滑动窗口常用技巧总结

作者：vcjmhg

原文链接：https://ld246.com/article/1605441703995

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

概述

在解决字串问题时，滑动窗口技巧可能经常会使用，其本身思想并不难理解，难在灵活。因而本文从个最小覆盖字串问题入手总结一个通用的算法框架以解决常见的滑动窗口问题。

算法与框架

下边我们先看一个最小覆盖子串问题：

76. 最小覆盖子串

难度 困难 825 收藏 分享 切换为英文 接收动态 反馈

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：如果 `s` 中存在这样的子串，我们保证它是唯一的答案。

示例 1：

```
输入：s = "ADOBECODEBANC", t = "ABC"
输出："BANC"
```

示例 2：

```
输入：s = "a", t = "a"
输出："a"
```

题目本身不难理解，主要就是从S(source)中找到包含T(target)中全部字幕的一个子串，顺序无所谓个数相同且**子串中一定是所有可能子串中最短的。**

最简单的思路是通过**暴力法**，通过两层搜索来解决，但时间复杂度很高，甚至大于O(n^2)。

此类问题实际上我们可以通过滑动窗口的思路来解决。具体思路如下：

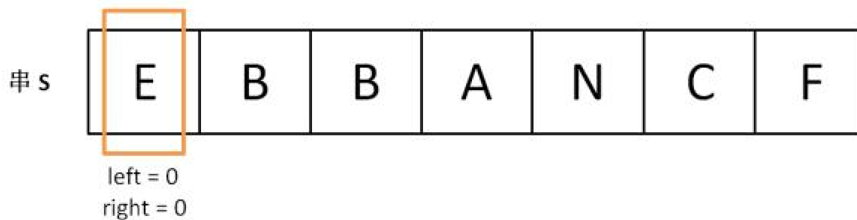
1. 在字符串S中使用双指针中左右指针的技巧，初始化 left = right = 0，把索引区间[left,right]称为一个[窗口]。
2. 不断的增加right指针扩大窗口 [left,right]，直到窗口中的字符串符合要求（窗口包含T中所有字）。
3. 停止增加right，转而 **增加left指针**，进而缩小窗口直到窗口不再符合要求。同时每增加一个left要更新一轮结果。
4. 重复2和3，直到right达到字符串S的尽头。

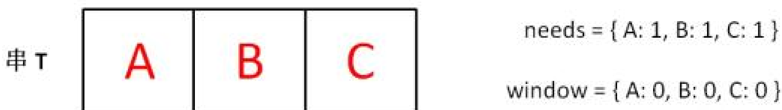
整个过程思路并不难，其中**第2步相当于在找一个可行解，第3步在优化这个可行解，每轮都进行结果新，最后找到最优解。**

下边我们结合整下边的图来理解算法的整个过程。needs和windows相当于计数器，分别记录T中字串出现的次数和窗口中的对应字符出现的次数。

第1步：初始状态，left和righ都为0

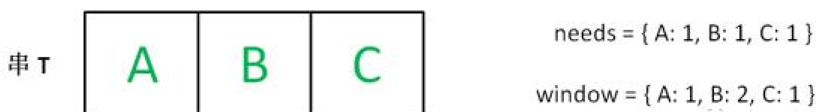
初始状态：





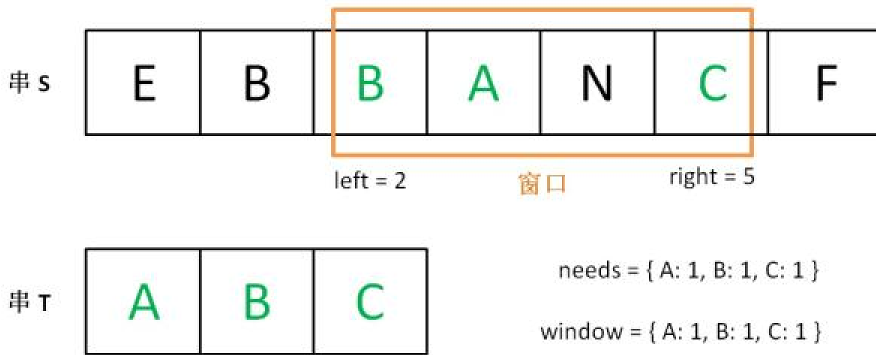
第2步：向右移动right寻找可行解

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



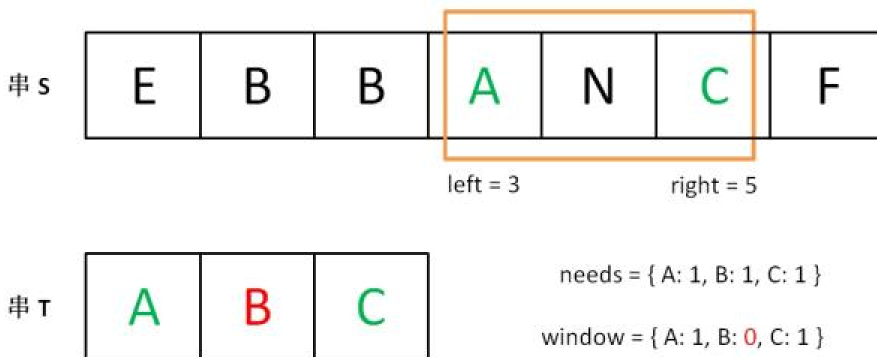
第3步：向右移动left，优化可行解

原文链接：滑动窗口常用技巧总结



第4步：重复2和3直到，right到达右边界

直到窗口中的字符串不再符合要求，left 不再继续移动。



上述过程可以简单写出如下的代码框架：

```
public String slidingWindow(String s, String t) {
    //定义两个窗口
    Map<Character, Integer> need = new HashMap<>(), window = new HashMap<>();
    // 初始化need窗口
    for (char c : t.toCharArray()) {
      need.put(c, need.getOrDefault(c, 0) + 1);
    }

    int left = 0, right = 0;
    // 已经和need匹配的字符串个数
    int valid = 0;
    while (right < s.length()) {
      char c = s.charAt(right);
      // move to right
      right++;
      // 进行窗口内一系列数据的更新
      ...

    // 判断左侧窗口是否要收缩
    while (window needs shrink) {
        // d 是将移出窗口的字符
        char d = s.charAt(left);
        // 左移窗口
        left++;
```

```
        // 进行窗口内数据的一系列更新
        ...
    }
}
```

其中两处...表示更新窗口数据的地方，根据不同的问题，进行填充即可。

针对最小覆盖子串问题，开始套模板，只需要考虑如下四个问题：

1. 移动right扩大窗口，即加入字符时需要考虑哪些数据？
2. 什么条件下，窗口应该暂停扩大，开始移动left缩小窗口？
3. 当移动left缩小窗口，即移除字符时，应该更新哪些数据？
4. 我们要的结果应该在扩大窗口时还是缩小窗口时进行更新？

如果一个字符进入窗口，应该增加 window 计数器；如果一个字符将移出窗口的时候，应该减少 window 计数器；当 valid 满足 need 时应该收缩窗口；应该在收缩窗口的时候更新最终结果。

针对该问题我们将代码进行填充后得到如何解法：

```
string minWindow(string s, string t) {
    unordered_map<char, int> need, window;
    for (char c : t) need[c]++;

    int left = 0, right = 0;
    int valid = 0;
    // 记录最小覆盖子串的起始索引及长度
    int start = 0, len = INT_MAX;
    while (right < s.size()) {
        // c 是将移入窗口的字符
        char c = s[right];
        // 右移窗口
        right++;
        // 进行窗口内数据的一系列更新
        if (need.count(c)) {
            window[c]++;
            if (window[c] == need[c])
                valid++;
        }

        // 判断左侧窗口是否要收缩
        //必须使用equlals来判断，不能使用 ==
        while (valid.equals(need.size())) {
            // 在这里更新最小覆盖子串
            if (right - left < len) {
                start = left;
                len = right - left;
            }
            // d 是将移出窗口的字符
            char d = s[left];
            // 左移窗口
            left++;
            // 进行窗口内数据的一系列更新
            if (need.count(d)) {
```

```
            if (window[d].euqals(need[d]))
                valid--;
            window[d]--;
        }
      }
    }
    // 返回最小覆盖子串
    return len == INT_MAX ?
        "" : s.substr(start, len);
}
```

应用

接下来我们再看一下另一个中等难度的题目**字符串的排列**

567. 字符串的排列

难度 中等 195 收藏 分享 切换为英文 接收动态 反馈

给定两个字符串 **s1** 和 **s2**，写一个函数来判断 **s2** 是否包含 **s1** 的排列。

换句话说，第一个字符串的排列之一是第二个字符串的子串。

示例1:

```
输入: s1 = "ab" s2 = "eidbaooo"
输出: True
解释: s2 包含 s1 的排列之一 ("ba").
```

示例2:

```
输入: s1= "ab" s2 = "eidboaoo"
输出: False
```

题意很好理解，就是判断s2是否包含s1的某种排列。我们比较容易想到用暴力法。但会发现时间复杂过高无法通过。然后考虑到是子串问题，尝试使用滑动窗口方法。

结合模板，考虑两个问题：

1. 右侧窗口滑动时，做哪些操作
2. 左侧窗口滑动的条件，以及所做操作

针对第一个问题，我们考虑到当右侧窗口滑动获取一个字符时要判断当前字符是否在need中，如果在进行windows计数

针对第二个问题，如果窗口的长度大于字符串t的长度，则需要进行窗口左移操作，进行窗口“瘦身”

该问题具体代码实现如下：

```
public boolean checkInclusion(String t, String s) {
    if (t.length() > s.length()) {
      return false;
    }
    Map<Character, Integer> need = new HashMap<>();
```

```
    Map<Character, Integer> window = new HashMap<>();
    // init need
    for (char c : t.toCharArray()) {
      need.put(c, need.getOrDefault(c, 0) + 1);
    }
    // define variable
    int left = 0, right = 0;
    int valid = 0;
    while (right < s.length()) {
      char c = s.charAt(right);
      right++;
      // update right window
      if (need.containsKey(c)) {
        window.put(c, window.getOrDefault(c, 0) + 1);
        if (need.get(c).equals(window.get(c))) {
          valid++;
        }
      }
      // shrink left window
      // 每一次窗口的尺寸比need的尺寸大的时候都会进行瘦身操作,一直移动到比need的尺寸小1结束
      while (right - left >= t.length()) {
        if (valid == need.size()) {
          return true;
        }
        char d = s.charAt(left);
        left++;
        if (need.containsKey(d)) {
          if (window.get(d).equals(need.get(d))) {
            valid--;
          }
          window.put(d, window.get(d) - 1);
        }
      }
    }
    return false;
  }
```

总结

简单来说滑动窗口问题其实只要记下这个框架，大部分类似问题都可迎刃而解。

参考

1. https://labuladong.gitbook.io/algo/shu-ju-jie-gou-xi-lie/2.5-shou-ba-shou-shua-shu-zu-timu/hua-dong-chuang-kou-ji-qiao-jin-jie