239.滑动窗口最大值

题目大意：

给你一个整数数组 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 。

解题思路：

方法一：还是双指针滑动窗口，不过由于窗口固定长度，因此只需改变左指针即可。
（最大值获取直接用了max函数，貌似题目不让用。。。）

方法二（官方解）：堆
使用大顶堆实时维护一系列元素中的最大值。
Steps:
1.将数组 nums 的前 k 个元素放入优先队列中。
2.向右滑动窗口，把一个新的元素放入优先队列中，此时堆顶的元素就是堆中所有元素的最大值。

时间复杂度分析：

时间复杂度：O(nlogn)，其中 n 是数组 nums 的长度。在最坏情况下，数组 nums 中的元素单调递增，那么最终优先队列中包含了所有元素，没有元素被移除。由于将一个元素放入优先队列的时间复杂度为 O(logn)，因此总时间复杂度为 O(nlogn)。
空间复杂度：O(n)，即为优先队列需要使用的空间。这里所有的空间复杂度分析都不考虑返回的答案需要的 O(n) 空间，只计算额外的空间使用。

完整代码：
方法一：

```
def maxSlidingWindow(self, nums:List[int], k:int) -> List[int]:
    n = len(nums)
    left = 0  # 窗口左指针
    steps = n - k + 1  # 窗口滑动的总步数
    result = []

    for left in range(steps):
        current_list = nums[left:left+k]
        result.append(max(current_list))  # 获取最大值附加至结果中

    return result
```

方法二（官方解）：

```
def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
    n = len(nums)
    # 注意 Python 默认的优先队列是小顶堆（因此取负来得到大顶堆）
    q = [(-nums[i], i) for i in range(k)]
    heapq.heapify(q)  # 将列表转化为堆

    ans = [-q[0][0]]  # 初始化答案列表ans。其中，第一个滑动窗口的最大值就是堆顶元素的负值

    # 开始从第k个元素向后遍历，逐步滑动窗口
    for i in range(k, n):
        heapq.heappush(q, (-nums[i], i))  # 将新元素加入堆（注意是其负值和索引）

        # 如果堆顶元素的索引已经超出了当前滑动窗口的范围（即不再属于当前窗口），则将其从堆中移除。
        while q[0][1] <= i - k:
            heapq.heappop(q)

        # 将当前窗口的最大值（堆顶元素的负值）加入答案列表 ans
        ans.append(-q[0][0])

    return ans
```

76.最小覆盖子串

题目大意：

```
给你一个字符串s、一个字符串t。返回s中涵盖t所有字符的最小子串。如果s中不存在涵盖t所有字符的子串，则返回空字符串""。
注意：
1. 对于 t 中重复字符，我们寻找的子字符串中该字符数量必须不少于 t 中该字符数量。
2. 如果 s 中存在这样的子串，我们保证它是唯一的答案。
```

解题思路：

```
滑动窗口、哈希表
Steps：
1.双指针滑动窗口
    right 从左至右遍历s，扩展窗口，直到窗口内包含了t所有字符
    当窗口满足条件时，尝试移动left来缩小窗口，寻找更小的符合条件的子串。
2.使用哈希表统计字符
    用一个哈希表 t_count 表示 t 中所有的字符以及它们的个数
    用另一个哈希表 window_count 动态维护当前窗口中所有的字符以及它们的个数
    使用一个计数器have来记录当前窗口中满足要求的字符种类数。如果窗口中每种字符的数量都满足或超过了t中对应字符的数量，则窗口有效。
3.更新最小子串
    每当窗口有效时，检查当前窗口是否比之前记录的最小窗口小，如果小则更新最小窗口。
```

时间复杂度分析：

时间复杂度：O(n)，其中 n 是字符串 s 的长度。我们只遍历一次字符串 s，对于每个字符，左右指针最多各移动一次，因此总体时间复杂度是线性的。
空间复杂度：O(m)，其中 m 是 t 中不同字符的种类数。我们使用了 Counter 来记录 t 和窗口中的字符频率，因此空间复杂度主要取决于字符集的大小。最坏情况下，t 中每个字符都是不同的，因此需要 O(m) 的空间。

完整代码：

```
def minWindow(self, s:str, t:str) -> str:
    # 如果t比s长，直接返回空字符串
    if len(s) < len(t):
        return ""

    # 记录 t 中每个字符的频
    t_count = Counter(t)   # Counter 会返回一个字典，其中键是元素，值是元素的出现次数。
    window_count = Counter()

    # 记录符合条件的最小子串的信息
    left, right = 0, 0
    min_len = float('inf')  # 初始化一个非常大的值，这个值的意义是，当开始计算最小子串的长度时，任何合法的子串长度都会比这个初始值小。
    min_substr = ""

    # 需要满足的字符种类数
    required = len(t_count)
    have = 0  # 当前窗口内满足条件的字符种类数

    # 遍历右边界
    while right < len(s):
        # 扩展窗口
        char = s[right]  # 当前字符
        window_count[char] += 1  # 记录当前字符的出现次数

        # ******如果当前字符在t中，且窗口中的数量等于t中的数量，增加have
        if char in t_count and window_count[char] == t_count[char]:
            have += 1

        # 当窗口满足条件时，尝试收缩窗口
        while have == required:
            # 在收缩窗口之前，更新最小子串
            if right - left + 1 < min_len:
                min_len = right - left + 1
                min_substr = s[left:right + 1]

            # 收缩左边界
            window_count[s[left]] -= 1  # 将窗口中 s[left] 字符的计数减1

            # 如果窗口中的字符数量已经小于 t_count 中的需求，表示窗口不再满足条件
```

```
            if s[left] in t_count and window_count[s[left]] <
t_count[s[left]]:
                have -= 1  # 窗口中包含满足条件的字符种类数减少，因为该字符不
再满足 t 中的需求
            left += 1   # 移动左指针，缩小窗口

        # 扩展右边界
        right += 1

    return min_substr
```

53.最大子数组和

题目大意：

```
给你一个整数数组 nums ，请你找出一个具有最大和的连续子数组（子数组最少包含一个元
素），返回其最大和。
子数组是数组中的一个连续部分。
```

解题思路：

```
动态规划----Kadane 算法
Steps:
1.tmp：记录当前子数组的和
2.max_sum：记录目前为止的最大子数组和
3.遍历数组的每个元素：
    * 对于每个元素，决定是否把它加入当前子数组，或者从这个元素重新开始一个新的子
数组（这两者选择最大的）。
    * 更新 max_sum 为当前的最大值。

通过这种方式，我们可以保证每一步的选择都是局部最优的，而最终得到的 max_sum 就是
全局最优的解。
```

时间复杂度分析：

```
时间复杂度：O(n)，其中 n 是数组的长度。我们只遍历了一遍数组。
空间复杂度：O(1)，只用了 tmp 和 max_sum 两个额外的变量，不依赖于输入数组的大
小。
```

完整代码：

```
def maxSubArray(self, nums:List[int]) -> int:
    tmp = nums[0]  # 初始化当前子数组的和
    max_sum = tmp  # 初始化最大和为第一个元素
    n = len(nums)

    # 从第二个元素开始遍历
    for i in range(1,n):
        # 更新当前子数组的和，选择要么扩展当前子数组，要么从当前元素开始新的子数
组
```

```
        # 换言之，看之前的tmp对当前值有无增益，如果没有就从当前值重新开始新的子数组
        tmp = max(nums[i], tmp + nums[i])
        # 更新最大和
        max_sum = max(max_sum, tmp)

    return max_sum
```

56.合并区间

题目大意：

```
以数组 intervals 表示若干个区间的集合，其中单个区间为 intervals[i] = [starti, endi] 。请你合并所有重叠的区间，并返回一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间。
```

解题思路：

```
按照左端点对各区间进行排序
Steps:
1. 定义数组merge存储最终返回答案
2. 按照左端点对各区间进行排序
3. 将第一个区间加入merge中，按顺序依次考虑之后的每个区间：
    1.如果当前区间左端点在merge中最后一个区间的右端点之后，则两个区间不重合：直接将当前区间加入merge末尾
    2.否则，两个区间重合：用当前区间的右端点更新merge中最后一个区间的右端点（置为二者中较大值）
```

时间复杂度分析：

```
时间复杂度：o(nlogn)，其中n为区间数量，主要开销为排序的o(nlogn)
空间复杂度：o(logn)，其中n为区间的数量，主要为排序所需要的空间复杂度
```

完整代码：

```
def merge(self, intervals:List[List[int]]) -> List[List[int]]:
    # 排序(按照左端点)
    intervals.sort(key=lambda x:x[0])  # key 参数接收一个函数，lambda 函数是一个匿名函数，它接收列表中的每个元素 x（这里是一个区间元组），并返回 x[0]，即每个区间的第一个元素。排序时，Python 会根据这些返回值进行排序。

    merged = []  # 存储最终答案
    # 遍历每一个区间
    for interval in intervals:
        # 如果列表为空，或者当前区间与上一区间不重合，直接添加
        if not merged or merged[-1][1] < interval[0]:
            merged.append(interval)
        else:
            # 否则，与上一个区间合并(右端点取两区间右端点中较大值)
            merged[-1][1] = max(merged[-1][1], interval[1])
```

```
    return merged
```

189.轮转数组

题目大意：

```
给定一个整数数组 nums，将数组中的元素向右轮转 k 个位置，其中 k 是非负数。
```

解题思路：

```
使用额外的数组
1.遍历原数组(长度为n)，将原数组下标为i的元素放至新数组下标为(i+k)%n的位置
2.将新数组拷贝至原数组
```

时间复杂度分析：

```
时间复杂度：o(n)
空间复杂度：o(n)
```

完整代码：

```
def rotate(self, nums:List[int], k:int) -> None:
    n = len(nums)
    tmp = [0] * n  # 定义数组存储结果

    # 遍历原数组，并将各元素轮转后赋给tmp
    for i in range(n):
        tmp[(i+k)%n] = nums[i]

    nums[:] = tmp
```