算法期末理论报告：双指针

双指针是一种简单灵活的算法，一般用在数组，单链表等数据结构中，可以极大程度降低时间复杂度。

目前常见的双指针算法分为三类：快慢指针、对撞指针、滑动窗口

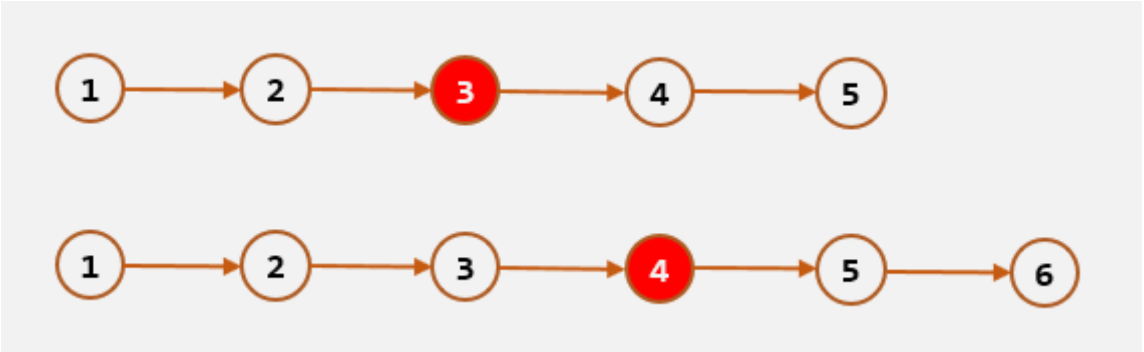
快慢指针

两个指针，初始在同一位置，然后向相同方向移动，一个移动速度快，一个移动速度慢。

适用场景：查找链表中间节点、链表是否包含环、原地修改数组。

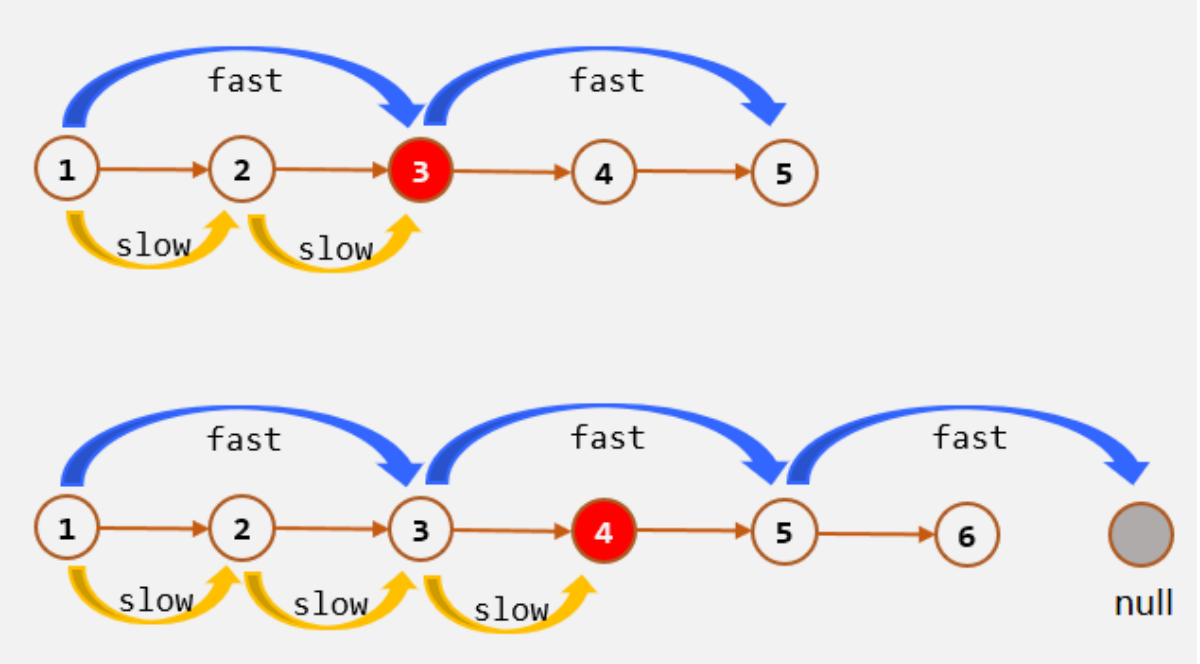
1.查找链表中间节点

给定单链表的头结点 head ，请找出并返回链表的中间结点。如果有两个中间结点，则返回第二个中间结点。



普通方法：通过遍历 2 次链表来找到中间节点，第 1 次遍历获取链表长度 LEN，第 2 次遍历到 LEN/2 的位置。

双指针方法：建立快慢指针，快指针一次移动两步，慢指针一次移动一步，快指针走到尽头时慢指针恰好到达中间结点，使用快慢指针只需要遍历一次就可以解决，明显运行效率可以得到提升。

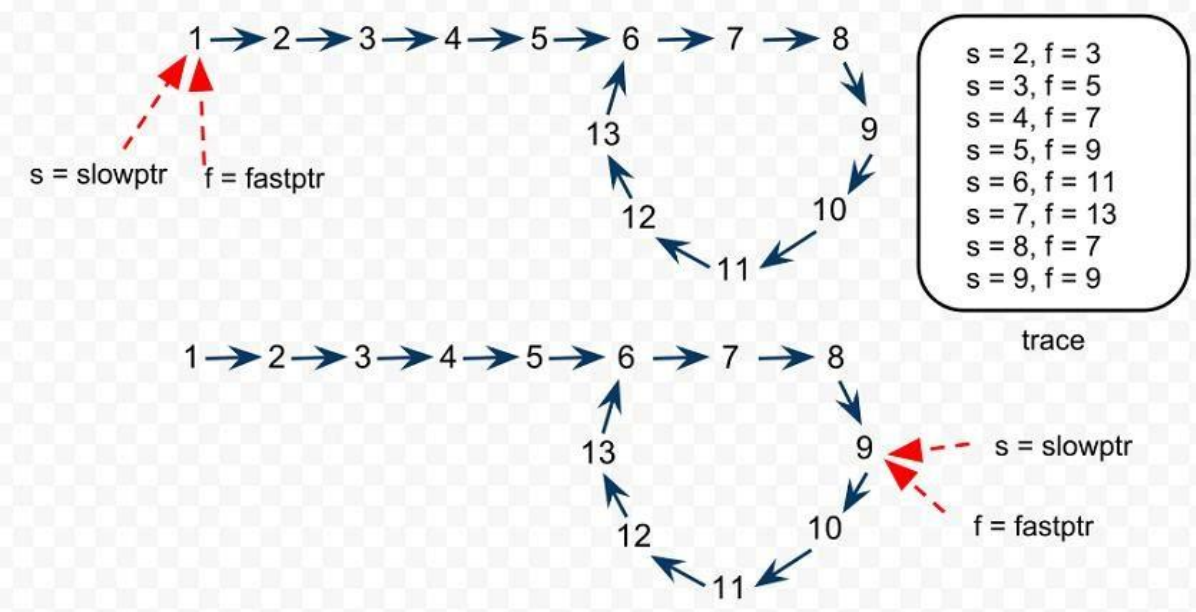


```
//中间节点
ListNode middleNode(ListNode head)
{
    ListNode fast = head;
    ListNode slow = head;
    while (fast != null && fast.next != null)
    {
        fast = fast.next.next;
        slow = slow.next;
    }
    return slow;
}
```

2.判断链表是否有环

给定链表的头结点 head，判断链表中是否有环

如果链表中存在环，则在链表上不断前进的指针会一直在环里绕圈子，且不能知道链表是否有环。但如果使用快慢指针，当链表中存在环时，两个指针最终会在环中相遇。



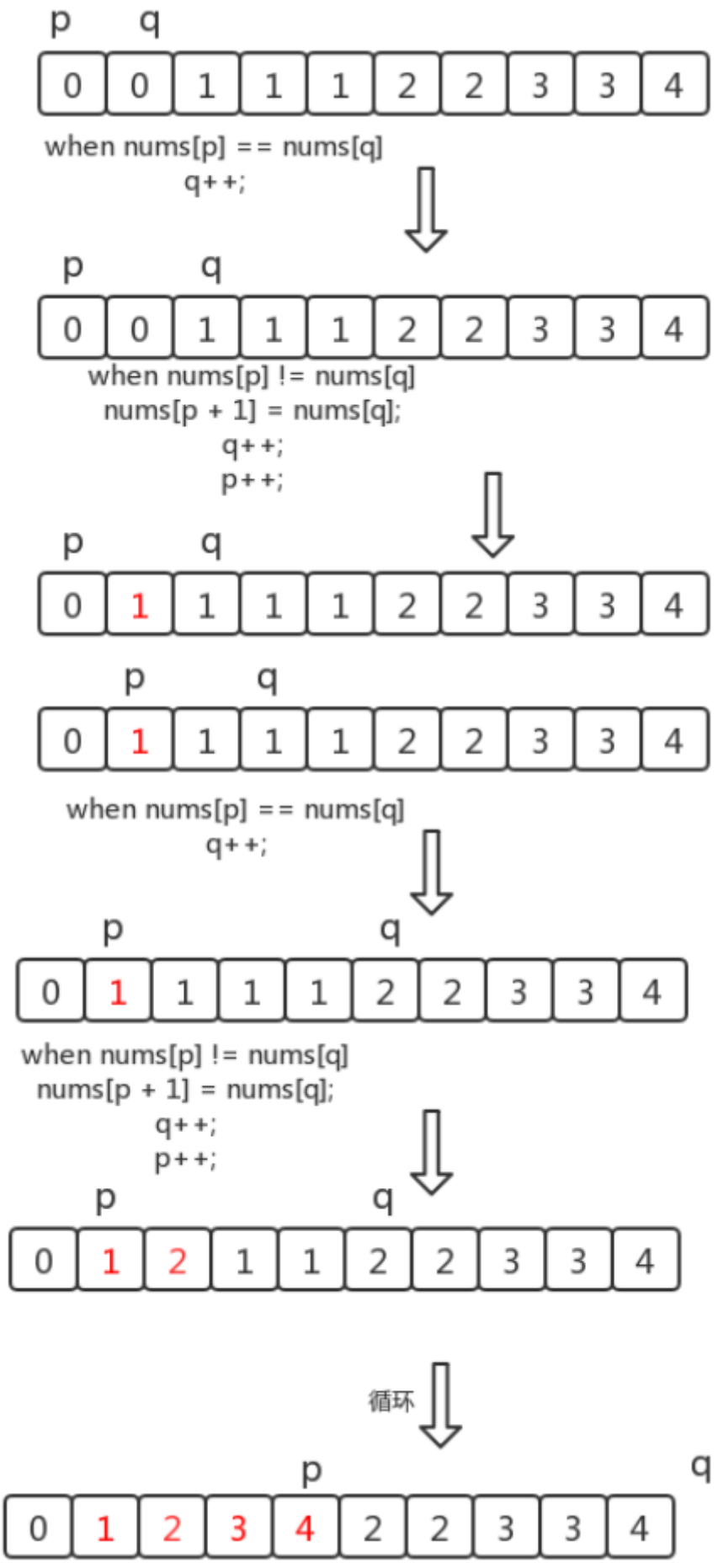

```
//判断链表是否有环
bool hasCycle(ListNode *head) {
        if(head==null||head->next==null)return false;
        ListNode * slow = head;
        ListNode * fast = slow->next;
        while(fast!=slow)
        {
            if(fast==null||fast->next==null)return false;
            slow = slow->next;
            fast = fast->next->next;
        }
        return true;
    }
```

3.删除有序数组中的重复项

给定一个有序数组 nums，请原地删除重复出现的元素，使每个元素只出现一次 ，返回删除后数组的新长度。元素的相对顺序应该保持一致 。然后返回 nums 中唯一元素的个数。

如果不是原地修改的话，可以用一个新数组保存去重之后的元素，返回新数组长度即可。但是原地修改，只能在原始数组上操作，所以，需要考虑使用快慢指针。

让慢指针 slow 走在后面，快指针 fast 走在前面，fast找到一个不重复的元素就赋值给 slow 并让 slow 前进一步。这样，就保证了 nums[0..slow] 都是不重复的元素，当 fast 指针遍历完整个数组 nums 后，nums[0..slow] 就是整个数组去重之后的结果

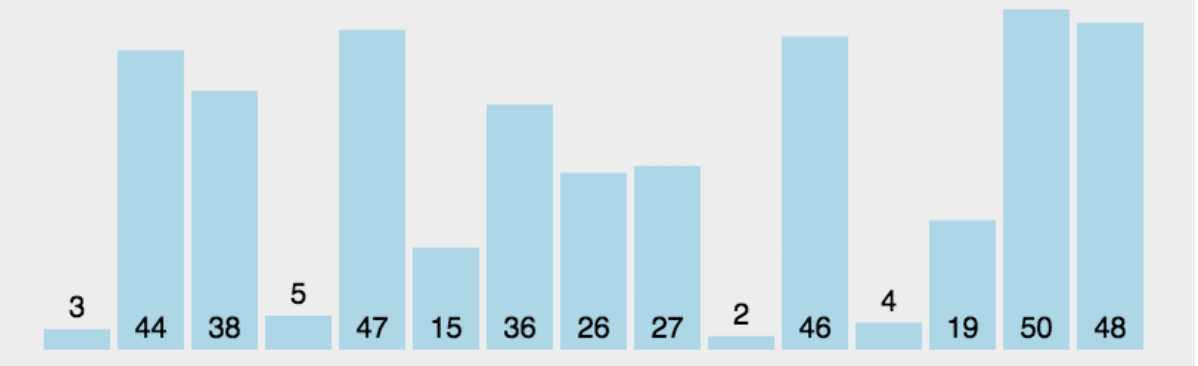



```
//删除有序数组中的重复项
int removeDuplicates(int[] nums) {
    int fast = 1;
```

```
    int slow = 0;
    int len = nums.length;
    while (fast < len) {
        // 出现不重复元素
        if (nums[fast] != nums[slow]) {
            slow++;
            // nums[0...slow] 保存不重复元素
            nums[slow] = nums[fast];
        }
        fast++;
    }
    return slow + 1;
}
```

4.选择排序

应用双指针，begin指针指向首，end指针指向尾 使用for循环在begin——end范围内寻找并记录最大的数下标maxi和最小数字的下标mini，想将数字进行升序排列，只需将begin——end范围内的最小数下标与begin交换即可。




```
#include<stdio.h>

void Swap(int* p1, int* p2)
{
    int tmp = *p2;
    *p2 = *p1;
    *p1 = tmp;
}
void SelectSort(int* a, int n)
{
    int begin = 0;
    int end = n - 1;
    while (begin < end)
    {
        int mini = begin;
        int maxi = begin;
        for (int i = begin; i <= end; i++)
        {
            //寻找最小的数字的下标
            if (a[i] < a[mini])
            {
                mini = i;
            }
            //寻找最大数字的下标
```

```
            if (a[i] > a[maxi])
            {
                maxi = i;
            }
        }
        Swap(&a[begin],&a[mini]);//最小的放到第一位
        if (begin == maxi)
        {
            maxi = mini;
        }
        Swap(&a[end],&a[maxi]);//最大的放到最后一位
        ++begin;
        --end;
    }
}
void Printarray(int* a, int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("%d ", a[i]);
    }
}
void TestSelectSort()
{
    int a[] = { 3,5,4,2,1 };
    int n = sizeof(a) / sizeof(a[0]);
    SelectSort(a, n);
    Printarray(a, n);
}
int main()
{
    TestSelectSort();
    return 0;
}
```

对撞指针

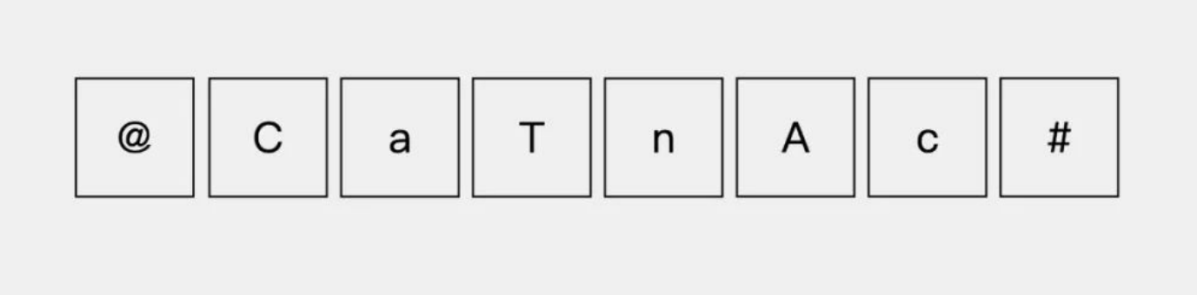
两个指针，初始一个在左、一个在右，左指针向右移动，右指针向左移动，直到相撞停止。

适用场景：二分查找、反转数组、回文判定。

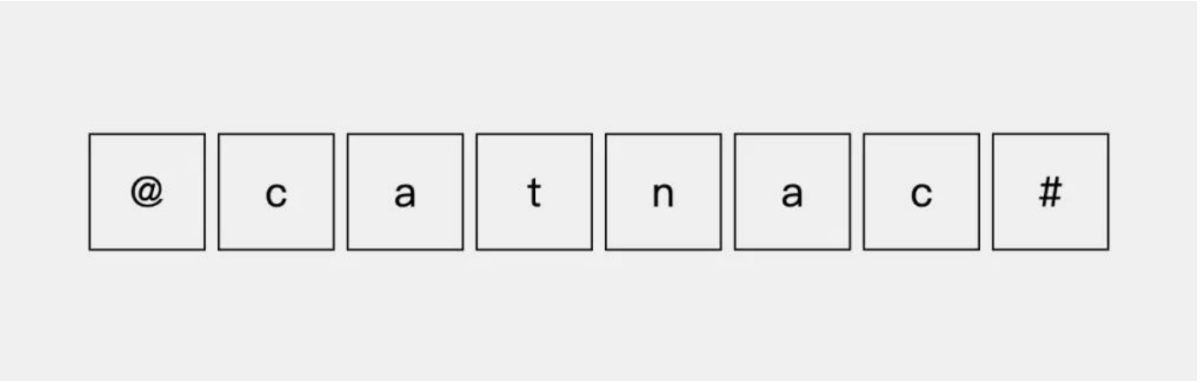
1.验证回文串

给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，忽略字母的大小写。在本题中，将空字符串定义为有效的回文串。

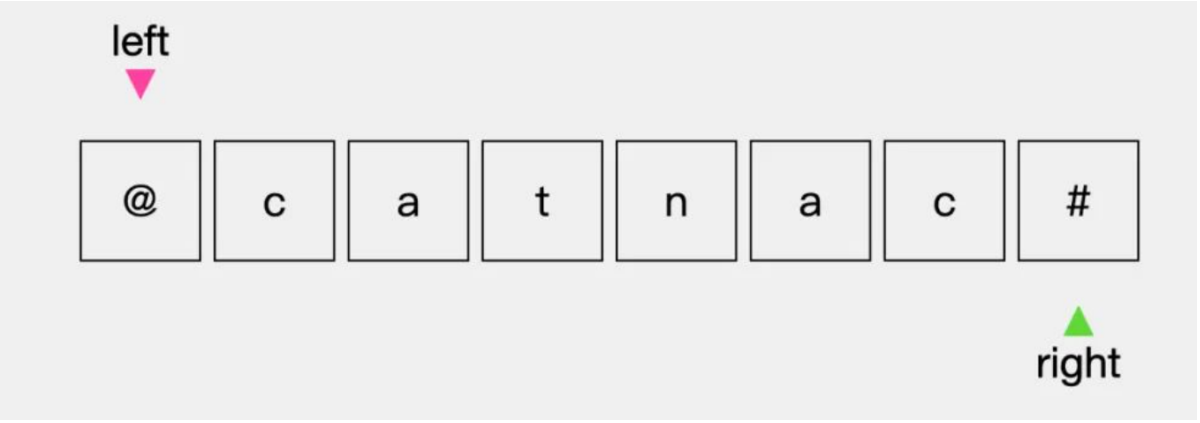
我们以字符串"@CaTnAc#"为例来看一下如何用对撞指针的方法判断一个字符串是否是回文串



因为题目描述中是忽略字符串大小写的，因此先将字符串中所有字符转为小写字母。

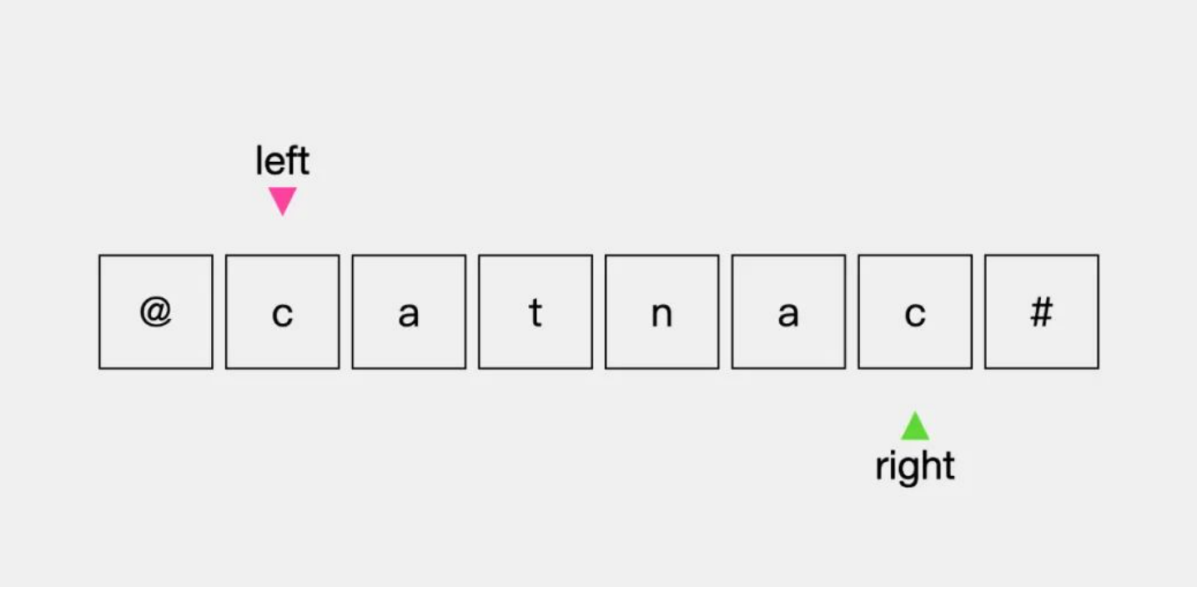


然后建立左指针left指向字符串左边第一个元素，右指针right指向字符串右边第一个元素。

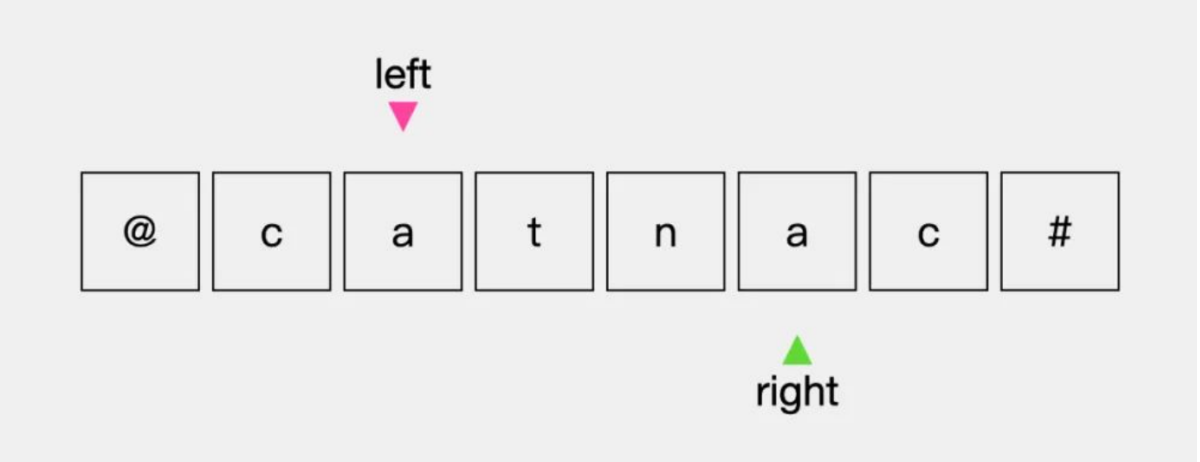


先看左指针left，当前指向的元素是”@“字符，不是字母也不是数字。因此，left需要向右移动一位。同理，指针right也应向左移动一位。

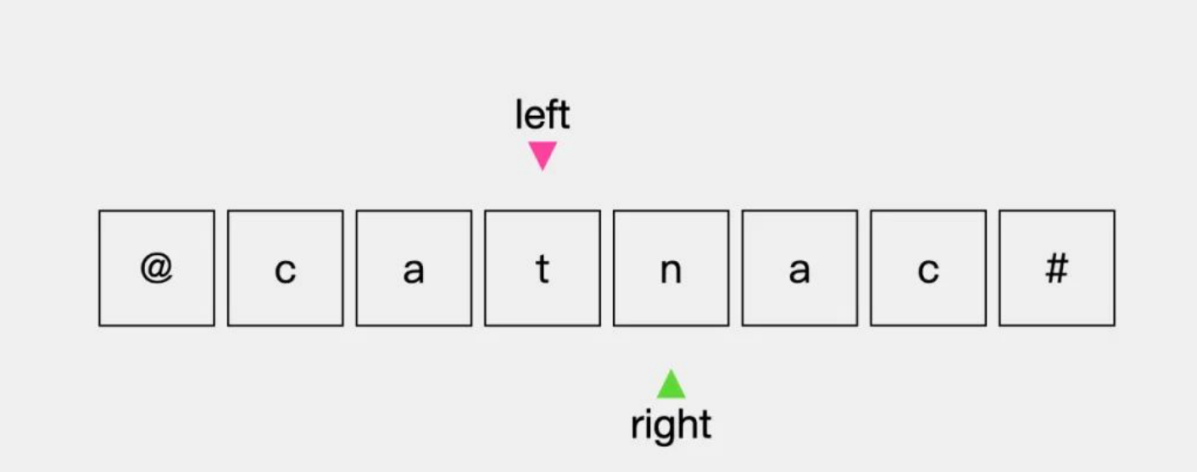
这时指针left指向的字符”c“与指针right指向的字符”c“是一样的。



因此指针left向右继续移动一位，指针right向左继续移动一位，这时指针left指向的字符"a"与指针right指向的字符"a"是一样的



因此指针left向右继续移动一位，指针right向左继续移动一位，这时，指针left指向的字符"t"与right指向的字符"n"是不同的，也就是说字符串"@CaTnAc#"不是回文串。至此，即使有剩余的字符也就不需要考虑了。




```
 boolean isPalindrome(String s) {
    String lowerCase = s.toLowerCase();
    int left = 0;
    int right = lowerCase.length() - 1;
    while (left < right) {
        // 指针left小于指针right且当前考察的字符不是字母或数字，指针left向左移动
        while (left < right &&
!Character.isLetterOrDigit(lowerCase.charAt(left))) {
            left++;
        }
        // 指针left小于指针right且当前考察的字符不是字母或数字，指针right向右移动
        while (left < right &&
!Character.isLetterOrDigit(lowerCase.charAt(right))) {
            right--;
        }
        // 如果指针left指向的字符与指针right指向的字符不同，则不是回文串
        if (lowerCase.charAt(left) != lowerCase.charAt(right)) {
            return false;
        }
        // 指针left左移，指针right右移，继续考察下一对字符
        left++;
        right--;
```

```
    }
    return true;
}
```

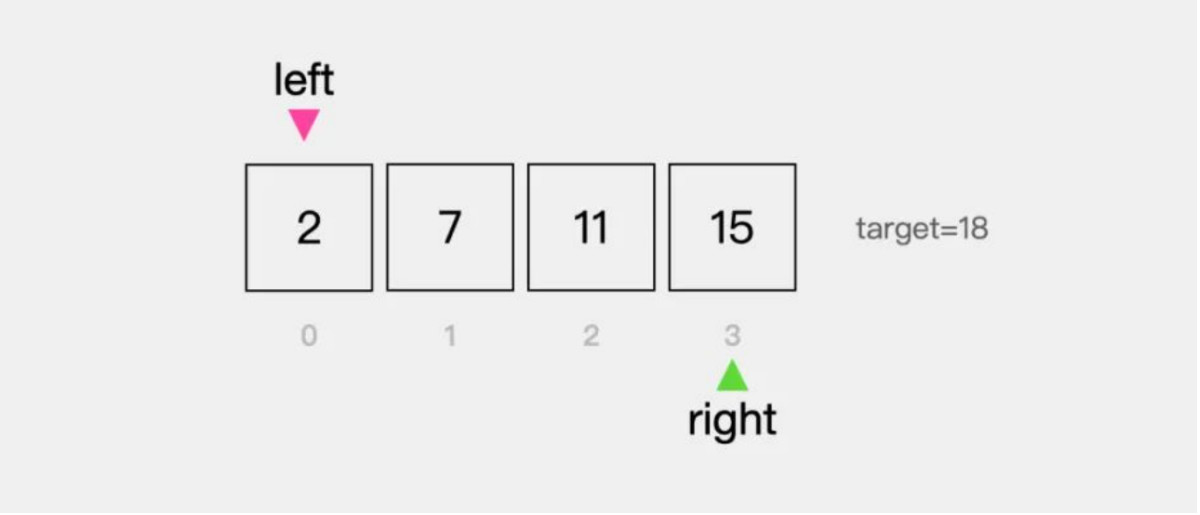
2.两数之和

给定一个已按照升序排列的有序数组，找到两个数使得它们相加之和等于目标数。函数应该返回这两个下标值 index1 和 index2，其中 index1 必须小于 index2。

普通方法：用暴力解法来解决，使用双重for循环，第一重for循环每次选取一个数，第二重for循环每次从剩余的数中选取一个数，然后计算两数之和，将其值与目标值比较。

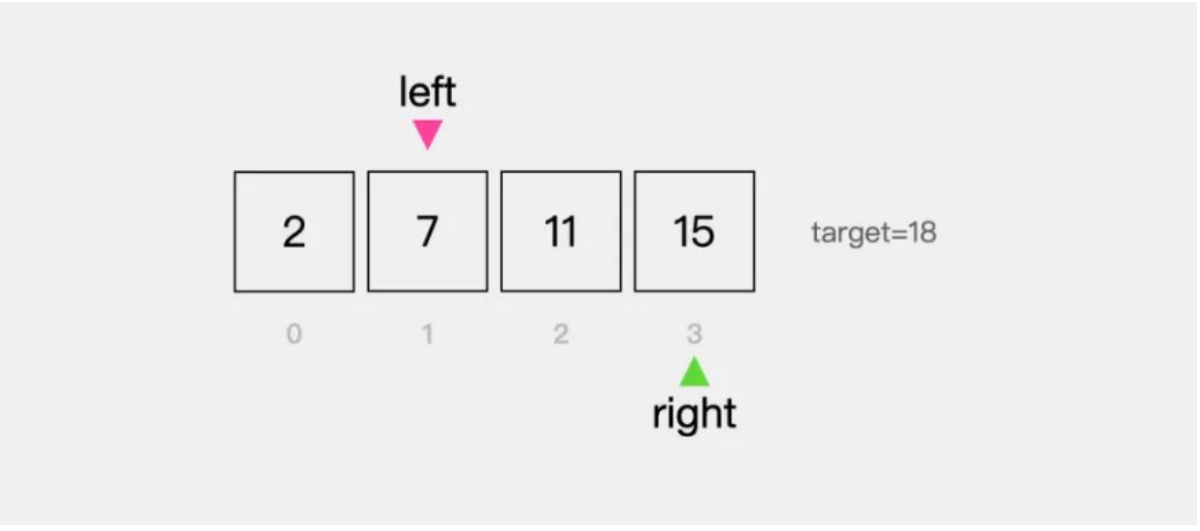
双指针方法：

首先，定义左侧指针left指向数组中第一个元素，右侧指针right指向数组中最后一个元素

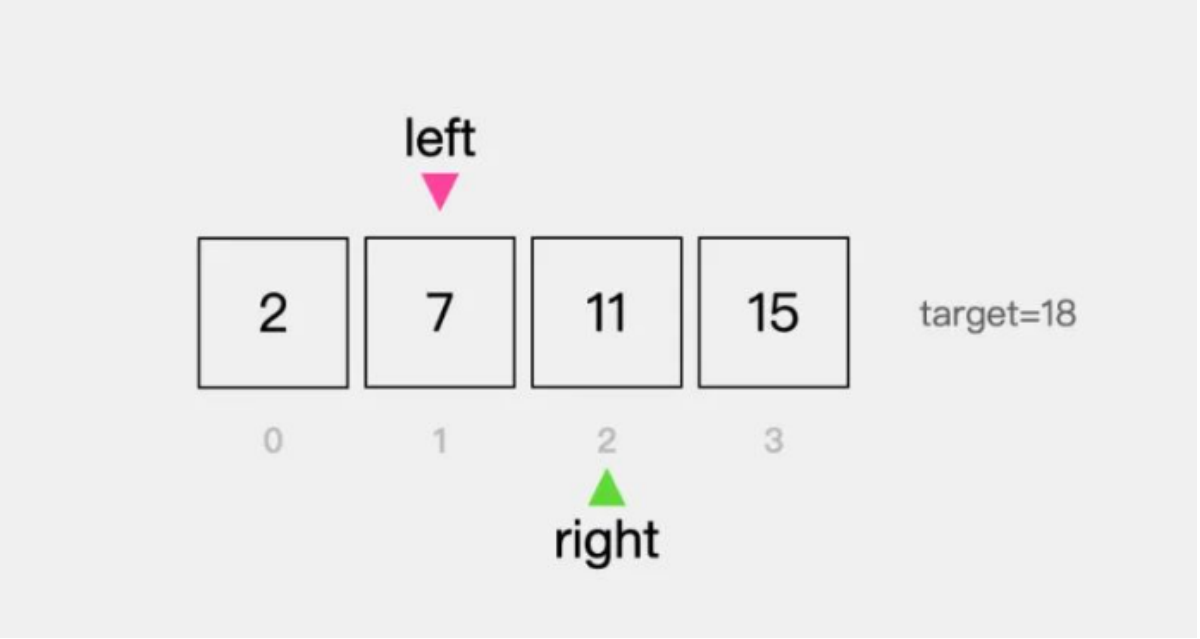


因为数组是升序排列的，为了让两个数的和变大一些，应将左侧指针left向右移动一位

此时，numbers[left]+numbers[right]=22大于target=18



由于数组是升序排列的，为了让两个数的和变小一些，应将右侧指针向左移动一位



此时，numbers[left]+numbers[right]=18等于target=18。因此，找到了两个数7和11，其和等于目标值

滑动窗口

滑动窗口通常定义 left 和 right 两个指针，两指针同向移动，指针需要通过判断区间 [left , right] 是否合法来决定指针的走向；因为在指针移动过程中，区间 [left , right] 很像一个滑动的窗口，故称为滑动窗口算法

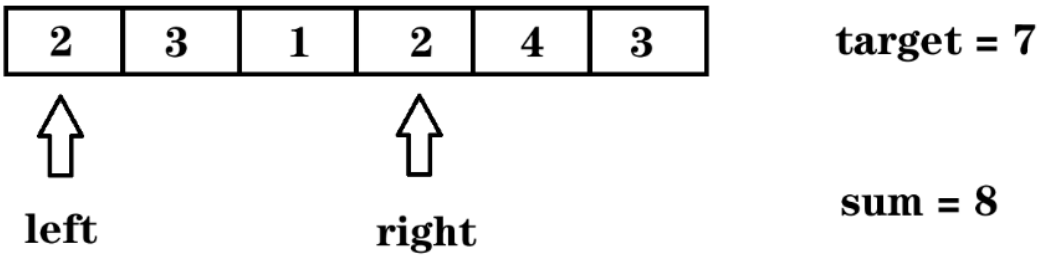
1.长度最小的子数组

给定一个含有 n 个正整数的数组和一个正整数 target 。找出该数组中满足其总和大于等于 target 的长度最小的子数组[numsl, numsl+1, ..., numsr-1, numsr] ，并返回其长度。如果不存在符合条件的子数组，返回 0 。

普通方法：找到第一个合法数组，先记录长度，再让 left 向后移一位，right 回到 left 的位置向后穷举，时间复杂度O(N^2)

双指针方法：

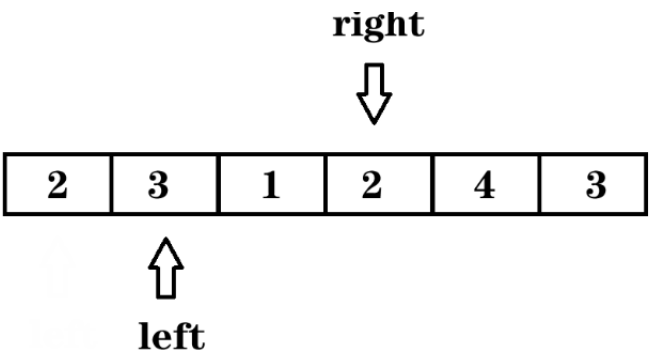
先建立滑动窗口，即找出第一个满足题目要求的合法窗口



接下来

如果窗口合法就尝试压缩窗口，即left右移

如果窗口不合法就尝试增大窗口，即right右移



ret = 4

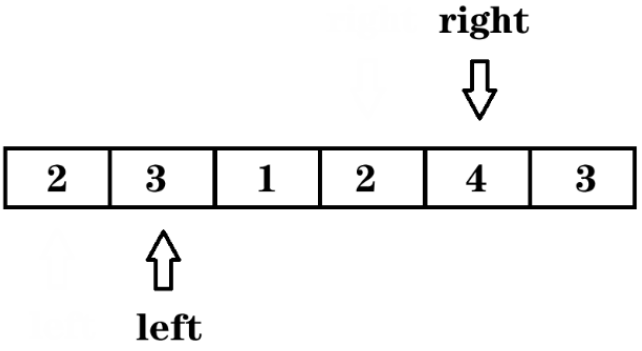
窗口合法性：0

1 为合法

0 为非法

sum = 6

target = 7



ret = 4

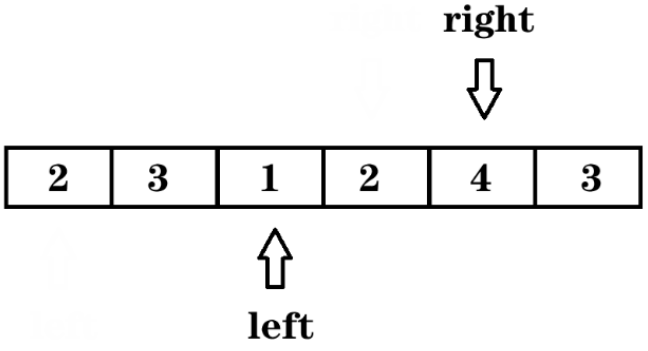
窗口合法性： 1

1 为合法

0 为非法

sum = 10

target = 7



ret = 3

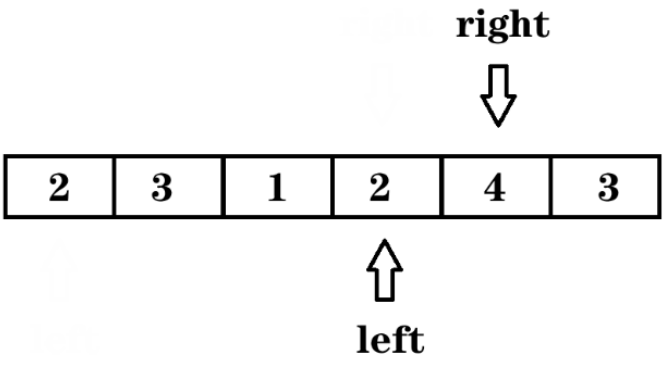
窗口合法性： 1

1 为合法

0 为非法

sum = 7

target = 7



ret = 3

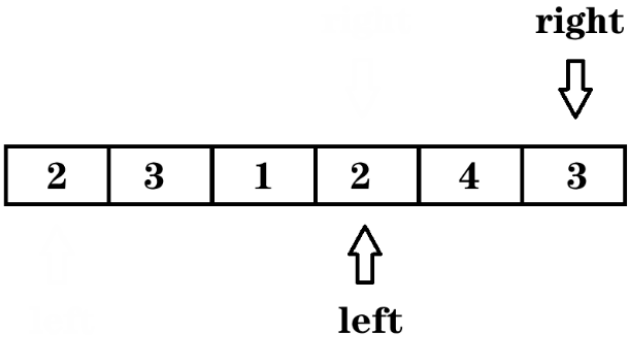
窗口合法性： 0

1 为合法

0 为非法

sum = 6

target = 7



ret = 3

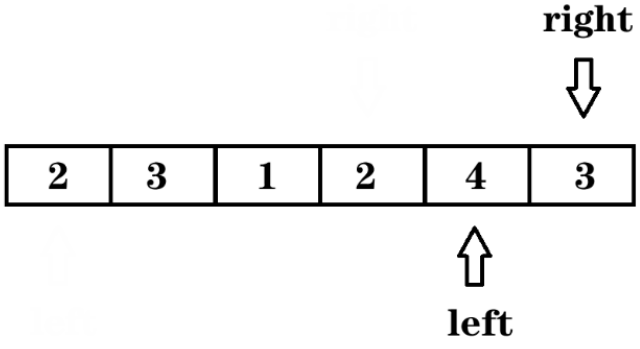
窗口合法性： 1

1 为合法

0 为非法

sum = 9

target = 7



ret = 2

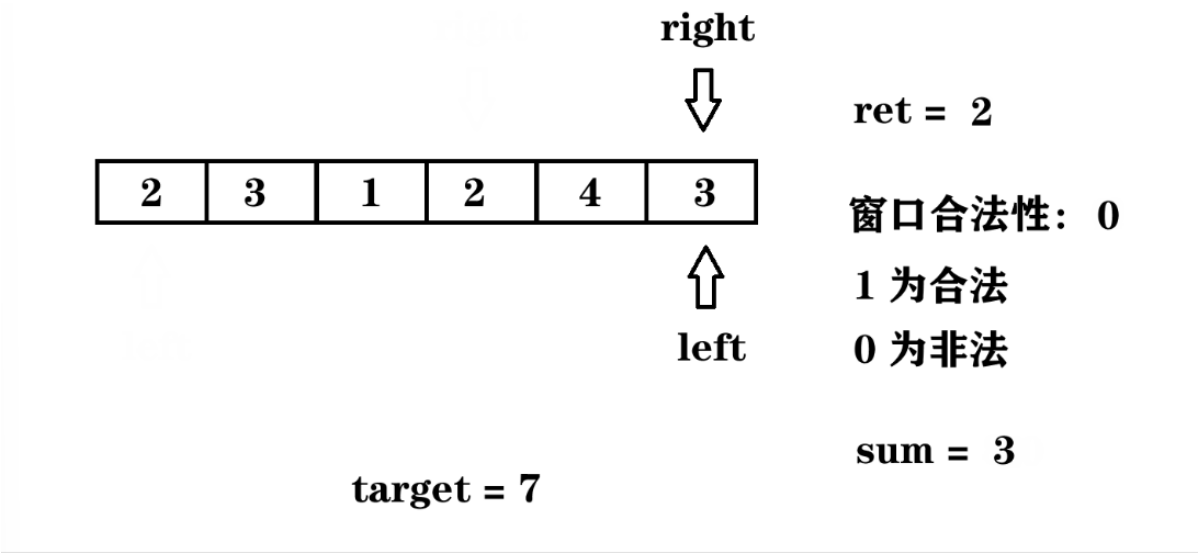
窗口合法性： 1

1 为合法

0 为非法

sum = 7

target = 7



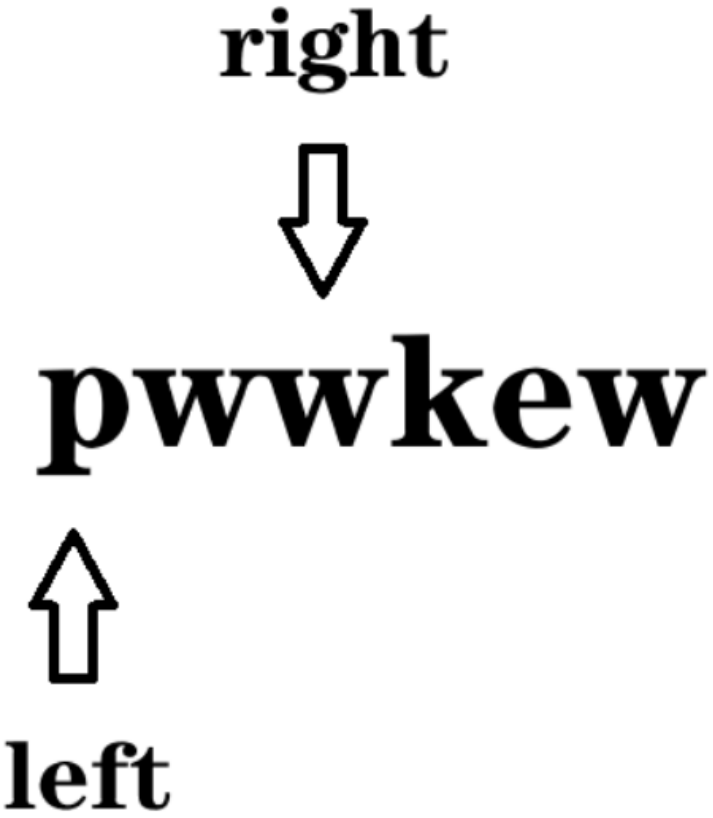
最终可以找到满足条件的最小窗口为[4,3]，此方法时间复杂度为O(N)

```
int minSubArrayLen(int target, vector<int>& nums)
    {
        size_t left = 0,right = 0,sz = nums.size();
        size_t ret = -1,sum = 0;    // 用 sum 维护窗口
        while(right < sz)
        {
            // 更新 sum 维护窗口
            sum += nums[right];
            // 判断窗口的合法性
            while(sum >= target)
            {
                // 窗口合法，更新结果
                ret = min(ret,right - left + 1);
                // 出窗口
                sum -= nums[left++];
            }
            // 窗口不合法，进窗口
            right++;
        }
        // 特殊情况的处理，未找到结果
        if(ret == -1)
            return 0;
        else
            return ret;
    }
```

2.无重复字符的最长子串

给定一个字符串s，找出其中不含有重复字符的最长子串的长度

普通方法：找到第一个不合法子串，让 left 向后移一位，right 回到 left 的位置向后穷举

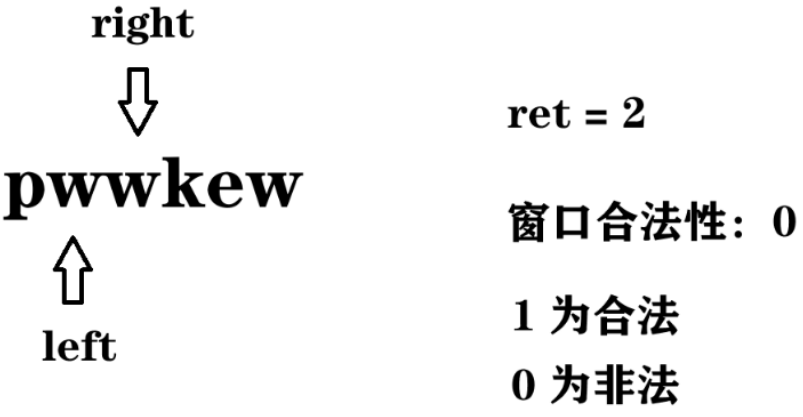


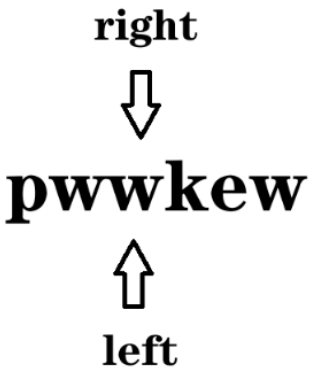
双指针方法：

找到第一个不合法窗口后

如果窗口合法就尝试增大窗口，即right右移

如果窗口不合法就尝试缩小窗口，即left右移



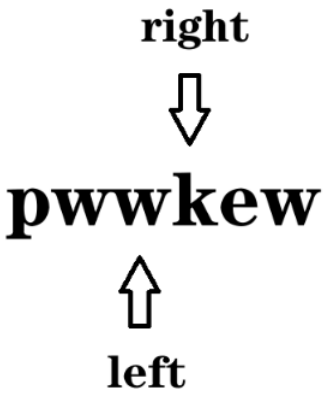


ret = 2

窗口合法性：1

1 为合法

0 为非法

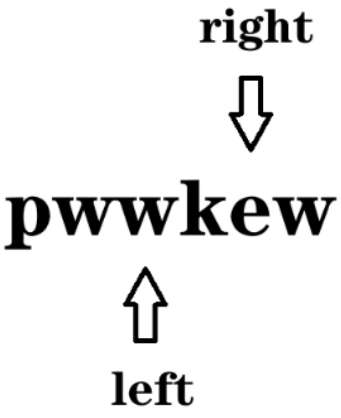


ret = 2

窗口合法性：1

1 为合法

0 为非法

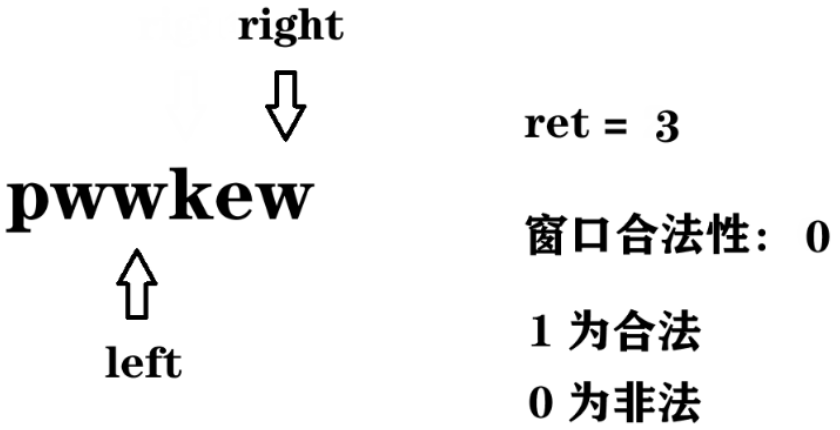


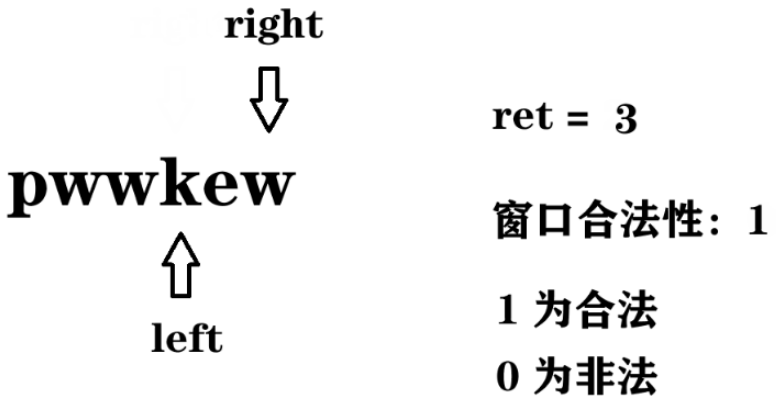
ret = 3

窗口合法性：1

1 为合法

0 为非法





最后可以找到[w,k,e]和[k,e,w]两个最大窗口，长度为3，时间复杂度为O(N)

```
int lengthOfLongestSubstring(string s)
    {
        int hash[128] = {0};    // 哈希表统计窗口内每个字符出现的频次，用来维护窗口
        size_t left = 0, right = 0, sz = s.size();
        size_t ret = 0;
        while (right < sz)
        {
            hash[s[right]]++;       // 把 right 指向字符丢进哈希表
            while (hash[s[right]] > 1)      // 若丢进哈希表的字符频次大于 1 ，子串出现重复字符，窗口不合法
                hash[s[left++]]--;          // left++ 出窗口，并把出窗口的字符在哈希表中的频次减一
            ret = max(ret, right - left + 1);       // 更新结果
            right++;            // right++进窗口
        }
        return ret;
    }
```

总结

双指针将一个两层循环转化成了一层循环，时间复杂度也从n^2变成了n，那么什么时候会需要使用双指针呢?

一般来讲，当遇到需要对一个数组进行重复遍历时，可以想到使用双指针法，可以帮助我们降低时间复杂度，提高程序运行效率