

# 刷题yyds

author: Zilin Xu

date: 2023.05.25

reference: Labuladong, Leetcode, ChatGPT etc.

This file records the questions of leetcode questions. The content is divided types of datastructures and algorithms.

## 目录


- 刷题yyds
  - 目录
  - 第一章，基础数据结构
    - 二分搜索
      - 34. 在排序数组中查找元素的第一个和最后一个位置
      - 704. 二分搜索
      - 剑指 Offer 53 - I. 在排序数组中查找数字 I
      - 35. 搜索插入位置
      - 240. 搜索二维矩阵 II
      - 74. 搜索二维矩阵
      - 392-判断子序列
      - 658.找到 K 个最接近的元素
      - 852. 山脉数组的峰顶索引
      - 剑指 Offer 53 - II. 0~n-1中缺失的数字
    - 滑动窗口
      - 3. 无重复字符的最长子串
      - 438. 找到字符串中所有字母异位词
      - 567. 字符串的排列
      - 76. 最小覆盖子串
      - 239. 滑动窗口最大值
    - 其他题目
      - 26. 删除有序数组中的重复项
      - 27. 移除元素
      - 283. 移动零
      - 83. 删除排序链表中的重复元素
      - 剑指 Offer 21. 调整数组顺序使奇数位于偶数前面
      - 剑指 Offer 57. 和为s的两个数字
      - 82. 删除排序链表中的重复元素 II

## 第一章，基础数据结构

### 二分搜索

<https://labuladong.github.io/algorithm/di-ling-zh-bfe1b/wo-xie-le--3c789/>

#### 34. 在排序数组中查找元素的第一个和最后一个位置

 截屏2023-05-25 22.34.48.png

```
class Solution {
    public int[] searchRange(int[] nums, int target) {
        return new int[]
{left_bound(nums,target),right_bound(nums,target)};
    }
    //use binary search to find left and right bound of result
    int left_bound(int[] nums, int target){
        int left = 0, right = nums.length-1;
        while(left <= right){
            int mid = left + (right - left) / 2;
            //which means target is in right half area of array
            if(target > nums[mid]){
                left = mid + 1;
            }
            else if(target < nums[mid]){
                right = mid - 1;
            }
            //shrink right window to find most left index
            //如果说, 找到了target, 但我们的while loop还没结束的话, 说明这不是最左边
            //那么我们就要继续收缩右窗口
            else if(target == nums[mid]){
                right = mid - 1;
            }
        }
        //check if the left is valid
        if(left >= nums.length || nums[left] != target){
            return -1;
        }
        return left;
    }
    //same as above
    int right_bound(int[] nums, int target){
        int left = 0, right = nums.length - 1;
        while(left <= right){
            int mid = left + (right - left)/2;
            if(nums[mid] == target){
                left = mid + 1;
            }
            else if (nums[mid] > target){
                right = mid - 1;
            }
            else if(nums[mid] < target){
                left = mid + 1;
            }
        }
        if (right < 0 || nums[right] != target){
            return -1;
        }
        return right;
    }
}
```

```
}  
}
```

这道题目的要求是在一个排序数组中找到目标值的左右边界。由于数组是排序的，我们可以利用二分查找的思想来解决这个问题。

二分查找算法可以在排序数组中高效地查找目标值，时间复杂度为  $O(\log n)$ ，其中  $n$  是数组的长度。这种算法通过反复将查找范围缩小为一半来快速定位目标值。

在这段代码中，我们使用了两次二分查找来找到目标值的左右边界。具体思路如下：

#### 1. 左边界的查找：

- 使用二分查找方法找到目标值的位置。
- 如果找到目标值，我们继续向左收缩右边界，直到找到最左边的目标值。
- 如果最终左边界超出了数组范围或者左边界对应的值不等于目标值，说明数组中不存在目标值，返回 -1。


#### 2. 右边界的查找：

- 使用二分查找方法找到目标值的位置。
- 如果找到目标值，我们继续向右收缩左边界，直到找到最右边的目标值。
- 如果最终右边界越界（小于 0）或者右边界对应的值不等于目标值，说明数组中不存在目标值，返回 -1。

通过这种方法，我们可以找到目标值在排序数组中的左右边界。由于二分查找的时间复杂度较低，这种解决方案具有较好的效率。

该解决方案的**时间复杂度**是  $O(\log n)$ ，其中  $n$  是数组的长度。这是因为解决方案使用了两次二分查找来找到目标值的左右边界。每次二分查找都会将搜索范围缩小一半，因此在最坏情况下，每次二分查找都需要遍历数组的一半，直到找到目标值或者搜索范围为空。所以每次二分查找的时间复杂度是  $O(\log n)$ 。整体的解决方案包含两次二分查找，因此时间复杂度是  $O(\log n) + O(\log n) = O(\log n)$ 。对于**空间复杂度**，该解决方案只使用了常数级别的额外空间，因此空间复杂度是  $O(1)$ 。这是因为解决方案并没有使用任何与输入数组大小相关的额外空间。它只使用了一些变量来存储搜索范围的左右边界、中间位置等信息，这些变量的数量是固定的，与输入数组的大小无关。因此，空间复杂度是  $O(1)$ 。

## 704. 二分搜索

 截屏2023-05-26 09.08.55.png

```
class Solution {  
    public int search(int[] nums, int target) {  
        int left = 0, right = nums.length - 1;  
        while(left <= right){  
            int mid = left + (right - left)/2;  
            if(nums[mid] == target){  
                return mid;  
            }  
            else if (nums[mid] < target){
```

```
        left = mid + 1;
    }
    else if(nums[mid] > target){
        right = mid - 1;
    }
}
return -1;
}
```

## 1、为什么 while 循环的条件中是 `<=`，而不是 `<`？

答：因为初始化 `right` 的赋值是 `nums.length - 1`，即最后一个元素的索引，而不是 `nums.length`。

这二者可能出现在不同功能的二分查找中，区别是：前者相当于两端都闭区间 `[left, right]`，后者相当于左闭右开区间 `[left, right)`。因为索引大小为 `nums.length` 是越界的，所以我们将 `right` 这一边视为开区间。

我们这个算法中使用的是前者 `[left, right]` 两端都闭的区间。这个区间其实就是每次进行搜索的区间。

什么时候应该停止搜索呢？当然，找到了目标值的时候可以终止：

```
if(nums[mid] == target)
    return mid;
```

但如果没找到，就需要 while 循环终止，然后返回 -1。那 while 循环什么时候应该终止？搜索区间为空的时候应该终止，意味着你没得找了，就等于没找到嘛。

`while(left <= right)` 的终止条件是 `left == right + 1`，写成区间的形式就是 `[right + 1, right]`，或者带个具体的数字进去 `[3, 2]`，可见这时候区间为空，因为没有数字既大于等于 3 又小于等于 2 的吧。所以这时候 while 循环终止是正确的，直接返回 -1 即可。

`while(left < right)` 的终止条件是 `left == right`，写成区间的形式就是 `[right, right]`，或者带个具体的数字进去 `[2, 2]`，这时候区间非空，还有一个数 2，但此时 while 循环终止了。也就是说区间 `[2, 2]` 被漏掉了，索引 2 没有被搜索，如果这时候直接返回 -1 就是错误的。

当然，如果你非要用 `while(left < right)` 也可以，我们已经知道了出错的原因，就打个补丁好了：

```
//...
while(left < right) {
    // ...
}
return nums[left] == target ? left : -1;
```

## 2、为什么 `left = mid + 1`，`right = mid - 1`？我看有的代码是 `right = mid` 或者 `left = mid`，没有这些加加减减，到底怎么回事，怎么判断？

答：这也是二分查找的一个难点，不过只要你能理解前面的内容，就能够很容易判断。

刚才明确了「搜索区间」这个概念，而且本算法的搜索区间是两端都闭的，即 `[left, right]`。那么当我们发现索引 `mid` 不是要找的 `target` 时，下一步应该去搜索哪里呢？

当然是去搜索区间 `[left, mid-1]` 或者区间 `[mid+1, right]` 对不对？因为 `mid` 已经搜索过，应该从搜索区间中去除。

### 3、此算法有什么缺陷？

答：至此，你应该已经掌握了该算法的所有细节，以及这样处理的原因。但是，这个算法存在局限性。

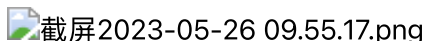
比如说给你有序数组 `nums = [1,2,2,2,3]`，`target` 为 2，此算法返回的索引是 2，没错。但是如果我想得到 `target` 的左侧边界，即索引 1，或者我想得到 `target` 的右侧边界，即索引 3，这样的话此算法是无法处理的。

这样的需求很常见，你也许会说，找到一个 `target`，然后向左或向右线性搜索不行吗？可以，但是不好，因为这样难以保证二分查找对数级的复杂度了。

我们后续的算法就来讨论这两种二分查找的算法。

TC:  $O(\log n)$

### 剑指 Offer 53 - I. 在排序数组中查找数字 I

 截屏2023-05-26 09.55.17.png

```
class Solution {
    public int search(int[] nums, int target) {
        //same with 34, find left bound and right bound
        //get left and right bound
        int left = left_bound(nums, target);
        //if left index is -1, means we didn't have target in the array
        if(left == -1){
            return 0;
        }
        int right = right_bound(nums, target);
        return right - left + 1;
    }

    int left_bound(int[] nums, int target){
        int left = 0, right = nums.length - 1;
        while(left <= right){
            int mid = left + (right - left)/2;
            if(nums[mid] == target){
                right = mid - 1;
            }
            else if (nums[mid] < target){
                left = mid + 1;
            }
            else if (nums[mid] > target){
                right = mid - 1;
            }
        }
    }
}
```


```
    }

    }
    if(left > nums.length - 1 || nums[left] != target){
        return -1;
    }
    return left;
}

int right_bound(int[] nums, int target){
    int left = 0, right = nums.length - 1;
    while(left <= right){
        int mid = left + (right - left)/2;
        if(nums[mid] == target){
            left = mid + 1;
        }
        else if(nums[mid] < target){
            left = mid + 1;
        }
        else if(nums[mid] > target){
            right = mid - 1;
        }
    }
    if(right < 0 || nums[right] != target){
        return -1;
    }
    return right;
}
}
```

和Leetcode 34 类似，我们先找到left bound 和 right bound，在主函数中我们要判断left bound是不是 -1。

### 35. 搜索插入位置

 截屏2023-05-26 10.22.17.png

```
class Solution {
    public int searchInsert(int[] nums, int target) {
        //找到left bound
        return left_bound(nums, target);
    }

    int left_bound(int[] nums, int target){
        int left = 0, right = nums.length - 1;
        while(left <= right){
            int mid = left + (right - left)/2;
            if (nums[mid] == target){
                right = mid - 1;
            }
            else if(nums[mid] < target){
```

```
        left = mid + 1;
    }
    else if(nums[mid] > target){
        right = mid - 1;
    }
}
//直接返回
return left;
}
}
```

### 基本思路

这道题就是考察搜索左侧边界的二分算法的细节理解，前文 二分搜索详解 着重讲了数组中存在目标元素重复的情况，没仔细讲目标元素不存在的情况。


当目标元素 `target` 不存在数组 `nums` 中时，搜索左侧边界的二分搜索的返回值可以做以下几种解读：

- 1、返回的这个值是 `nums` 中大于等于 `target` 的最小元素索引。
- 2、返回的这个值是 `target` 应该插入在 `nums` 中的索引位置。
- 3、返回的这个值是 `nums` 中小于 `target` 的元素个数。

比如在有序数组 `nums = [2,3,5,7]` 中搜索 `target = 4`，搜索左边界的二分算法会返回 2，你带入上面的说法，都是对的。

所以以上三种解读都是等价的，可以根据具体题目场景灵活运用，显然这里我们需要的是第二种。

## 240. 搜索二维矩阵 II

 截屏2023-05-26 17.05.08.png

```
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        //binary search, 从右上角开始
        int i = 0, j = matrix[0].length - 1;
        while(i < matrix.length && j >= 0 ){
            if(matrix[i][j] == target){
                return true;
            }
            if(matrix[i][j] < target){
                i ++;
            }
            else{
                j --;
            }
        }
        return false;
    }
}
```

```
}  
}
```

### 基本思路

作为 74. 搜索二维矩阵，更像 一个方法秒杀所有 N 数之和问题，因为它们的思想上有些类似。

这道题说 matrix 从上到下递增，从左到右递增，显然左上角是最小元素，右下角是最大元素。我们如果想高效在 matrix 中搜索一个元素，肯定需要从某个角开始，比如说从左上角开始，然后每次只能向右或向下移动，不要走回头路。

如果真从左上角开始的话，就会发现无论向右还是向下走，元素大小都会增加，那么到底向右还是向下？不确定，那只好用类似 动态规划算法 的思路穷举了。


但实际上不用这么麻烦，我们不要从左上角开始，而是从右上角开始，规定只能向左或向下移动。

你注意，如果向左移动，元素在减小，如果向下移动，元素在增大，这样的话我们就可以根据当前位置的元素和 target 的相对大小来判断应该往哪移动，不断接近从而找到 target 的位置。

时间复杂度：在最坏情况下，即当目标值位于矩阵的左下角时，算法的时间复杂度为 $O(m+n)$ ，其中 $m$ 为矩阵的行数， $n$ 为矩阵的列数。这是因为每次比较都会使得行索引增加1或列索引减少1，而最多进行 $m+n$ 次比较即可找到目标值或确定其不存在。

空间复杂度：该算法的空间复杂度为 $O(1)$ ，因为除了使用常量级别的变量 $i$ 、 $j$ 和输入的matrix外，没有使用额外的空间。

## 74. 搜索二维矩阵

 截屏2023-07-13 15.59.09.png

```
class Solution {  
    public boolean searchMatrix(int[][] matrix, int target) {  
        //把这个二维数组当作一位数组来看  
        int m = matrix.length;  
        int n = matrix[0].length;  
        int left = 0, right = m * n - 1;  
  
        //start loop  
        while(left <= right){  
            //这时候的mid意思是，matrix中第几个元素（从0开始数）  
            int mid = left + (right - left)/2;  
  
            //找到当左右边界为left, right的时候,对应到matrix里面元素的位置  
            int i = mid / n;  
            int j = mid % n;  
  
            //start finding  
            if(matrix[i][j] == target){  
                return true;  
            }  
        }  
    }  
}
```



```
        }else if(matrix[i][j] < target){
            left = mid + 1;
        }else {
            right = mid - 1;
        }

    }

    return false;
}
}
```


时间复杂度分析：

- 获取矩阵的行数和列数的操作的时间复杂度为  $O(1)$ ，因为它们只需要访问矩阵的常量个元素。
- 在二分搜索过程中，每次循环将搜索范围减半，因此最多需要进行  $O(\log(m * n))$  次迭代。
- 在每次迭代中，计算中间索引以及获取对应的行索引和列索引的操作都只需要常量时间  $O(1)$ 。
- 综上所述，整个搜索过程的时间复杂度为  $O(\log(m * n))$ 。

空间复杂度分析：

- 空间复杂度取决于所使用的额外空间。在该算法中，除了输入的矩阵和几个整数变量之外，没有使用额外的空间。
- 因此，额外空间的使用为  $O(1)$ ，是常量级别的。

### 392-判断子序列

 截屏2023-07-17 15.28.31.png

```
class Solution {
    public boolean isSubsequence(String s, String t) {
        //双指针
        int i = 0, j = 0;
        while(i < s.length() && j < t.length()){
            if(s.charAt(i) == t.charAt(j)){
                i++;
            }
            j++;
        }
        return i == s.length();
    }
}
```


该解决方案使用了双指针方法来判断字符串s是否为字符串t的子序列。下面是对时间和空间复杂度的分析：

时间复杂度分析：该算法的时间复杂度取决于字符串s和t的长度，分别为s.length()和t.length()。在最坏的情况下，指针i和j都需要遍历整个字符串s和t，因此时间复杂度为 $O(s.length() + t.length())$ 。

空间复杂度分析：该算法的空间复杂度是 $O(1)$ ，因为它只使用了常数级别的额外空间。无论输入的字符串s和t有多长，算法所使用的额外空间都保持不变。

综上所述，该算法的时间复杂度为 $O(s.length() + t.length())$ ，空间复杂度为 $O(1)$ 。

## 658.找到 K 个最接近的元素

 截屏2023-07-17 15.28.31.png

```
class Solution {
    public List<Integer> findClosestElements(int[] arr, int k, int x) {
        //find left bound position of x
        int p = left_bound(arr,x);
        int left = p - 1, right = p;
        //result linkedlist
        LinkedList<Integer> res = new LinkedList<>();
        while(right - left - 1 < k){
            if(left == -1){
                res.addLast(arr[right]);
                right ++;
            }else if(right == arr.length){
                res.addFirst(arr[left]);
                left --;
            }else if(x - arr[left] > arr[right] - x){
                res.addLast(arr[right]);
                right ++;
            }else{
                res.addFirst(arr[left]);
                left --;
            }
        }
        return res;
    }

    //找到左侧边界
    int left_bound(int[] arr, int target){
        int left = 0, right = arr.length - 1;
        while(left < right){
            int mid = left + (right - left)/2;
            if(arr[mid] == target){
                right = mid;
            }else if(arr[mid] < target){
                //find on right part
                left = mid + 1;
            }else if(arr[mid] > target){
                right = mid;
            }
        }
    }
}
```

```
    }  
    return left;  
}  
}
```

### 为什么要用LinkedList?

因为题目要求结果也是升序排列，所以我们要用到`addFirst`和`addLast`

时间复杂度分析：

1. 在 `left_bound` 方法中，使用二分查找找到左侧边界位置，时间复杂度为  $O(\log n)$ ，其中  $n$  是数组 `arr` 的长度。
2. 在 `findClosestElements` 方法中，循环执行的次数为  $k$ ，每次循环的操作都是常数时间复杂度，因此循环的时间复杂度为  $O(k)$ 。
  - 在每次循环中，执行插入操作 `res.addLast(arr[right])` 或 `res.addFirst(arr[left])`，插入操作的时间复杂度为  $O(1)$ 。
  - 执行 `right++` 和 `left--` 的操作也是常数时间复杂度。
  - 因此，整个 `while` 循环的时间复杂度为  $O(k)$ 。


综上所述，算法的总时间复杂度为  $O(\log n + k)$ 。

空间复杂度分析：

1. 空间复杂度主要取决于存储结果的链表 `res`，以及方法中使用的常数级别的额外空间。
2. 结果链表 `res` 的空间复杂度为  $O(k)$ ，因为最多存储  $k$  个元素。
3. 其他常数级别的额外空间的使用不会随输入规模的增加而增加，可以忽略不计。


综上所述，算法的总空间复杂度为  $O(k)$ 。

## 852. 山脉数组的峰顶索引

 截屏2023-07-18 13.34.28.png

```
class Solution {  
    public int peakIndexInMountainArray(int[] arr) {  
        //binary search  
        int left = 0, right = arr.length - 1;  
        while(left < right){  
            int mid = left + (right - left)/2;  
            if(arr[mid] > arr[mid + 1]){  
                //means we should find on left part  
                right = mid;  
            }else{  
                left = mid + 1;  
            }  
        }  
        return left;  
    }  
}
```

## 剑指 Offer 53 - II. 0~n-1中缺失的数字

 截屏2023-07-18 15.14.33.png

```
class Solution {
    public int missingNumber(int[] nums) {
        //binary search
        int left = 0, right = nums.length - 1;
        while(left <= right){
            int mid = left + (right - left)/2;
            if(nums[mid] > mid){
                //means the missing number is on the left part
                right = mid - 1;
            }else{
                //means right part
                left = mid + 1;
            }
        }
        return left;
    }
}
```

这道题考察二分查找算法。常规的二分搜索让你在 `nums` 中搜索目标值 `target`，但这道题没有给你一个显式的 `target`，怎么办呢？

其实，二分搜索的关键在于，你是否能够找到一些规律，能够在搜索区间中一次排除掉一半。比如让你在 `nums` 中搜索 `target`，你可以通过判断 `nums[mid]` 和 `target` 的大小关系判断 `target` 在左边还是右边，一次排除半个数组。


所以这道题的关键是，你是否能够找到一些规律，能够判断缺失的元素在哪一边？

其实是有规律的，你可以观察 `nums[mid]` 和 `mid` 的关系，如果 `nums[mid]` 和 `mid` 相等，则缺失的元素在右半边，如果 `nums[mid]` 和 `mid` 不相等，则缺失的元素在左半边。

在二分查找算法中，有一种常见的应用是搜索左侧边界的二分查找，我们可以借鉴这种思路来定位缺失元素的位置。

## 滑动窗口

### 3. 无重复字符的最长子串

 截屏2023-07-18 16.41.50.png

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        //Java solution
        int n = s.length();
        int left = 0, right = 0;
```

```
int result = 0;
HashSet<Character> set = new HashSet<>();
//start sliding
while(right < n){
    //add the character into window if it is not already in window
    char c = s.charAt(right);
    char d = s.charAt(left);
    if(!set.contains(c)){
        set.add(c);
        //update the result with the window size
        result = Math.max(result, right - left + 1);
        right ++;
    }else{
        //means set has c, shrink the window
        set.remove(d);
        left ++;
    }
}
return result;
}
```

具体流程如下：


1. 初始化一个空的 HashSet，用于存储当前窗口中的字符。
2. 每次右指针右移时，判断当前字符是否已经存在于 HashSet 中：
  - 如果不存在，将当前字符加入 HashSet，并继续向右移动右指针。
  - 如果存在，说明出现了重复字符，需要移动左指针来缩小窗口的大小，同时从 HashSet 中移除左指针对应的字符，继续右移右指针。
3. 在每次移动右指针或左指针后，都可以通过计算右指针和左指针之间的距离来更新最长子串的长度。
4. 重复上述步骤，直到右指针达到字符串的末尾。

这个算法的时间复杂度是  $O(n)$ ，其中  $n$  是字符串的长度。在最坏的情况下，我们需要遍历整个字符串一次。

空间复杂度是  $O(\min(n, m))$ ，其中  $n$  是字符串的长度， $m$  是字符集的大小（在这个问题中为 256，即 ASCII 字符集的大小）。在 HashSet 中最多存储  $m$  个字符，即字符集的大小。

因此，该算法使用了线性的时间复杂度和额外的空间来存储 HashSet，以判断字符是否重复。

### 438. 找到字符串中所有字母异位词

 截屏2023-07-18 17.28.46.png

```
class Solution {
    public List<Integer> findAnagrams(String s, String p) {
        int left = 0, right = 0;
        int valid = 0;
        List<Integer> res = new ArrayList<>();
        //corner case
        if(p.length() > s.length()){
            return res;
        }

        //one map for window, one for our need
        Map<Character, Integer> need = new HashMap<>();
        Map<Character, Integer> window = new HashMap<>();

        for(char c: p.toCharArray()){
            need.put(c, need.getDefault(c,0) + 1);
        }
        while(right < s.length()){
            char rightChar = s.charAt(right);
            window.put(rightChar, window.getDefault(rightChar, 0) + 1);
            //update valid
            if(need.containsKey(rightChar) &&
need.get(rightChar).equals(window.get(rightChar))){
                valid ++;
            }

            //当窗口大小等于我们p的长度的时候，我们就要收缩窗口并检查是不是异位词
            if(right - left + 1 == p.length()){
                //check if we are going to update result
                if(valid == need.size()){
                    res.add(left);
                }
                char leftChar = s.charAt(left);
                //update valid
                if(need.containsKey(leftChar) &&
need.get(leftChar).equals(window.get(leftChar))){
                    valid --;
                }
                window.put(leftChar, window.get(leftChar) - 1);
                left ++;
            }
            right ++;
        }
        return res;
    }
}
```

### 为什么要用equals, 不能用==么

在 Java 中，对于 `Integer` 类型的对象比较，不能直接使用 `==` 运算符，而是要使用 `equals()` 方法来进行比较。

这是因为 `==` 运算符在比较引用类型时，比较的是对象的引用地址，而不是对象的内容。而 `equals()` 方法被重写，用于比较对象的内容。

在上述代码中，`windowCount.get(rightChar)` 返回的是一个 `Integer` 对象，而 `pCount.get(rightChar)` 也返回一个 `Integer` 对象。我们需要比较这两个对象的内容是否相等，而不仅仅是比较它们的引用地址。

因此，需要使用 `equals()` 方法来判断这两个 `Integer` 对象的内容是否相等。如果内容相等，表示当前字符在窗口中的出现次数与 `p` 中的出现次数相同，我们才将 `matched` 计数器加一。

如果使用 `==` 运算符进行比较，将会比较两个 `Integer` 对象的引用地址，而不是它们的内容，这不符合我们的需求。

所以，在这种情况下，我们应该使用 `equals()` 方法来进行比较，以确保比较的是对象的内容。

希望这样的解释能够帮助你理解为什么要使用 `equals()` 方法而不是 `==` 运算符。如果还有其他问题，请随时提问。

### 我们什么时候要收缩window呢

具体来说，当窗口的大小等于字符串 `p` 的长度时，我们需要检查当前窗口是否是一个字母异位词。如果是，我们将窗口的左指针添加到结果列表中。无论是否是字母异位词，我们都需要收缩窗口。

在这个算法中，时间复杂度和空间复杂度如下：

时间复杂度：


- 遍历字符串 `p` 并构建 `need` 哈希表的时间复杂度为  $O(p)$ ，其中 `p` 是字符串 `p` 的长度。
- 在滑动窗口的过程中，我们遍历字符串 `s` 一次，每个字符只遍历一次，因此时间复杂度为  $O(s)$ ，其中 `s` 是字符串 `s` 的长度。
- 综合起来，算法的总时间复杂度为  $O(p + s)$ 。

空间复杂度：

- 需要额外的空间来存储 `need` 和 `window` 两个哈希表。在最坏的情况下，字符串 `p` 中的字符都是唯一的，因此需要存储 `p` 中的每个字符及其出现次数，所以 `need` 哈希表的空间复杂度为  $O(p)$ 。
- 窗口的大小最多为字符串 `p` 的长度，因此 `window` 哈希表的空间复杂度也为  $O(p)$ 。
- 综合起来，算法的总空间复杂度为  $O(p)$ 。

综上所述，该算法的时间复杂度为  $O(p + s)$ ，空间复杂度为  $O(p)$ 。

## 567. 字符串的排列

 截屏2023-07-18 22.39.56.png

```
class Solution {
    public boolean checkInclusion(String s1, String s2) {
        //滑动窗口
        int left = 0, right = 0;
        int valid = 0;
        Map<Character, Integer> need = new HashMap<>();
```

```
Map<Character, Integer> window = new HashMap<>();

//corner case
if(s1.length() > s2.length()){
    return false;
}

//fill in the need
for(char c: s1.toCharArray()){
    need.put(c, need.getOrDefault(c,0) + 1);
}

//sliding
while(right < s2.length()){
    char rightChar = s2.charAt(right);
    window.put(rightChar, window.getOrDefault(rightChar,0) + 1);
    //update valid
    if(need.containsKey(rightChar) &&
need.get(rightChar).equals(window.get(rightChar))){
        valid ++;
    }

    //if the length of window is length of s1, check and shrink
    if(right - left + 1 == s1.length()){
        if(valid == need.size()){
            return true;
        }
        char leftChar = s2.charAt(left);
        if(need.containsKey(leftChar) &&
need.get(leftChar).equals(window.get(leftChar))){
            valid --; //update valid
        }
        window.put(leftChar, window.get(leftChar) - 1);
        left ++;
    }

    //move right
    right ++;
}
return false;
}
```

## 注意

再想到收缩窗口的同时，也要注意在if语句中检查是否valid然后决定return的值

在这个算法中，时间复杂度和空间复杂度如下：

时间复杂度：

- 填充 need 哈希表的时间复杂度为  $O(s1)$ ，其中  $s1$  是字符串  $s1$  的长度。



- 在滑动窗口的过程中，我们遍历字符串 `s2` 一次，每个字符只遍历一次，因此时间复杂度为  $O(s2)$ ，其中 `s2` 是字符串 `s2` 的长度。
- 综合起来，算法的总时间复杂度为  $O(s1 + s2)$ 。


空间复杂度：

- 需要额外的空间来存储 `need` 和 `window` 两个哈希表。在最坏的情况下，字符串 `s1` 中的字符都是唯一的，因此需要存储 `s1` 中的每个字符及其出现次数，所以 `need` 哈希表的空间复杂度为  $O(s1)$ 。
- 窗口的大小最多为字符串 `s1` 的长度，因此 `window` 哈希表的空间复杂度也为  $O(s1)$ 。
- 综合起来，算法的总空间复杂度为  $O(s1)$ 。

综上所述，该算法的时间复杂度为  $O(s1 + s2)$ ，空间复杂度为  $O(s1)$ 。

希望这个分析对你有所帮助。如果你还有其他问题，请随时提问。

## 76. 最小覆盖子串

 截屏2023-07-19 12.16.05.png

```
class Solution {
    public String minWindow(String s, String t) {
        int left = 0, right = 0;
        //the start of the string
        int start = 0;
        int valid = 0;

        //the length of the string
        int len = Integer.MAX_VALUE;

        Map<Character, Integer> need = new HashMap<>();
        Map<Character, Integer> window = new HashMap<>();

        for (char c : t.toCharArray()) {
            need.put(c, need.getOrDefault(c, 0) + 1);
        }

        while (right < s.length()) {
            char rightChar = s.charAt(right);
            right++;
            //if the char is what we need, we put it into the window
            if (need.containsKey(rightChar)) {
                window.put(rightChar, window.getOrDefault(rightChar, 0) + 1);

                //update valid
                if (need.get(rightChar).equals(window.get(rightChar))) {
                    valid++;
                }
            }

            //if we find the window is valid, we shrink the window
            while (valid == need.size()) {

```

```

        char leftChar = s.charAt(left);
        //update len and start
        if (right - left + 1 < len) {
            start = left;
            len = right - left + 1;
        }

        left++;

        if (need.containsKey(leftChar)) {
            //update valid
            if (need.get(leftChar).equals(window.get(leftChar))) {
                valid--;
            }
            //move the element out of the window
            window.put(leftChar, window.get(leftChar) - 1);
        }

    }

    return len == Integer.MAX_VALUE ? "" : s.substring(start, start +
len - 1);
}
}

```

### 注意


- Java中substring用法。
- 这道题中window只要添加符合要求的char就行。

上面提供的代码的时间复杂度为 $O(n)$ ，其中 $n$ 是输入字符串 $s$ 的长度。这是因为代码使用两个指针（ $left$ 和 $right$ ）遍历字符串 $s$ ，并执行依赖于 $s$ 长度的操作。在最坏情况下，内部的 $while$ 循环可能运行整个字符串 $s$ 的长度，导致线性时间复杂度。

代码的空间复杂度为 $O(m)$ ，其中 $m$ 是输入字符串 $t$ 中不同字符的数量。这是因为代码使用两个哈希映射（ $need$ 和 $window$ ）来存储字符串 $t$ 中字符的频率以及当前窗口 $s$ 中的字符。哈希映射所需的空间取决于字符串 $t$ 中不同字符的数量，即 $m$ 。

因此，代码的总体时间复杂度为 $O(n)$ ，空间复杂度为 $O(m)$ 。

## 239. 滑动窗口最大值

 截屏2023-07-19 15.51.34.png

```

class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        Monotonicqueue window = new Monotonicqueue();
        List<Integer> res = new ArrayList<>();

        for(int i = 0; i < nums.length; i++){

```

```

        if(i < k - 1){//add first k-2 elements
            window.push(nums[i]);
        }else{
            window.push(nums[i]);
            res.add(window.max());
            //用得到这句话的情况有[最大的，比这个小的其他元素]，我们需要把最大的去
            掉
            window.pop(nums[i - k + 1]);
        }
    }

    int[] result = new int[res.size()];
    for(int i = 0; i < result.length; i++){
        result[i] = res.get(i);
    }

    return result;
}

class Monotonicqueue{
    LinkedList<Integer> maxq = new LinkedList<>();

    //保证队列的头是最大的元素，并且先进先出
    public void push(int n){
        while(!maxq.isEmpty() && maxq.getLast() < n){
            maxq.pollLast();//把小的元素都去掉，没用了
        }
        maxq.addLast(n);
    }

    public int max(){
        return maxq.getFirst();
    }

    public void pop(int n){
        //我们的window要么是有且只有一个最大的元素，要么就是[最大的，比这个小的其他元
        素]

        if(maxq.getFirst() == n){
            maxq.pollFirst();
        }
    }
}

```

### 为什么要用单调队列

保证了队列始终维持递减的顺序，并且可以在  $O(1)$  的时间复杂度内获取当前队列的最大值。这样，在一系列元素的动态更新过程中，我们可以高效地跟踪最大值的变化。

- 时间复杂度：整个算法的时间复杂度是  $O(n)$ ，其中  $n$  是数组 `nums` 的长度。遍历一次数组需要  $O(n)$  的时间，每个元素最多进出队列一次，而队列的操作复杂度是  $O(1)$ 。所以总体时间复杂度

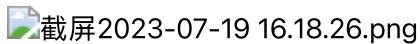
是  $O(n)$ 。

- 空间复杂度：算法使用了一个 `res` 列表来存储结果，其最大长度是  $n-k+1$ ，所以空间复杂度是  $O(n-k+1)$ 。另外，`Monotonicqueue` 类内部使用了一个双向链表来存储元素，其长度最大不超过  $k$ ，所以空间复杂度是  $O(k)$ 。综合起来，算法的空间复杂度是  $O(\max(n-k+1, k))$ 。

因此，这段代码实现了一个时间复杂度为  $O(n)$ ，空间复杂度为  $O(\max(n-k+1, k))$  的滑动窗口最大值算法。

## 其他题目

### 26. 删除有序数组中的重复项

 截屏2023-07-19 16.18.26.png

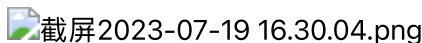
```
class Solution {
    public int removeDuplicates(int[] nums) {
        //双指针
        if(nums.length == 0){
            return 0;
        }

        int slow = 0, fast = 0;
        while(fast < nums.length){
            if(nums[slow] != nums[fast]){
                nums[slow + 1] = nums[fast];
                slow ++;
            }
            fast ++;
        }
        return slow + 1;
    }
}
```

这道题的思路就是，双指针遍历：如果我遇到相同元素（包括最开始两个指针都在0号位的情况），我就只动 `fast`；如果不同，就说明 `slow+1` 的元素应该是我 `fast` 元素。

这段代码实现了一个时间复杂度为  $O(n)$ ，空间复杂度为  $O(1)$  的移除有序数组重复元素的算法。

### 27. 移除元素


 截屏2023-07-19 16.30.04.png

```
class Solution {
    public int removeElement(int[] nums, int val) {
        //双指针
        int slow = 0, fast = 0;
        while(fast < nums.length){
            if(nums[fast] != val){
```

```
        nums[slow] = nums[fast];
        slow ++;
    }
    fast ++;
}
return slow;
}
```

这道题的思路是，如果 `fast != val`,就说明这是我们要的值，把它给slow并且移动slow。如果 `fast == val`，那么只要动fast。


### 283. 移动零

 截屏2023-07-19 17.08.21.png

```
class Solution {
    public void moveZeroes(int[] nums) {
        int fast = 0, slow = 0;
        while(fast < nums.length){
            if(nums[fast] != 0){
                nums[slow] = nums[fast];
                slow ++;
            }
            fast ++;
        }
        while(slow < nums.length){
            nums[slow] = 0;
            slow ++;
        }
    }
}
```

和上一道题一样，只不过后面多了一个加0的步骤

### 83. 删除排序链表中的重复元素

 截屏2023-07-19 17.22.08.png

```
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode slow = head, fast = head;
        while(fast != null){
            if(slow.val != fast.val){
                slow.next = fast;
                slow = fast;
            }
            fast = fast.next;
        }
    }
}
```


```
    }  
    slow.next = null;  
    return head;  
}  
}
```

### 注意

`slow.next = null;` 这句话少不了

这段代码实现了一个时间复杂度为  $O(n)$ ，空间复杂度为  $O(1)$  的删除链表中重复元素的算法。

### 剑指 Offer 21. 调整数组顺序使奇数位于偶数前面


 截屏2023-07-19 22.54.53.png

```
class Solution {  
    public int[] exchange(int[] nums) {  
        int slow = 0, fast = 0 ;  
        while(fast < nums.length){  
            if(nums[fast] % 2 == 1){  
                int temp = nums[fast];  
                nums[fast] = nums[slow];  
                nums[slow] = temp;  
                slow ++;  
            }  
            fast ++;  
        }  
        return nums;  
    }  
}
```

### 解题思路

如果遇到奇数就和slow调换位置

### 剑指 Offer 57. 和为s的两个数字


 截屏2023-07-20 10.33.39.png

```
class Solution {  
    public int[] twoSum(int[] nums, int target) {  
        int left = 0 , right = nums.length - 1;  
        //左右指针  
        while(left < right){  
            //get the sum  
            int sum = nums[left] + nums[right];  
            if( sum == target){  
                return new int[]{nums[left],nums[right]};  
            }else if(sum < target){  
                left ++;  
            }else if(sum > target){  
                right --;  
            }  
        }  
        return new int[]{};  
    }  
}
```

```
        left ++;
    }else{
        right --;
    }
}
return null;
}
```

时间复杂度：O(N)

## 82. 删除排序链表中的重复元素 II

 截屏2023-07-20 11.05.45.png

```
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode dummy = new ListNode(-1);
        ListNode slow = dummy, fast = head;
        while(fast != null){
            if(fast.next != null && fast.next.val == fast.val){
                while(fast.next != null && fast.next.val == fast.val){
                    fast = fast.next;
                }
                fast = fast.next;
                if(fast == null){
                    slow.next = fast;
                }
            }else{
                slow.next = fast;
                fast = fast.next;
                slow = slow.next;
            }
        }
        return dummy.next;
    }
}
```

### 注意

- 这个题目是要把重复的元素去掉，也就是不留下来。所以当我们发现重复元素时，需要多加一个 while，然后下一次走if的时候保证fast指针在重复元素的next；
- 不排除一种情况是 1 2 2 2， 这样的话需要在判断里面多加个if来让slow指向null

给定的代码是用于删除链表中重复元素的问题。以下是对代码的复杂度分析：

#### 1. 时间复杂度：

- 初始化 dummy 节点和两个指针 slow 和 fast：O(1)。
- while 循环：最坏情况下，需要遍历整个链表，即 O(n)，其中 n 是链表的长度。

- 在 while 循环中，执行常数时间的操作：比较节点值，更新指针。
- 内层 while 循环（用于跳过重复元素）的总体时间复杂度为  $O(k)$ ，其中  $k$  是重复元素的数量，但  $k \leq n$ 。
- 总体时间复杂度为  $O(n)$ 。

## 2. 空间复杂度：

- 创建了一个 dummy 节点： $O(1)$ 。
- 使用了两个指针 slow 和 fast： $O(1)$ 。
- 没有使用额外的数据结构，只使用了常数级的额外空间。

综上所述，给定代码的时间复杂度为  $O(n)$ ，空间复杂度为  $O(1)$ 。