出现错误
            int temp_i = i % nums.size();
```

```
            if (nums[temp_i] <= nums[st.top()]) {//栈顶保存到目前为止最小的元素的下标
                st.push(temp_i);
            }
            else {
                while (!st.empty() && nums[temp_i] > nums[st.top()]) {
                    result[st.top()] = nums[temp_i];//注意这里没有求距离，而是求的下一个
元素
                    st.pop();
                }
                st.push(temp_i);
            }
        }

        return result;
    }
};
```

暴力解法

把长度拓展成两倍后再进行遍历

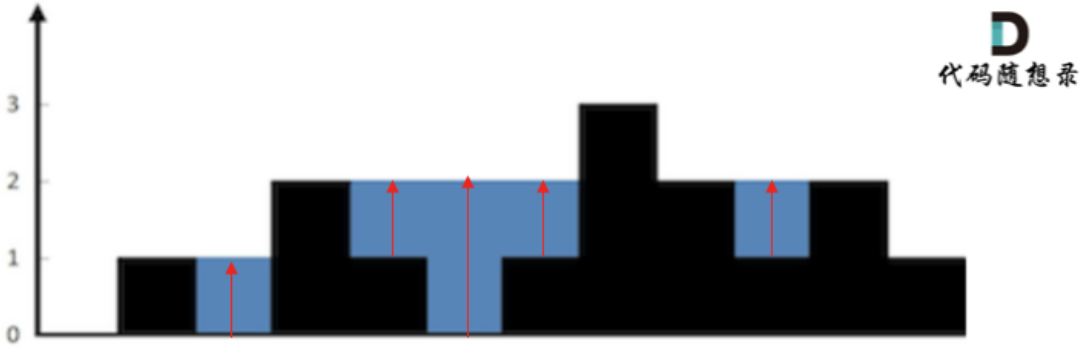
```
class Solution {
public:
    vector<int> nextGreaterElements(vector<int>& nums) {
        int len = nums.size();
        vector<int> result(len, -1);
        nums.insert(nums.end(), nums.begin(), nums.end());//扩展成两倍

        for (int i = 0;i < len;i++) {
            for (int j = i;j <len*2;j++) {
                if (nums[j] > nums[i]) {
                    result[i] = nums[j];
                    break;
                }
            }
        }

        return result;
    }
};
```

接雨水

暴力解法

暴击解法的思路是，求当前柱左边最高的柱子以及右端最高的柱子中的最小值（木桶效应）

然后只计算当前柱能容纳的水的体积

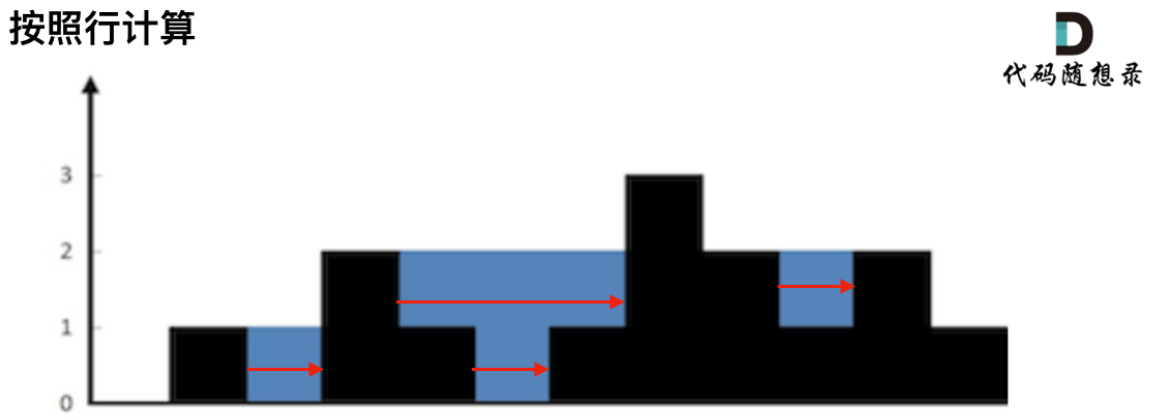



```
class Solution {//法一：暴力算法
public:
    int trap(vector<int>& height) {
        int sum = 0;
        for (int i = 0;i < height.size();i++) {
            if (i == 0 || i == height.size() - 1) {//首尾柱子不能接水
                continue;
            }

            int RightMaxHeight = height[i];
            int LeftMaxHeight = height[i];
            for (int right = i + 1;right < height.size();right++) {
                RightMaxHeight = max(RightMaxHeight, height[right]);
            }
            for (int left = i - 1;left >= 0;left--) {
                LeftMaxHeight = max(LeftMaxHeight, height[left]);
            }
            sum += min(LeftMaxHeight, RightMaxHeight) - height[i];
        }

        return sum;
    }
};
```

单调栈

单调栈本质上是横着求水的体积。

栈顶保存的是：遍历到当前位置的最小的柱子高度的下标。从栈顶到栈底，柱子高度递增



上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。 **感谢 Marcos** 贡献此图。

```
class Solution {//法二：单调栈
public:
    int trap(vector<int>& height) {
        stack<int> st;
        int sum = 0;
        st.push(0);

        for (int i = 1;i < height.size();i++) {
            if (height[i] > height[st.top()]) {//此时栈顶一定是凹陷的地方
                while (!st.empty() && height[i] > height[st.top()]) {
                    int mid = height[st.top()];
                    st.pop();
                    if (!st.empty()) {//计算中间值后删除，此时删除后必须再一次判断是否为空
                        int h = min(height[i], height[st.top()]) - mid;
                        int w = i - st.top() - 1;
                        sum += h * w;
                    }

                }
                st.push(i);
            }
            else if (height[i] == height[st.top()]) {
                st.pop();
                st.push(i);
            }
            else {
                st.push(i);
```

```
            }
        }

        return sum;
    }
};
```

最大的矩形面积

暴力解法

暴力解法的思路是，记录左右第一个比当前柱子矮的柱子的下标，用来统计宽度

高度即当前柱子的高度。高度乘以宽度得到当前柱子的面积

```
class Solution {
public:
    int largestRectangleArea(vector<int>& heights) {//暴力解法，时间复杂度为O(n^2)
        int sum = 0;
        for (int i = 0;i < heights.size();i++) {

            int height = heights[i];
            int right = i;
            int left = i;//用来记录宽度

            for (;left >= 0;left--) {
                if (heights[left] < height) {
                    break;
                }
            }

            for (;right < heights.size();right++) {
                if (heights[right] < height) {
                    break;
                }
            }

            int width = right - left - 1;
            sum = max(sum, height * width);
        }

        return sum;
```

```
    }
};
```

单调栈

单调栈的思路是，也是记录其左右第一个比柱子矮的下标，从而得到宽度。

栈顶的元素是，到目前为止遍历的最大元素的下标，从栈顶到栈底元素依次递减（这样就可以求当前元素的第一个比其本身小的元素）

```
class Solution {//单调栈解法
public:
    int largestRectangleArea(vector<int>& heights) {
        stack<int> st;
        st.push(0);

        //首位插入0方便判断
        heights.push_back(0);
        heights.insert(heights.begin(), 0);
        int Max = 0;

        for (int i = 1;i < heights.size();i++) {
            if (heights[i] > heights[st.top()]) {
                st.push(i);
            }
            else if (heights[i] == heights[st.top()]) {
                st.pop();
                st.push(i);
            }
            else {
                while (!st.empty() && heights[i] < heights[st.top()]) {//此时一定会出现最高峰，小大小
                    int height = heights[st.top()];
                    int right = i;
                    st.pop();
                    if (!st.empty()) {
                        int left = st.top();
                        Max = max(Max, (right - left - 1) * height);//只要记录最大的面积即可
                    }
                }
                st.push(i);
            }
        }
```

```
        return Max;
    }
};
```