H1 滑动窗口

H2 模板

```cpp
/* 滑动窗口算法框架 */
void slidingWindow(string s, string t) {
    unordered_map<char, int> need, window;
    for (char c : t) need[c]++;

    int left = 0, right = 0;
    int valid = 0;
    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
        ...

        /*** debug 输出的位置 ***/
        printf("window: [%d, %d)\n", left, right);
        /********************/

        // 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[left];
            // 左移窗口
            left++;
            // 进行窗口内数据的一系列更新
            ...
        }
    }
}
```

需要变化的地方

- 1、右指针右移之后窗口数据更新
- 2、判断窗口是否要收缩
- 3、左指针右移之后窗口数据更新
- 4、根据题意计算结果

H2 示例

minimum-window-substring

> 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

```go
func minWindow(s string, t string) string {
    // 保存滑动窗口字符集
    win := make(map[byte]int)
    // 保存需要的字符集
    need := make(map[byte]int)
    for i := 0; i < len(t); i++ {
        need[t[i]]++
    }
    // 窗口
    left := 0
    right := 0
    // match匹配次数
    match := 0
    start := 0
    end := 0
    min := math.MaxInt64
    var c byte
    for right < len(s) {
        c = s[right]
        right++
        // 在需要的字符集里面，添加到窗口字符集里面
        if need[c] != 0 {
            win[c]++
            // 如果当前字符的数量匹配需要的字符的数量，则match值+1
            if win[c] == need[c] {
                match++
            }
        }

        // 当所有字符数量都匹配之后，开始缩紧窗口
        for match == len(need) {
            // 获取结果
            if right-left < min {
                min = right - left
                start = left
                end = right
            }
            c = s[left]
            left++
            // 左指针指向不在需要字符集则直接跳过
            if need[c] != 0 {
```

```
                    // 左指针指向字符数量和需要的字符相等时，右移之后match值就不匹配则减一
                    // 因为win里面的字符数可能比较多，如有10个A，但需要的字符数量可能为3
                    // 所以在压死骆驼的最后一根稻草时，match才减一，这时候才跳出循环
                    if win[c] == need[c] {
                        match--
                    }
                    win[c]--
                }
            }
        }
        if min == math.MaxInt64 {
            return ""
        }
        return s[start:end]
    }
```

permutation-in-string

> 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。

```
func checkInclusion(s1 string, s2 string) bool {
    win := make(map[byte]int)
    need := make(map[byte]int)
    for i := 0; i < len(s1); i++ {
        need[s1[i]]++
    }
    left := 0
    right := 0
    match := 0
    for right < len(s2) {
        c := s2[right]
        right++
        if need[c] != 0 {
            win[c]++
            if win[c] == need[c] {
                match++
            }
        }
        // 当窗口长度大于字符串长度，缩紧窗口
        for right-left >= len(s1) {
            // 当窗口长度和字符串匹配，并且里面每个字符数量也匹配时，满足条件
            if match == len(need) {
                return true
            }
```

```
            d := s2[left]
            left++
            if need[d] != 0 {
                if win[d] == need[d] {
                    match--
                }
                win[d]--
            }
        }
    }
    return false
}

```

find-all-anagrams-in-a-string

> 给定一个字符串 **s** 和一个非空字符串 **p**，找到 **s** 中所有是 **p** 的字母异位词的子串，返回这些子串的起始索引。

```
func findAnagrams(s string, p string) []int {
    win := make(map[byte]int)
    need := make(map[byte]int)
    for i := 0; i < len(p); i++ {
        need[p[i]]++
    }
    left := 0
    right := 0
    match := 0
    ans:=make([]int,0)
    for right < len(s) {
        c := s[right]
        right++
        if need[c] != 0 {
            win[c]++
            if win[c] == need[c] {
                match++
            }
        }
        // 当窗口长度大于字符串长度，缩紧窗口
        for right-left >= len(p) {
            // 当窗口长度和字符串匹配，并且里面每个字符数量也匹配时，满足条件
            if right-left == len(p)&& match == len(need) {
                ans=append(ans,left)
            }
            d := s[left]
            left++
            if need[d] != 0 {
                if win[d] == need[d] {
```

```
                    match--
                }
                win[d]--
            }
        }
    }
    return ans
}
```

longest-substring-without-repeating-characters

> 给定一个字符串，请你找出其中不含有重复字符的 最长子串 的长度。
> 示例 1:
>
> 输入: "abcabcbb"
> 输出: 3
> 解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。

```
func lengthOfLongestSubstring(s string) int {
    // 滑动窗口核心点：1、右指针右移 2、根据题意收缩窗口 3、左指针右移更新窗口 4、根据题意计算结果
    if len(s)==0{
        return 0
    }
    win:=make(map[byte]int)
    left:=0
    right:=0
    ans:=1
    for right<len(s){
        c:=s[right]
        right++
        win[c]++
        // 缩小窗口
        for win[c]>1{
            d:=s[left]
            left++
            win[d]--
        }
        // 计算结果
        ans=max(right-left,ans)
    }
    return ans
}
func max(a,b int)int{
    if a>b{
        return a
    }
```

```
        return b
    }
```

总结

- 和双指针题目类似，更像双指针的升级版，滑动窗口核心点是维护一个窗口集，根据窗口集来进行处理
- 核心步骤
    - right 右移
    - 收缩
    - left 右移
    - 求结果

练习

- [ ] minimum-window-substring
- [ ] permutation-in-string
- [ ] find-all-anagrams-in-a-string
- [ ] longest-substring-without-repeating-characters