LeetCode 滑动窗口（Sliding Window）类问题总结

导语

> 滑动窗口类问题是面试当中的高频题，问题本身其实并不复杂，但是实现起来细节思考非常的多，想着想着可能因为变量变化，指针移动等等问题，导致程序反复删来改去，有思路，但是程序写不出是这类问题最大的障碍。本文会将 LeetCode 里面的大部分滑动窗口问题分析、总结、分类，并提供一个可以参考的模版，相信可以有效减少面试当中的算法实现部分的不确定性。

题目概览

滑动窗口这类问题一般需要用到双指针来进行求解，另外一类比较特殊则是需要用到特定的数据结构，像是 sorted_map。后者有特定的题型，后面会列出来，但是，对于前者，题形变化非常的大，一般都是基于字符串和数组的，所以我们重点总结这种基于双指针的滑动窗口问题。

题目问法大致有这几种：

- 给两个字符串，一长一短，问其中短的是否在长的中满足一定的条件存在，例如：
  - 求长的的最短子串，该子串必须涵盖短的的所有字符
  - 短的的 anagram 在长的中出现的所有位置
  - ...
- 给一个字符串或者数组，问这个字符串的子串或者子数组是否满足一定的条件，例如：
  - 含有少于 k 个不同字符的最长子串
  - 所有字符都只出现一次的最长子串
  - ...

除此之外，还有一些其他的问法，但是不变的是，这类题目脱离不开主串（主数组）和子串（子数组）的关系，要求的时间复杂度往往是 $O(n)$，空间复杂度往往是常数级的。之所以是滑动窗口，是因为，遍历的时候，两个指针一前一后夹着的子串（子数组）类似一个窗口，这个窗口大小和范围会随着前后指针的移动发生变化。

解题思路

根据前面的描述，滑动窗口就是这类题目的重点，换句话说，窗口的移动就是重点。我们要控制前后指针的移动来控制窗口，这样的移动是有条件的，也就是要想清楚在什么情况下移动，在什么情况下保持不变，我在这里讲讲我的想法，我的思路是保证右指针每次往前移动一格，每次移动都会有新的一个元素进入窗口，这时条件可能就会发生改变，然后根据当前条件来决定左指针是否移动，以及移动多少格。我写来一个模版在这里，可以参考：

```
public int slidingWindowTemplate(String[] a, ...) {
    // 输入参数有效性判断
    if (...) {
        ...
    }

    // 申请一个散列，用于记录窗口中具体元素的个数情况
    // 这里用数组的形式呈现，也可以考虑其他数据结构
    int[] hash = new int[...];

    // 预处理(可省)，一般情况是改变 hash
    ...

    // l 表示左指针
    // count 记录当前的条件，具体根据题目要求来定义
    // result 用来存放结果
    int l = 0, count = ..., result = ...;
    for (int r = 0; r < A.length; ++r) {
        // 更新新元素在散列中的数量
        hash[A[r]]--;

        // 根据窗口的变更结果来改变条件值
        if (hash[A[r]] == ...) {
            count++;
```

```
        }

        // 如果当前条件不满足，移动左指针直至条件满足为止
        while (count > K || ...) {
            ...
            if (...) {
                count--;
            }
            hash[A[l]]++;
            l++;
        }

        // 更新结果
        results = ...
    }

    return results;
}
```

这里面的 “移动左指针直至条件满足” 部分，需要具体题目具体分析，其他部分的变化不大，我现在来看看具体的题目。

具体题目分析及代码

438. Find All Anagrams in a String

别看这是一道 easy 难度的题目，如果限定你在 $O(n)$ 时间复杂度内实现呢？按照模版会很简单，首先窗口是固定的，窗口长度就是输入参数中第二个字符串的长度，也就是说，右指针移动到某个位置后，左指针必须跟着一同移动，且每次移动都是一格，模版中 count 用来记录窗口内满足条件的元素，直到 count 和窗口长度相等即可更新答案

```
public List<Integer> findAnagrams(String s, String p) {
    if (s.length() < p.length()) {
        return new ArrayList<Integer>();
    }

    char[] sArr = s.toCharArray();
    char[] pArr = p.toCharArray();

    int[] hash = new int[26];

    for (int i = 0; i < pArr.length; ++i) {
        hash[pArr[i] - 'a']++;
    }

    List<Integer> results = new ArrayList<>();

    int l = 0, count = 0, pLength = p.length();
    for (int r = 0; r < sArr.length; ++r) {
        hash[sArr[r] - 'a']--;

        if (hash[sArr[r] - 'a'] >= 0) {
            count++;
        }

        if (r > pLength - 1) {
            hash[sArr[l] - 'a']++;

            if (hash[sArr[l] - 'a'] > 0) {
                count--;
            }

            l++;
        }
```

```
            if (count == pLength) {
                results.add(l);
            }
        }

        return results;
    }
```

76. Minimum Window Substring

同样是两个字符串之间的关系问题，因为题目求的最小子串，也就是窗口的最小长度，说明这里的窗口大小是可变的，这里移动左指针的条件变成，只要左指针指向不需要的字符，就进行移动

```
    public String minWindow(String s, String t) {
        if (s.length() < t.length()) {
            return "";
        }

        char[] sArr = s.toCharArray();
        char[] tArr = t.toCharArray();

        int[] hash = new int[256];

        for (int i = 0; i < tArr.length; ++i) {
            hash[tArr[i]]++;
        }

        int l = 0, count = tArr.length, max = s.length() + 1;
        String result = "";
        for (int r = 0; r < sArr.length; ++r) {
            hash[sArr[r]]--;

            if (hash[sArr[r]] >= 0) {
```

```
                count--;
            }

            while (l < r && hash[sArr[l]] < 0) {
                hash[sArr[l]]++;
                l++;
            }

            if (count == 0 && max > r - l + 1) {
                max = r - l + 1;
                result = s.substring(l, r + 1);
            }
        }

        return result;
    }
```

159. Longest Substring with At Most Two Distinct Characters

这题窗口大小还是变化的，重点还是在何时更新左指针。这里的 count 可以用来记录当前有多少种不同的字符，只要当前进入窗口的字符使 count 不满足题目条件，我们就移动左指针让 count 满足条件。这里 result 初始化为 1，因为只要输入的字串不是空的，窗口大小不可能小于 1。

```
    public int lengthOfLongestSubstringTwoDistinct(String s) {
        if (s == null || s.length() == 0) {
            return 0;
        }

        char[] sArr = s.toCharArray();

        int[] hash = new int[256];

        int l = 0, count = 0, result = 1;
```

```
        for (int r = 0; r < sArr.length; ++r) {
            hash[sArr[r]]++;

            if (hash[sArr[r]] == 1) {
                count++;
            }

            while (count > 2) {
                hash[sArr[l]]--;

                if (hash[sArr[l]] == 0) {
                    count--;
                }

                l++;
            }

            result = Math.max(result, r - l + 1);
        }

        return result;
    }
```

340. Longest Substring with At Most K Distinct Characters

和上题一样，把 2 变成 k 即可

```
    public int lengthOfLongestSubstringKDistinct(String s, int k) {
        if (s == null || s.length() == 0 || k == 0) {
            return 0;
        }

        char[] sArr = s.toCharArray();
```

```
        int[] hash = new int[256];

        int l = 0, count = 0, result = 1;
        for (int r = 0; r < sArr.length; ++r) {
            hash[sArr[r]]++;

            if (hash[sArr[r]] == 1) {
                count++;
            }

            while (count > k) {
                hash[sArr[l]]--;

                if (hash[sArr[l]] == 0) {
                    count--;
                }

                l++;
            }

            result = Math.max(result, r - l + 1);
        }

        return result;
    }
```

3. Longest Substring Without Repeating Characters

输入只有一个字符串，要求子串里面不能够有重复的元素，这里 count 都不需要定义，直接判断哈希散列里面的元素是不是在窗口内即可，是的话得移动左指针去重。

```
public int lengthOfLongestSubstring(String s) {
    if (s == null || s.length() == 0) {
        return 0;
    }

    char[] sArr = s.toCharArray();
    int[] hash = new int[256];

    int l = 0, result = 1;
    for (int r = 0; r < sArr.length; ++r) {
        hash[sArr[r]]++;

        while (hash[sArr[r]] != 1) {
            hash[sArr[l]]--;
            l++;
        }

        result = Math.max(result, r - l + 1);
    }

    return result;
}
```

567. Permutation in String

和 438 那题很类似，但是这里不需要记录答案了，有就直接返回 true。

```
public boolean checkInclusion(String s1, String s2) {
    if (s1.length() > s2.length()) {
        return false;
    }
```

```
        char[] s1Arr = s1.toCharArray();
        char[] s2Arr = s2.toCharArray();

        int[] hash = new int[26];

        for (int i = 0; i < s1Arr.length; ++i) {
            hash[s1Arr[i] - 'a']++;
        }

        int l = 0, count = 0;
        for (int r = 0; r < s2Arr.length; ++r) {
            hash[s2Arr[r] - 'a']--;

            if (hash[s2Arr[r] - 'a'] >= 0) {
                count++;
            }

            if (r >= s1Arr.length) {
                hash[s2Arr[l] - 'a']++;

                if (hash[s2Arr[l] - 'a'] >= 1) {
                    count--;
                }

                l++;
            }

            if (count == s1Arr.length) {
                return true;
            }
        }

        return false;
    }
```

992. Subarrays with K Different Integers

看完了字符串类型的题目，这次来看看数组类型的，题目中的 subarray 已经明确了这个题可以考虑用滑动窗口，这题比较 trick 的一个地方在于，这里不是求最小值最大值，而是要你计数，但是如果每次仅仅加 1 的话又不太对，例如 `A = [1,2,1,2,3], K = 2` 这个例子，假如右指针移到 index 为 3 的位置，如果按之前的思路左指针根据 count 来移动，当前窗口是 `[1,2,1,2]`，但是怎么把 `[2,1]` 给考虑进去呢？可以从数组和子数组的关系来思考，假如 `[1,2,1,2]` 是符合条件的数组，如果要计数的话，`[1,2,1,2]` 要求的结果是否和 `[1,2,1]` 的结果存在联系？这两个数组的区别在于多了一个新进来的元素，之前子数组计数没考虑到这个元素，假如把这个元素放到之前符合条件的子数组中组成的新数组也是符合条件的，我们看看这个例子中所有满足条件的窗口以及对应的满足条件的子数组情况：

```
[1,2,1,2,3]  // 窗口满足条件
 l r         // 满足条件的子数组 [1,2]

[1,2,1,2,3]  // 窗口满足条件
 l   r       // 满足条件的子数组 [1,2],[2,1],[1,2,1]

[1,2,1,2,3]  // 窗口满足条件
 l     r     // 满足条件的子数组 [1,2],[2,1],[1,2,1],[1,2],[2,1,2],[1,2,1,2]

[1,2,1,2,3]  // 窗口不满足条件，移动左指针至满足条件
 l       r

[1,2,1,2,3]  // 窗口满足条件
       l r   // 满足条件的子数组 [1,2],[2,1],[1,2,1],[1,2],[2,1,2],[1,2,1,2],[2,3]
```

你可以看到对于一段连续的数组，新的元素进来，窗口增加 1，每次的增量都会在前一次增量的基础上加 1。当新的元素进来打破当前条件会使这个增量从新回到 1，这样我们左指针移动条件就是只要是移动不会改变条件，就移动，不然就停止

```
public int subarraysWithKDistinct(int[] A, int K) {
    if (A == null || A.length < K) {
```

```
            return 0;
        }

        int[] hash = new int[A.length + 1];

        int l = 0, results = 0, count = 0, result = 1;
        for (int r = 0; r < A.length; ++r) {
            hash[A[r]]++;

            if (hash[A[r]] == 1) {
                count++;
            }

            while (hash[A[l]] > 1 || count > K) {
                if (count > K) {
                    result = 1;
                    count--;
                } else {
                    result++;
                }
                hash[A[l]]--;
                l++;
            }

            if (count == K) {
                results += result;
            }
        }

        return results;
    }
```

424. Longest Repeating Character Replacement

这道题想 accept 的话不难，但是问题在于怎么知道当前窗口中数量最多的字符的数量，因为需要替换的字符就是当前窗口的大小减去窗口中数量最多的字符的数量。最简单的方法就是把哈希散列遍历一边找到最大的字符数量，但是仔细想想如果我们每次新进元素都更新这个最大数量，且只更新一次，我们保存的是当前遍历过的全局的最大值，它肯定是比实际的最大值大的，我们左指针移动的条件是 `r - l + 1 - maxCount > k`，保存的结果是 `result = Math.max(r - l + 1, result);` 这里 maxCount 比实际偏大的话，虽然导致左指针不能移动，但是不会记录当前的结果，所以最后的答案并不会受影响

```
public int characterReplacement(String s, int k) {
    if (s == null || s.length() == 0) {
        return 0;
    }

    char[] sArr = s.toCharArray();

    int[] hash = new int[26];

    int l = 0, maxCount = 0, result = 0;
    for (int r = 0; r < sArr.length; ++r) {
        hash[sArr[r] - 'A']++;

        maxCount = Math.max(maxCount, hash[sArr[r] - 'A']);

        while (r - l + 1 - maxCount > k) {
            hash[sArr[l] - 'A']--;
            l++;
        }

        result = Math.max(r - l + 1, result);
    }

    return result;
}
```

其他题型

239. Sliding Window Maximum

可以参考我之前写的 Arts 的算法部分 -> Arts 第三周

480. Sliding Window Median

求中值的问题比较特殊，如果是乱序的情况下最好就是借助堆，利用两个堆，一个大顶，一个小顶，并且保证两个堆内存放的数目相差不超过 1，这样如果当前元素数目是偶数，两个堆顶元素的均值就是结果，如果是奇数，存放元素多的那个堆的堆顶元素就是结果。

```
public double[] medianSlidingWindow(int[] nums, int k) {
    if (nums == null || nums.length < k ) {
        return new double[0];
    }

    double[] results = new double[nums.length - k + 1];

    PriorityQueue<Integer> minHeap = new PriorityQueue<>();

    PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder

    for (int i = 0; i < nums.length; ++i) {
        // add current element into queue
        maxHeap.offer(nums[i]);
        minHeap.offer(maxHeap.poll());

        if (minHeap.size() > maxHeap.size()) {
            maxHeap.offer(minHeap.poll());
        }

        // record answer
        if (i >= k - 1) {
```

```
            results[i - k + 1] = minHeap.size() < maxHeap.size()
                    ? maxHeap.peek() : ((long)maxHeap.peek() + minHeap.peek()) *

            if (maxHeap.contains(nums[i - k + 1])) {
                maxHeap.remove(nums[i - k + 1]);
            } else {
                minHeap.remove(nums[i - k + 1]);
            }
        }
    }

    return results;
}
```

总结

双指针类的滑动窗口问题思维复杂度并不高，但是出错点往往在细节。记忆常用的解题模版还是很有必要的，特别是对于这种变量名多，容易混淆的题型。有了这个框架，思考的点就转化为 “什么条件下移动左指针”，无关信息少了，思考加实现自然不是问题。