滑动窗口算法基本原理与实践 - huansky - 博客园

cnblogs.com/huansky/p/13488234.html

滑动窗口算法基本原理与实践

学过计算机网络的同学，都知道滑动窗口协议（Sliding Window Protocol），该协议是 TCP协议 的一种应用，用于网络数据传输时的流量控制，以避免拥塞的发生。该协议允许发送方在停止并等待确认前发送多个数据分组。由于发送方不必每发一个分组就停下来等待确认。因此该协议可以加速数据的传输，提高网络吞吐量。

滑动窗口算法其实和这个是一样的，只是用的地方场景不一样，可以根据需要调整窗口的大小，有时也可以是固定窗口大小。

滑动窗口算法（Sliding Window Algorithm）

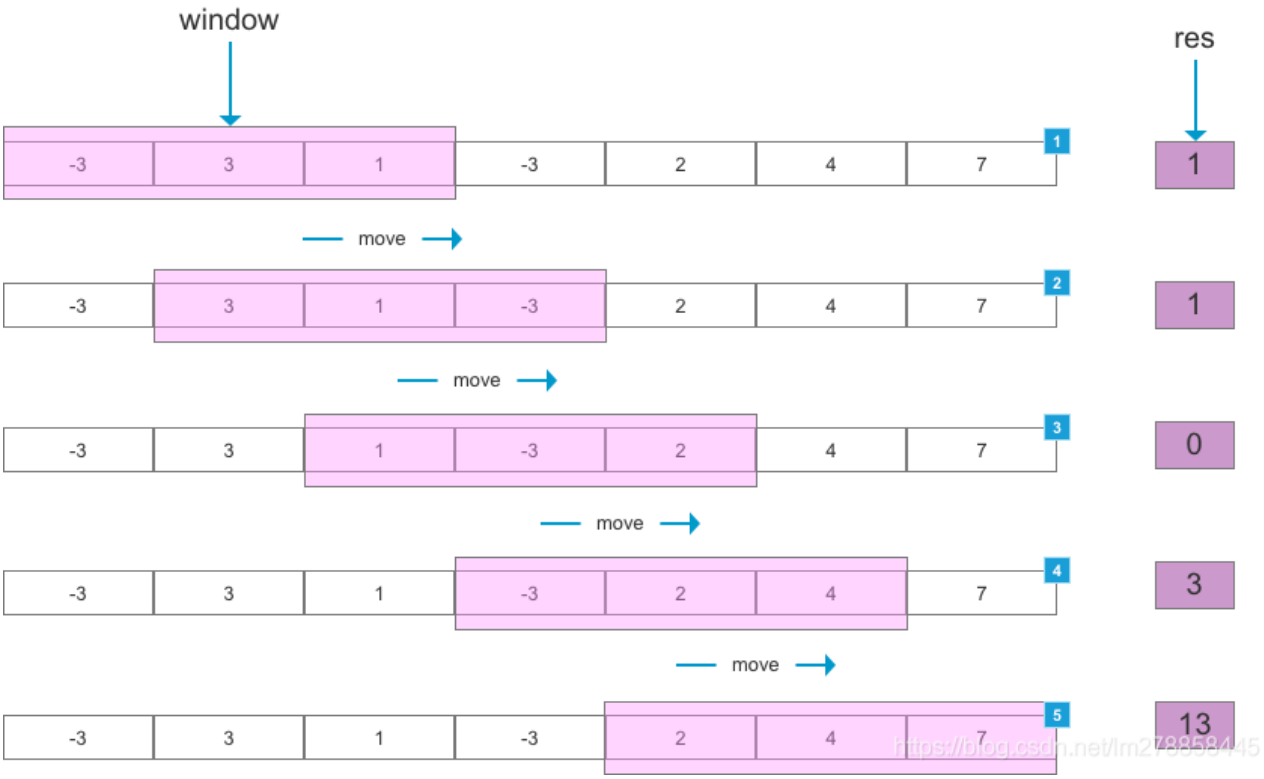
> Sliding window algorithm is used to perform required operation on specific window size of given large buffer or array.
>
> 滑动窗口算法是在给定特定窗口大小的数组或字符串上执行要求的操作。

> This technique shows how a nested for loop in few problems can be converted to single for loop and hence reducing the time complexity.
>
> 该技术可以将一部分问题中的嵌套循环转变为一个单循环，因此它可以减少时间复杂度。

简而言之，滑动窗口算法在一个特定大小的字符串或数组上进行操作，而不在整个字符串和数组上操作，这样就降低了问题的复杂度，从而也达到降低了循环的嵌套深度。其实这里就可以看出来滑动窗口主要应用在数组和字符串上。

基本示例

如下图所示，设定滑动窗口（window）大小为 3，当滑动窗口每次划过数组时，计算当前滑动窗口中元素的和，得到结果 res。



可以用来解决一些查找满足一定条件的连续区间的性质（长度等）的问题。由于区间连续，因此当区间发生变化时，可以通过旧有的计算结果对搜索空间进行剪枝，这样便减少了重复计算，降低了时间复杂度。往往类似于“ 请找到满足 xx 的最 x 的区间（子串、子数组）的 xx ”这类问题都可以使用该方法进行解决。

需要注意的是，滑动窗口算法更多的是一种思想，而非某种数据结构的使用。

滑动窗口法的大体框架

在介绍滑动窗口的框架时候，大家先从字面理解下：

- **滑动**：说明这个窗口是移动的，也就是移动是按照一定方向来的。
- **窗口**：窗口大小并不是固定的，可以不断扩容直到满足一定的条件；也可以不断缩小，直到找到一个满足条件的最小窗口；当然也可以是固定大小。

为了便于理解，这里采用的是字符串来讲解。但是对于数组其实也是一样的。滑动窗口算法的思路是这样：

1. 我们在字符串 S 中使用双指针中的左右指针技巧，初始化 left = right = 0，把索引闭区间 [left, right] 称为一个「窗口」。
2. 我们先不断地增加 right 指针扩大窗口 [left, right]，直到窗口中的字符串符合要求（包含了 T 中的所有字符）。

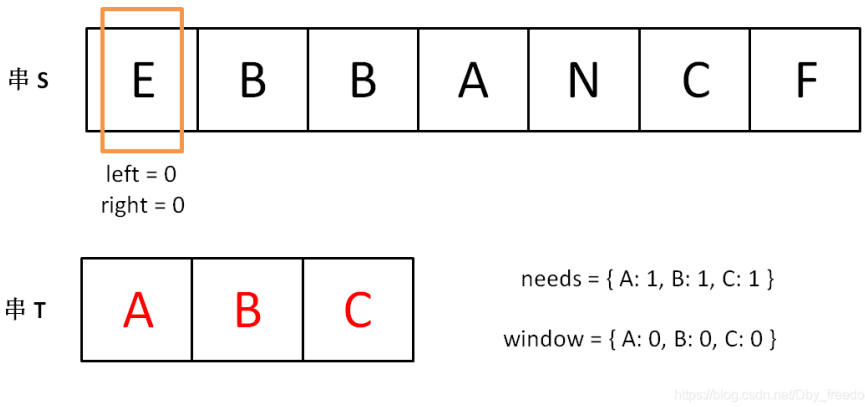
3. 此时，我们停止增加 right，转而不断增加 left 指针缩小窗口 [left, right]，直到窗口中的字符串不再符合要求（不包含 T 中的所有字符了）。同时，每次增加 left，我们都要更新一轮结果。

4. 重复第 2 和第 3 步，直到 right 到达字符串 S 的尽头。

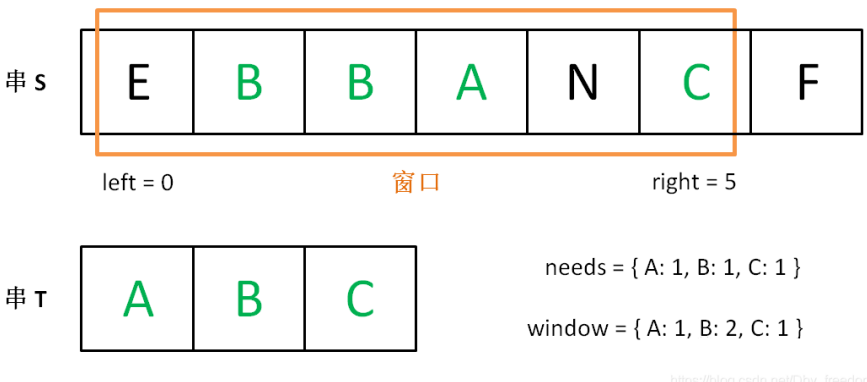
这个思路其实也不难，第 2 步相当于在寻找一个「可行解」，然后第 3 步在优化这个「可行解」，最终找到最优解。左右指针轮流前进，窗口大小增增减减，窗口不断向右滑动。

下面画图理解一下，needs 和 window 相当于计数器，分别记录 T 中字符出现次数和窗口中的相应字符的出现次数。

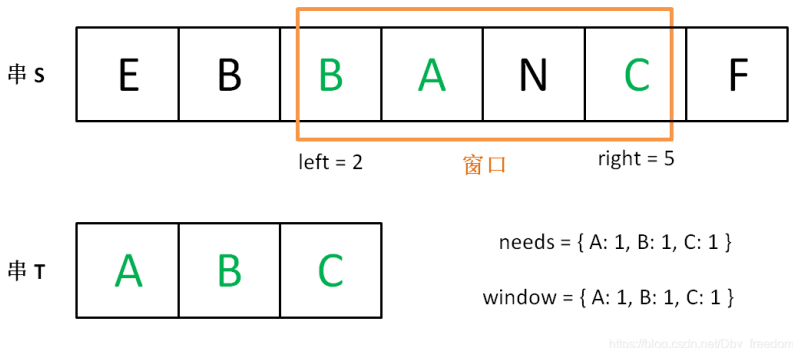
初始状态：



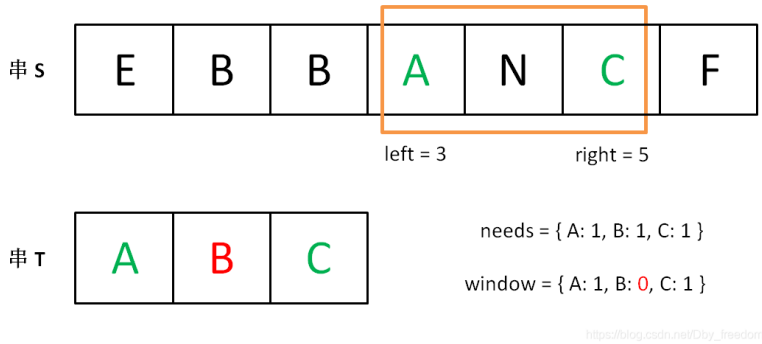
增加 right，直到窗口 [left, right] 包含了 T 中所有字符：



现在开始增加 left，缩小窗口 [left, right]。



直到窗口中的字符串不再符合要求，left 不再继续移动。



之后重复上述过程，先移动 right，再移动 left..... 直到 right 指针到达字符串 S 的末端，算法结束。

如果你能够理解上述过程，恭喜，你已经完全掌握了滑动窗口算法思想。至于如何具体到问题，如何得出此题的答案，都是编程问题，等会提供一套模板，理解一下就会了。

上述过程对于非固定大小的滑动窗口，可以简单地写出如下伪码框架：

```
string s, t;
// 在 s 中寻找 t 的「最小覆盖子串」
int left = 0, right = 0;
string res = s;

while(right < s.size()) {
    window.add(s[right]);
    right++;
    // 如果符合要求，说明窗口构造完成，移动 left 缩小窗口
    while (window 符合要求) {
        // 如果这个窗口的子串更短，则更新 res
        res = minLen(res, window);
        window.remove(s[left]);
        left++;
    }
}
return res;
```

但是，对于固定窗口大小，可以总结如下：

```
// 固定窗口大小为 k
 string s;
 // 在 s 中寻找窗口大小为 k 时的所包含最大元音字母个数
 int  right = 0;while(right < s.size()) {
     window.add(s[right]);
     right++;
     // 如果符合要求，说明窗口构造完成，
     if (right>=k) {
         // 这是已经是一个窗口了，根据条件做一些事情
        // ... 可以计算窗口最大值等
         // 最后不要忘记把 right -k 位置元素从窗口里面移除
     }
 }
 return res;
```

可以发现此时不需要依赖 left 指针了。因为窗口固定所以其实就没必要使用left，right 双指针来控制窗口的大小。

其次是对于窗口是固定的，可以轻易获取到 left 的位置，此处 left = right-k;

实际上，对于窗口的构造是很重要的。具体可以看下面的实例。

算法实例

1208. 尽可能使字符串相等

给你两个长度相同的字符串，s 和 t。

将 s 中的第 i 个字符变到 t 中的第 i 个字符需要 |s[i] - t[i]| 的开销（开销可能为 0），也就是两个字符的 ASCII 码值的差的绝对值。

用于变更字符串的最大预算是 maxCost。在转化字符串时，总开销应当小于等于该预算，这也意味着字符串的转化可能是不完全的。

如果你可以将 s 的子字符串转化为它在 t 中对应的子字符串，则返回可以转化的最大长度。

如果 s 中没有子字符串可以转化成 t 中对应的子字符串，则返回 0。

示例 1:

```
输入：s = "abcd", t = "bcdf", cost = 3
输出：3
解释：s 中的 "abc" 可以变为 "bcd"。开销为 3，所以最大长度为 3。
```

示例 2:

```
输入：s = "abcd", t = "cdef", cost = 3
输出：1
解释：s 中的任一字符要想变成 t 中对应的字符，其开销都是 2。因此，最大长度为 1。
```

示例 3：

```
输入：s = "abcd", t = "acde", cost = 0
输出：1
解释：你无法作出任何改动，所以最大长度为 1。
```

代码

```
class Solution {
    public int equalSubstring(String s, String t, int maxCost) {
        int left = 0, right =0;
        int sum = 0;
        int res = 0;
        // 构造窗口
        while (right < s.length()) {
            sum += Math.abs(s.charAt(right) - t.charAt(right));
            right++;
            // 窗口构造完成，这时候要根据条件当前的窗口调整窗口大小
            while (sum > maxCost) {
                sum -=  Math.abs(s.charAt(left) - t.charAt(left));
                left++;
            }
            // 记录此时窗口的大小
            res = Math.max(res, right -left);
        }
        return res;
    }
}
```

这里跟前面总结的框架不一样的一个点就是，前面的框架是求最小窗口大小，这里是求最大窗口大小，大家要学会灵活变通。

239. 滑动窗口最大值

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。

进阶：

你能在线性时间复杂度内解决此题吗?

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:

```
1 <= nums.length <= 10^5
-10^4 <= nums[i] <= 10^4
1 <= k <= nums.length
```

解答:

```
class Solution {
    public static int[] maxSlidingWindow(int[] nums, int k) {
        int right =0;
        int[] res = new int[nums.length -k +1];
        int index=0;
        LinkedList<Integer> list = new LinkedList<>();
        // 开始构造窗口
        while (right < nums.length) {
            // 这里的list的首位必须是窗口中最大的那位
            while (!list.isEmpty() && nums[right] > list.peekLast()) {
                list.removeLast();
            }
            // 不断添加
            list.addLast(nums[right]);
            right++;
            // 构造窗口完成，这时候需要根据条件做一些操作
            if (right >= k){
                res[index++]=list.peekFirst();
                 // 如果发现第一个已经在窗口外面了，就移除
                if(list.peekFirst() == nums[right-k]) {
                    list.removeFirst();
                }
            }
        }
        return res;
    }
}
```

这道题难度是困难。当然我们也会发现，这道题目和前面的非固定大小滑动窗口还是不一样的。

看了一道困难的题目后，接下来看一道中等难度的就会发现是小菜一碟。

1456. 定长子串中元音的最大数目

给你字符串 s 和整数 k 。

请返回字符串 s 中长度为 k 的单个子字符串中可能包含的最大元音字母数。

英文中的 元音字母 为（a, e, i, o, u）。

示例 1：

```
输入：s = "abciiidef", k = 3
输出：3
解释：子字符串 "iii" 包含 3 个元音字母。
```

示例 2：

```
输入：s = "aeiou", k = 2
输出：2
解释：任意长度为 2 的子字符串都包含 2 个元音字母。
```

示例 3：

```
输入：s = "leetcode", k = 3
输出：2
解释："lee"、"eet" 和 "ode" 都包含 2 个元音字母。
```

示例 4：

```
输入：s = "rhythms", k = 4
输出：0
解释：字符串 s 中不含任何元音字母。
```

示例 5：

```
输入：s = "tryhard", k = 4
输出：1
```

提示：

```
1 <= s.length <= 10^5
s 由小写英文字母组成
1 <= k <= s.length
```

解答

```
class Solution {
    public int maxVowels(String s, int k) {
        int right =0;
        int sum = 0;
        int max = 0;
        while (right < s.length()) {
            sum += isYuan(s.charAt(right)) ;
            right++;
            if (right >=k) {
                max = Math.max(max, sum);
                sum -= isYuan(s.charAt(right-k));
            }
        }
        return max;
    }

    public int isYuan(char s) {
        return s=='a' || s=='e' ||s=='i' ||s=='o' ||s=='u' ? 1:0;
    }
}
```

参考文章

算法与数据结构（一）： 滑动窗口法总结

树林美丽、幽暗而深邃，但我有诺言尚待实现，还要奔行百里方可沉睡。 -- 罗伯特·弗罗斯特

标签: 数据结构和算法
« 上一篇： 分治算法基本原理和实践
» 下一篇： 双指针算法基本原理和实践
刷新评论刷新页面返回顶部
发表评论

编辑 预览

退出 订阅评论

[Ctrl+Enter快捷键提交]

历史上的今天:
2017-08-16 安卓和 java 学习笔记