LeetCode 167. 两数之和 II - 输入有序数组

167. 两数之和 II - 输入有序数组

难度 简单 254

给定一个已按照*升序排列*的有序数组，找到两个数使得它们相加之和等于目标数。

函数应该返回这两个下标值 index1 和 index2，其中 index1 必须小于 index2。

说明:

- 返回的下标值（index1 和 index2）不是从零开始的。
- 你可以假设每个输入只对应唯一的答案，而且你不可以重复使用相同的元素。

示例:

```
输入: numbers = [2, 7, 11, 15], target = 9
输出: [1,2]
解释: 2 与 7 之和等于目标数 9 。因此 index1 = 1, index2 = 2 。
```

算法

(双指针扫描) $O(n)$

用两个指针 i, j 分别从数组首尾往中间扫描，每次将 i 后移一位，然后不断前移 j，直到 $numbers[i] + numbers[j] \leq target$ 为止。如果 $numbers[i] + numbers[j] == target$，则找到了一组方案。

时间复杂度分析：两个指针总共将数组扫描一次，所以时间复杂度是 $O(n)$。

```cpp
/*
双指针算法：先想暴力，如果有单调性，就可以优化
暴力：两重循环，暴力枚举
双指针：i后移1位，j从后往前移动
*/
class Solution {
public:
    vector<int> twoSum(vector<int>& numbers, int target) {
        for (int i = 0, j = numbers.size() - 1; i < j; i ++ )
        {
            while (numbers[j] + numbers[i] > target) j -- ;
            if (numbers[j] + numbers[i] == target)
            {
                vector<int> res;
                res.push_back(i + 1), res.push_back(j + 1);
                return res;
            }
        }
        return {-1, -1};
    }
```

```
};
**************************************************************************
class Solution {
public:
    vector<int> twoSum(vector<int>& numbers, int target) {
        for(int i = 0,j = numbers.size() - 1;i < numbers.size();i ++)
        {
            while(numbers[i] + numbers[j] > target) j --;
            if(numbers[i] + numbers[j] == target) return {i + 1,j + 1};
        }
        return {-1,-1};
    }
};
```

LeetCode 88. 合并两个有序数组

88. 合并两个有序数组

难度 简单 444

给你两个有序整数数组 *nums1* 和 *nums2*，请你将 *nums2* 合并到 *nums1* 中，使 *num1* 成为一个有序数组。

说明:

- 初始化 *nums1* 和 *nums2* 的元素数量分别为 *m* 和 *n* 。
- 你可以假设 *nums1* 有足够的空间（空间大小大于或等于 *m* + *n*）来保存 *nums2* 中的元素。

示例:

```
输入：
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6],       n = 3

输出：[1,2,2,3,5,6]
```

算法1

(线性合并) 时间复杂度 $O(m+n)$, 空间复杂度$O(1)$

1. 设置 `cur` 指针指向合并后的 `nums1` 数组(大小为m+n)的最后一个元素， `p` 指向合并前的 `nums1` 数组(大小为m)的最后一个元素， `q` 指向 `nums2` 数组(大小为n)的最后一个元素。
2. 比较 `p` 指向的值和 `q` 指向的值，将大的值挪进 `nums1[cur]` 。
3. `cur` 指针往前挪， `p` 或者 `q` 指针也相应往前挪。
4. 循环以上步骤直到p=0或q=0
5. 若q>=0,将 `nums2` 数组剩余的元素挪进 `nums1` 。

```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        int p = m - 1 , q = n - 1 , cur = m + n - 1 ;
        while(p>= 0 && q >= 0){
```

```
            nums1[cur--] = ( nums1[p] >= nums2[q] ? nums1[p--] : nums2[q--]);
        }
        while(q >= 0){
            nums1[cur--] = nums2[q--];
        }
    }
};
**********************************************************************************
//把两数组的中的较大值放入cur，指针都往前走一步
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        int p = m - 1,q = n - 1,cur = m + n - 1;
        while(p >= 0 && q >= 0)
        {
            if(nums1[p] >= nums2[q]) nums1[cur --] = nums1[p --];
            else nums1[cur --] = nums2[q --];
        }
        while(p >= 0) nums1[cur --] = nums1[p --];
        while(q >= 0) nums1[cur --] = nums2[q --];
    }
};
```

LeetCode 26. 删除排序数组中的重复项

26. 删除排序数组中的重复项

难度 简单 1355

给定一个排序数组，你需要在 **原地** 删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在 **原地** **修改输入数组** 并在使用 O(1) 额外空间的条件下完成。

示例 1:

```
给定数组 nums = [1,1,2],

函数应该返回新的长度 2, 并且原数组 nums 的前两个元素被修改为 1, 2。

你不需要考虑数组中超出新长度后面的元素。
```

示例 2:

```
给定 nums = [0,0,1,1,1,2,2,3,3,4],

函数应该返回新的长度 5, 并且原数组 nums 的前五个元素被修改为 0, 1, 2, 3, 4。

你不需要考虑数组中超出新长度后面的元素。
```

说明:

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以**「引用」**方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

(双指针移动) $O(n)$

1. 如果 `nums` 的长度是 `0` ，直接返回 `0` 。
2. 初始令 `k` 为 `0` ， `i` 从位置 `1` 开始遍历，若发现 `nums[i]` 和 `nums[k]` 不相等，则说明找到新的元素，并且 `nums[++k]` 赋值为 `nums[i]` 。
3. `i` 向后移动直到末尾。

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if (nums.size() == 0)
            return 0;

        int k = 0;
        for (int i = 1; i < nums.size(); i++)
            if (nums[i] != nums[k])
                nums[++k] = nums[i];
```

```
        return k + 1;
    }
};
**********************************************************************
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if(nums.empty()) return 0;
        int k = 1;
        for(int i = 1;i < nums.size();i ++)
        {
            if(nums[i] != nums[i - 1])
                nums[k ++] = nums[i];
        }
        return k;
    }
};
```

LeetCode 76. 最小覆盖子串 !!!

76. 最小覆盖子串

难度 困难 369

给你一个字符串 S、一个字符串 T，请在字符串 S 里面找出：包含 T 所有字母的最小子串。

示例：

输入： S = "ADOBECODEBANC", T = "ABC"
输出： "BANC"

说明：

- 如果 S 中不存这样的子串，则返回空字符串 `""`。
- 如果 S 中存在这样的子串，我们保证它是唯一的答案。

(滑动窗口) $O(n)$

首先用哈希表统计出 T 中所有字符出现的次数，哈希表可以用C++中的 `unordered_map`，不了解用法的同学可以点这里。
然后我们用两个指针 $i, j(i \geq j)$来扫描整个字符串，同时用一个哈希表统计 i, j 之间每种字符出现的次数，每次遍历需要的操作如下：

1. 加入 $s[i]$，同时更新哈希表；
2. 检查 $s[j]$ 是否多余，如果是，则移除 $s[j]$；
3. 检查当前窗口是否已经满足 T 中所有字符，如果是，则更新答案；

时间复杂度分析：两个指针都严格递增，最多移动 n 次，所以总时间复杂度是 $O(n)$。

```
/*
滑动窗口
暴力：从前往后枚举终点，在往前走枚举子串中的字母，开一个hash，进行匹配
```

```
滑动窗口：i变大j一定不会变小
*/
class Solution {
public:
    string minWindow(string s, string t) {
        unordered_map<char, int> hash;//T中每个字母出现次数
        int cnt = 0;
        for (auto c : t)
        {
            if (!hash[c]) cnt ++ ;
            hash[c] ++ ;
        }

        string res = "";
        for (int i = 0, j = 0, c = 0; i < s.size(); i ++ )
        {
            if (hash[s[i]] == 1) c ++ ;
            hash[s[i]] -- ;//s[i]已经出现
            while (c == cnt && hash[s[j]] < 0) hash[s[j ++ ]] ++ ;
            if (c == cnt)
            {
                if (res.empty() || res.size() > i - j + 1) res = s.substr(j, i -
j + 1);
            }
        }

        return res;
    }
};
*****************************************************************************
class Solution {
public:
    string minWindow(string s, string t) {
        unordered_map<char,int> hash;

        for(auto c : t) hash[c] ++;//T中每个字母出现次数
        int cnt = hash.size();//不同字母的个数

        string res;
        for(int i = 0,j = 0,c = 0;i < s.size();i ++)//维护窗口
        {
            if(hash[s[i]] == 1) c ++;//c为已经存在字母的数
            hash[s[i]] --;//新加了字母
            while(hash[s[j]] < 0) hash[s[j ++ ]] ++;
            if(c == cnt)
            {
                if(res.empty() || res.size() > i - j + 1) res = s.substr(j,i - j
+ 1);
            }
        }
        return res;
    }
};
```

LeetCode 32. 最长有效括号

32. 最长有效括号

难度 困难 546

给定一个只包含 '(' 和 ')' 的字符串，找出最长的包含有效括号的子串的长度。

示例 1:

```
输入：“(()”
输出：2
解释：最长有效括号子串为 “()”
```

示例 2:

```
输入：“)()())”
输出：4
解释：最长有效括号子串为 “()()”
```

算法1

(两次线性扫描、贪心) $O(n)$

1. 假设当前从前到后统计合法括号子串，令 `(` 的权值为 `1`，`)` 的权值为 `-1`。首先记录 `start` 为某个起点，则在 `i` 向后移动的过程中，若当前 `[start,i]` 区间和等于 `0`，该字符串是合法的，更新答案；若区间和大于 `0`，则说明目前缺少右括号，可以不修改 `start`；若区间和小于 `0`，则说明区间已经不合法了，需要修正 `start` 为 `i+1`。初始时 `start` 从 `0` 开始即可。
2. 可是对于 `...((((合法)(((` 这种情况，以上算法不能够准确捕捉到最长的合法子串，此时我们逆向考虑，将以上过程反向，从后向前统计，即可处理所有的情况。

时间复杂度

- 两次线性扫描，故时间复杂度为 $O(n)$。

空间复杂度

- 仅需要常数的额外空间。

```
class Solution {
public:
    int longestValidParentheses(string s) {
        int n = s.length();
        int start = 0, val = 0, ans = 0;
        for (int i = 0; i < n; i++) {
            if (s[i] == '(') val++;
            else val--;
            if (val < 0) {
                val = 0;
                start = i + 1;
            }
            else if (val == 0)
                ans = max(ans, i - start + 1);
        }

        start = n - 1; val = 0;
        for (int i = n - 1; i >= 0; i--) {
            if (s[i] == ')') val++;
```

```
            else val--;
            if (val < 0) {
                val = 0;
                start = i - 1;
            }
            else if (val == 0)
                ans = max(ans, start - i + 1);
        }
        return ans;
    }
};
**********************************************************************************
*******
/*
左括号 = 1
右括号 = -1
括号序列合法 === 所有前缀和 >= 0,且总和等于0 ！！！！
每一个括号匹配的括号是一定

start点前枚举这一段的开头
cnt前缀和
（ = 1
） = -1
1.cnt[i] < 0 -----start = i + 1,cnt = 0只要不匹配直接跳过所有的
2.cnt[i] > 0 -----继续做
3.cnt[i] = 0 -----[start,i]是一段合法的括号序列
从左往右做一遍再从右往左做一遍
*/
class Solution {
public:

    int work(string s)
    {
        int res = 0;
        for(int i = 0,start = 0,cnt = 0;i <s.size();i ++)
            if(s[i] == '(') cnt ++;
            else
            {
                cnt --;
                if(cnt < 0) start  = i + 1,cnt = 0;
                else if(!cnt) res = max(res,i - start + 1);
            }
        return res;
    }
    int longestValidParentheses(string s) {
        int res = work(s);
        reverse(s.begin(),s.end());//翻转一下
        for(auto &c : s) c ^= 1;//左括号变右括号，右括号变左括号
        return max(res,work(s));
    }
};
```

算法2

(动态规划) $O(n)$

1. 设 $f(i)$ 为以 i 为结尾的最长合法子串。
2. 初始时，$f(0) = 0$。
3. 转移时，我们仅考虑当前字符是 `)` 的时候。如果上一个字符是 `(`，即 `...()` 结尾的情况，则 $f(i) = f(i-1) + 2$。
4. 如果上一个字符是 `)`，即 `...))` 的情况，则我们通过上一个字符的动规结果，判断是否能匹配末尾的 `)`。判断 `s[i - f(i - 1) - 1]` 是 `(`，即 `...((合法))`，则可以转移 $f(i) = f(i-1) + 2 + f(i - f(i-1) - 2)$。
5. 最终答案为动规数组中的最大值。

时间复杂度

- 状态数为 $O(n)$，每次转移有两种情况，故时间复杂度为 $O(n)$。

空间复杂度

- 需要额外 $O(n)$ 空间的记录状态。

```
class Solution {
public:
    int longestValidParentheses(string s) {
        int n = s.length();
        int start = 0, val = 0, ans = 0;
        for (int i = 0; i < n; i++) {
            if (s[i] == '(') val++;
            else val--;
            if (val < 0) {
                val = 0;
                start = i + 1;
            }
            else if (val == 0)
                ans = max(ans, i - start + 1);
        }

        start = n - 1; val = 0;
        for (int i = n - 1; i >= 0; i--) {
            if (s[i] == ')') val++;
            else val--;
            if (val < 0) {
                val = 0;
                start = i - 1;
            }
            else if (val == 0)
                ans = max(ans, start - i + 1);
        }
        return ans;
    }
};
```

算法3

(栈) $O(n)$

1. 用栈维护当前待匹配的左括号的位置。同时用 `start` 记录一个新的可能合法的子串的起始位置。初始设为 `0`。
2. 遇到左括号，当前位置进栈。
3. 遇到右括号，如果当前栈不空，则当前栈顶出栈。出栈后，如果栈为空，则更新答案 `i - start + 1`；否则更新答案 `i - st.top()`。
4. 遇到右括号且当前栈为空，则当前的 `start` 开始的子串不再可能为合法子串了，下一个合法子串的起始位置是 `i + 1`，更新 `start = i + 1`。

时间复杂度

- 每个位置遍历一次，最多进栈一次，故时间复杂度为 $O(n)$。

空间复杂度

- 需要额外 $O(n)$ 空间的维护栈。

```
class Solution {
public:
    int longestValidParentheses(string s) {
        int n = s.length();
        stack<int> st;

        int start = 0, ans = 0;
        for (int i = 0; i < n; i++) {
            if (s[i] == '(')
                st.push(i);
            else {
                if (!st.empty()) {
                    st.pop();
                    if (st.empty())
                        ans = max(ans, i - start + 1);
                    else
                        ans = max(ans, i - st.top());
                } else {
                    start = i + 1;
                }
            }
        }

        return ans;
    }
};
```

LeetCode 155.最小栈

155. 最小栈

难度 简单 401

设计一个支持 push，pop，top 操作，并能在常数时间内检索到最小元素的栈。

- push(x) -- 将元素 x 推入栈中。
- pop() -- 删除栈顶的元素。
- top() -- 获取栈顶元素。
- getMin() -- 检索栈中的最小元素。

示例:

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.getMin();   --> 返回 -2.
```

算法

(单调栈) $O(1)$

我们除了维护基本的栈结构之外，还需要维护一个单调栈，来实现返回最小值的操作。
下面介绍如何维护单调栈：

- 当我们向栈中压入一个数时，如果该数 $\leq$ 单调栈的栈顶元素，则将该数同时压入单调栈中；否则，不压入，这是由于栈具有先进后出性质，所以在该数被弹出之前，栈中一直存在一个数比该数小，所以该数一定不会被当做最小数输出。
- 当我们从栈中弹出一个数时，如果该数等于单调栈的栈顶元素，则同时将单调栈的栈顶元素弹出。
- 单调栈的栈顶元素，就是当前栈中的最小数。

时间复杂度分析：四种操作都只有常数次入栈出栈操作，所以时间复杂度都是 $O(1)$.

```
/*
前缀和
stack:-2  0  3
stk_min:前i个数的最小值
*/
class MinStack {
public:
    /** initialize your data structure here. */
    stack<int> stackValue;
    stack<int> stackMin;
    MinStack() {

    }

    void push(int x) {
        stackValue.push(x);
        if (stackMin.empty() || stackMin.top() >= x)
            stackMin.push(x);
```

```
    }

    void pop() {
        if (stackMin.top() == stackValue.top()) stackMin.pop();
        stackValue.pop();
    }

    int top() {
        return stackValue.top();
    }

    int getMin() {
        return stackMin.top();
    }
};

/**
 * Your MinStack object will be instantiated and called as such:
 * MinStack obj = new MinStack();
 * obj.push(x);
 * obj.pop();
 * int param_3 = obj.top();
 * int param_4 = obj.getMin();
 */
```

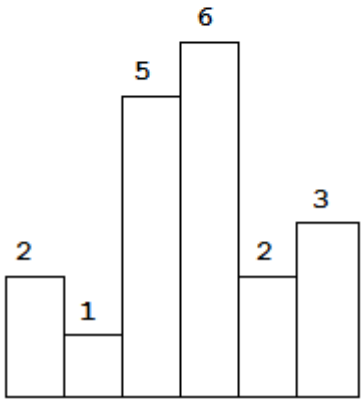
LeetCode 84.柱状图中最大的矩形

84. 柱状图中最大的矩形

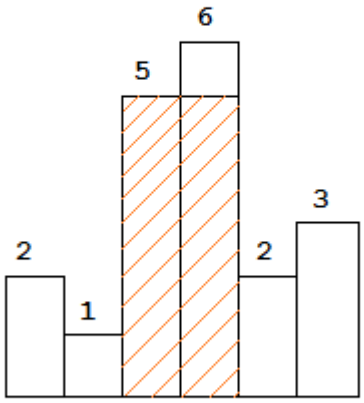
难度 困难 471

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。



以上是柱状图的示例，其中每个柱子的宽度为 1，给定的高度为 `[2,1,5,6,2,3]`。



图中阴影部分为所能勾勒出的最大矩形面积，其面积为 `10` 个单位。

算法

(单调栈) $O(n)$

1. 此题的本质是找到每个柱形条左边和右边最近的比自己低的矩形条，然后用宽度乘上当前柱形条的高度作为备选答案。
2. 解决此类问题的经典做法是单调栈，维护一个单调递增的栈，如果当前柱形条 `i` 的高度比栈顶要低，则栈顶元素 `cur` 出栈。出栈后，`cur` 右边第一个比它低的柱形条就是 `i`，左边第一个比它低的柱形条是当前栈中的 `top`。不断出栈直到栈为空或者柱形条 `i` 不再比 `top` 低。
3. 满足2之后，当前矩形条 `i` 进栈。

```
/*
单调栈：查找每个数左侧第一个比它小的数

如何枚举出所有情况
1.枚举所有柱形的上边界，作为整个矩形的上边界，
```

```
然后求出左右边界
    1.找出左边离它最近的比它小的柱形
    2.找出右边离它最近的比它小的柱形
*/
class Solution {
public:
    int largestRectangleArea(vector<int>& heights) {
        int n = heights.size(), ans = 0;
        heights.push_back(-1);
        // 为了算法书写方便，在数组末尾添加高度 -1
        // 这会使得栈中所有数字在最后出栈。
        stack<int> st;
        for (int i = 0; i <= n; i++) {
            while (!st.empty() && heights[i] < heights[st.top()]) {
                int cur = st.top();
                st.pop();
                if (st.empty())
                    ans = max(ans, heights[cur] * i);
                else
                    ans = max(ans, heights[cur] * (i - st.top() - 1));
            }
            st.push(i);
        }
        return ans;
    }
};

****************************************
class Solution {
public:
    int largestRectangleArea(vector<int>& heights) {
        int n = heights.size();
        vector<int> left(n),rihgt(n);//左右边界

        stack<int> stk;
        //左边第一个比它小的数
        for(int i = 0;i < n;i ++)
        {
            while(stk.size() && heights[stk.top()] >= heights[i]) stk.pop();
            if(stk.empty()) left[i] = -1;//左边没有一个大于等于它的标记-1
            else left[i] = stk.top();
            stk.push(i);
        }
        while(stk.size()) stk.pop();
        //右边第一个比它小的数
        for(int i = n - 1;i >= 0;i --)
        {
            while(stk.size() && heights[stk.top()] >= heights[i]) stk.pop();
            if(stk.empty()) rihgt[i] = n;//右有一个大于等于它的标记n
            else rihgt[i] = stk.top();
            stk.push(i);
        }

        int res = 0;
        //枚举每个边界，取最大值
        for(int i = 0;i < n;i ++) res = max(res,heights[i] * (rihgt[i] - left[i]
- 1));
        return res;
```

```
    }
};
```

LeetCode 42. 接雨水

42. 接雨水

难度 困难 919

给定 *n* 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

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。**

示例:

```
输入: [0,1,0,2,1,0,1,3,2,1,2,1]
输出: 6
```

算法1

(三次线性扫描) $O(n)$

1. 观察整个图形，想办法分解计算水的面积。
2. 注意到，每个矩形条上方所能接受的水的高度，是由它左边**最高的**矩形，和右边**最高的**矩形决定的。具体地，假设第 i 个矩形条的高度为 `height[i]`，且矩形条左边**最高的**矩形条的高度为 `left_max[i]`，右边**最高的**矩形条高度为 `right_max[i]`，则该矩形条上方能接受水的高度为 `min(left_max[i], right_max[i]) - height[i]`。
3. 需要分别从左向右扫描求 `left_max`，从右向左求 `right_max`，最后统计答案即可。
4. 注意特判 `n==0`。

```
class Solution {
public:
    int trap(vector<int>& height) {
        int n = height.size(), ans = 0;
        if (n == 0)
            return 0;
        vector<int> left_max(n), right_max(n);

        left_max[0] = height[0];
        for (int i = 1; i < n; i++)
            left_max[i] = max(left_max[i - 1], height[i]);

        right_max[n - 1] = height[n - 1];
        for (int i = n - 2; i >= 0; i--)
            right_max[i] = max(right_max[i + 1], height[i]);
```

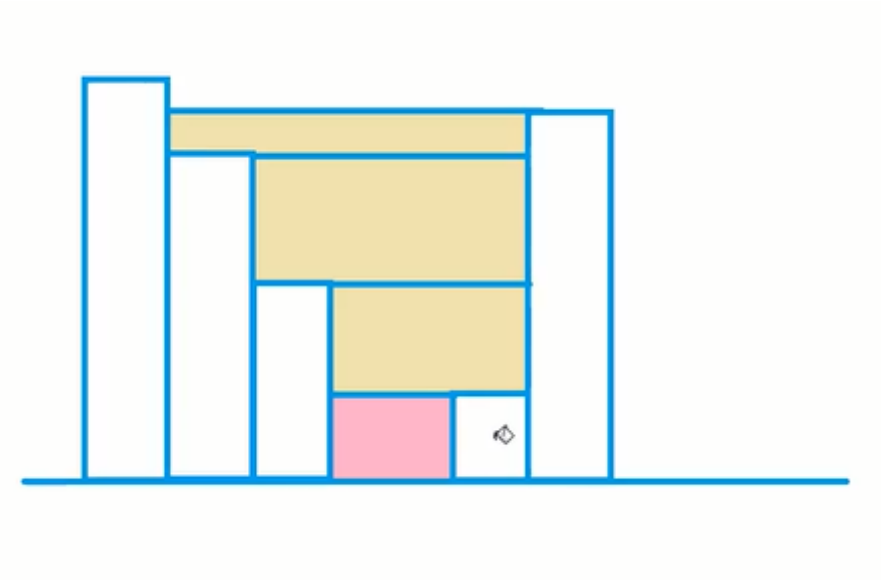
```
        for (int i = 0; i < n; i++)
            ans += min(left_max[i], right_max[i]) - height[i];

        return ans;
    }
};
```

算法2

(单调栈) $O(n)$

1. 换一种思路，考虑每个位置左边和右边 `第一个` 比它高的位置的矩形条，以及三个矩形条构成的 `U` 型。
2. 维护单调递减的单调栈，在每次出栈时， `i` 即为当前栈顶 `st.top()` 位置第一个比它高的矩形的位置，弹出栈顶，并将当前栈顶记为 `top` 。
3. 假设此时栈中仍然存在矩形，现在 `st.top()` 、 `top` 与 `i` 三个位置构成一个 `U` 型，其中 `top` 位置代表 `U` 型的底部，此时可以计算出该 `U` 型所能接受的水的面积为
   $(i - st.top() - 1) * (min(height[st.top()], height[i]) - height[top])$。
4. 如果想不清楚，建议根据代码在纸上模拟一下数据 `[3, 0, 0, 1, 0, 2, 0, 4]` ，这个例子中总共会出现五次 `U` 型。



```
/*
一层一层算面积
加上最后一个需要加的面积为橙色的面积
左边第一个比他大的位置，然后累加面积
*/
class Solution {
public:
    int trap(vector<int>& height) {
        int n = height.size(), ans = 0;
        stack<int> st;
        for (int i = 0; i < n; i++) {
            while (!st.empty() && height[st.top()] < height[i]) {
                int top = st.top();
                st.pop();
                if (st.empty()) break;
                ans += (i - st.top() - 1) * (min(height[st.top()], height[i]) -
height[top]);
            }
            st.push(i);
        }
```

```
        return ans;
    }
};
*********************************************
class Solution {
public:
    int trap(vector<int>& height) {
        int res = 0;
        stack<int> stk;

        for(int i = 0;i < height.size();i ++)
        {
            int last = 0;//存储上一层的高度
            while(stk.size() && height[stk.top()] <= height[i])
            {
                int t = stk.top();
                stk.pop();
                res += (i - t - 1) * (height[t] - last);//面积
                last = height[t];
            }

            if(stk.size()) res += (i - stk.top() - 1) * (height[i] - last);//最上面的一小层
            stk.push(i);
        }
        return res;
    }
};
```

LeetCode 239. 滑动窗口最大值

239. 滑动窗口最大值

难度 困难 259

给定一个数组 *nums*，有一个大小为 *k* 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 *k* 个数字。滑动窗口每次只向右移动一位。

返回滑动窗口中的最大值。

示例:

```
输入: nums = [1,3,-1,-3,5,3,6,7], 和 k = 3
输出: [3,3,5,5,6,7]
解释:

  滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

提示:

你可以假设 *k* 总是有效的，在输入数组不为空的情况下，1 ≤ k ≤ 输入数组的大小。

进阶:

你能在线性时间复杂度内解决此题吗?

算法

(单调队列) $O(n)$

1. 使用单调队列求解滑动窗口中的最大值。其中，单调队列是一个普通的双端队列，即队头和队尾都可以添加和弹出元素。我们假设该双端队列的**队头**是整个队列的最大元素所在下标，至**队尾**下标代表的元素值依次降低。
2. 初始时单调队列为空。随着对数组的遍历过程中，每次插入元素前，需要考察两个事情：(1)合法性检查：队头下标如果距离 *i* 超过了 *k*，则应该出队。(2)单调性维护：如果 `nums[i]` 大于或等于队尾元素下标所对应的值，则当前队尾再也不可能充当某个滑动窗口的最大值了，故需要队尾出队。始终保持队中元素从队头到队尾单调递减。
3. 如次遍历一遍数组，队头就是每个滑动窗口的最大值所在下标。

时间复杂度

- 遍历中，每个元素最多进队一次，出队一次，故时间复杂度为$O(n)$。

```
/*
单调队列：滑动窗口中的最大值
暴力：用一个队列模拟窗口，枚举窗口内的最大值
```

```
单调队列：只要前一个数小于等于后一个数就把前一个数删掉，最后是一个单调下降的队列，队头元素一定是最大值。
*/
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        int n = nums.size();
        vector<int> ans;
        deque<int> q;
        for (int i = 0; i < n; i++) {

            while (!q.empty() && i - q.front() >= k)
                q.pop_front();

            while (!q.empty() && nums[i] >= nums[q.back()])
                q.pop_back();

            q.push_back(i);

            if (i >= k - 1)
                ans.push_back(nums[q.front()]);
        }
        return ans;
    }
};
**********************************************************************
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        vector<int> res;
        deque<int> q;
        for(int i = 0;i < nums.size();i ++)
        {
            if(q.size() && i - k + 1 > q.front()) q.pop_front();//队头出窗口删掉
            while(q.size() && nums[q.back()] <= nums[i]) q.pop_back();//队尾元素小于等于当前元素就删掉
            q.push_back(i);//把当前元素加入队列
            if(i >= k - 1) res.push_back(nums[q.front()]);//窗口大小等于k就输出出来
        }
        return res;
    }
};
```

LeetCode 918. 环形子数组的最大和

918. 环形子数组的最大和

难度 中等 57

给定一个由整数数组 `A` 表示的**环形数组** `C`，求 `C` 的非空子数组的最大可能和。

在此处，*环形数组*意味着数组的末端将会与开头相连呈环状。（形式上，当 `0 <= i < A.length` 时 `C[i] = A[i]`，而当 `i >= 0` 时 `C[i+A.length] = C[i]`）

此外，子数组最多只能包含固定缓冲区 `A` 中的每个元素一次。（形式上，对于子数组 `C[i], C[i+1], ..., C[j]`，不存在 `i <= k1, k2 <= j` 其中 `k1 % A.length = k2 % A.length`）

示例 1：

```
输入：[1,-2,3,-2]
输出：3
解释：从子数组 [3] 得到最大和 3
```

示例 2：

```
输入：[5,-3,5]
输出：10
解释：从子数组 [5,5] 得到最大和 5 + 5 = 10
```

示例 3：

```
输入：[3,-1,2,-1]
输出：4
解释：从子数组 [2,-1,3] 得到最大和 2 + (-1) + 3 = 4
```

示例 4：

```
输入：[3,-2,2,-3]
输出：3
解释：从子数组 [3] 和 [3,-2,2] 都可以得到最大和 3
```

算法

(前缀和，单调队列) $O(n)$

1. 将原数组扩充一倍后，这道题可以视为长度最多为 n 最长连续子序列。先对求前缀和数组 sum。
2. 对于以 i 结尾的子数组，其最优答案是 $sum[i] - min(sum[j]), i - n \leq j < i$。在所有以 i 结尾的子数组中找到最大值即为答案。
3. 以上公式可以用单调队列来快速求解。维护一个单调递增的队列，队头元素为最小值，每次循环时首先将不满足长度的队头出队，然后更新当前的答案。
4. 入队时，检查队尾元素与当前 $sum[i]$ 值的大小，如果 $sum[i]$ 小于等于队尾元素，则队尾元素出队。最后 $sum[i]$ 进队。

```
/*
```

```
把大小为n的环展开成大下为2n的链，把环上的数字复制一遍
i端点前长度为n的窗口的最小值
*/
class Solution {
public:
    int maxSubarraySumCircular(vector<int>& A) {
        int n = A.size(), ans = A[0];

        vector<int> sum(2 * n + 1, 0);
        for (int i = 1; i <= 2 * n; i++) {
            if (i <= n)
                sum[i] = sum[i - 1] + A[i - 1];
            else
                sum[i] = sum[i - 1] + A[i - n - 1];
        }

        deque<int> q;
        q.push_back(0);

        for (int i = 1; i <= 2 * n; i++) {
            while (!q.empty() && i - q.front() > n)
                q.pop_front();

            ans = max(ans, sum[i] - sum[q.front()]);

            while (!q.empty() && sum[i] <= sum[q.back()])
                q.pop_back();

            q.push_back(i);
        }

        return ans;
    }
};
************************************************************
class Solution {
public:
    int maxSubarraySumCircular(vector<int>& A) {
        int n = A.size();
        for(int i = 0;i < n;i ++) A.push_back(A[i]);//复制一份儿
        vector<int> sum(n * 2 + 1);
        for(int i = 1;i <= n * 2;i ++) sum[i] = sum[i - 1] + A[i - 1];//前缀和从1
开始

        int res = INT_MIN;
        deque<int> q;
        q.push_back(0);//前0个数的和是0
        for(int i = 1;i <= n * 2;i ++)
        {
            if(q.size() && i - n > q.front()) q.pop_front();//维护窗口大小
            if(q.size()) res = max(res,sum[i] - sum[q.front()]);//队头是最小元素
            while(q.size() && sum[q.back()] >= sum[i]) q.pop_back();//求最小值
            q.push_back(i);
        }
        return res;
    }
};
```