大厂算法和数据结构解析

第一章 算法简介

作为程序员，大家对“算法”这个词一定非常熟悉。不论是理论学习、求职面试，还是项目开发，算法往往都是程序员关注的重点之一。那算法到底是什么？想要系统地学习算法，又应该从哪里入手呢？

本课程就将为大家打开算法学习的大门。首先，我们应该了解算法的基本概念。

1.1 算法的基本概念

1.1.1 什么是算法

算法（Algorithm），就是“计算方法”，指解决一个问题具体的步骤和方法。

对于计算机而言，就是一系列解决问题的清晰指令。也就是说，对于一个问题，我们可以通过算法来描述解决的策略和步骤；对于规范的输入，按照算法描述的步骤，就可以在有限时间内得到对应的输出。

1.1.2 为什么要学习算法

首先，算法是计算机程序的核心。在一个计算机程序中，有两个非常重要的部分：数据结构和算法。数据结构决定了程序中数据组织和存储的形式，而算法决定了程序的功能和效率。算法在具体应用时，与数据结构密不可分，是一个程序的灵魂。

其次，算法是程序员的核心竞争力。算法是解决问题的方法和步骤，所以掌握了算法，就可以拥有更强的解决问题的能力。对于一个程序员而言，这往往比学会用一个框架、一个工具更加重要。

再次，算法是 IT 大厂面试的必考环节。大家可能知道，在 IT 特别是互联网公司的面试过程中，数据结构和算法是非常重要的一部分，为什么大厂都青睐于考察算法呢？总结起来，考察的原因以下几点：

- 看一个程序员的技术功底是否扎实，考算法是最好的方式；
- 看一个程序员的学习能力，和成长潜力，最好就是看他的算法能力；
- 算法能力强，可以在面对新问题时，有更强的分析并解决问题的能力；
- 算法能力，是设计一个高性能系统、性能优化的必备基础。

所以，算法是程序员绕不过去的必修课，也是走向架构师的必经之路。

1.1.3 怎样学习算法

首先，在学习算法之前，应该至少熟练掌握一门编程语言。本课程中的代码实现，都会以 java 为例来讲解。

其次，算法和数据结构密不可分，在系统学习算法之前，最好能够对基本的数据结构，数组、链表、哈希表、栈、队列、树都有充分的了解。在后续的课程中，我们会以算法讲解为主、穿插复习一些数据结构的基本知识。当然，算法本身是解题方法，有些也是不依赖于数据结构的，所以即使没有系统学过数据结构，同样可以开始算法的学习。

最后，算法学习的捷径，就是用算法去解决大量的具体问题，也就是通常所说的“刷题”。如果目的在于通过大厂面试，那这一步更是必不可少的准备。最经典的刷题网站，毫无疑问就是 leetcode（力扣）。所以我们接下来的学习过程中，就会以 leetcode 的原题为例，分门别类进行算法的讲解，在具体解题的过程中巩固深化算法的学习。

1.2 算法的特征

一个算法应该具有以下五个重要的特征：

- 有穷性（Finiteness）

算法的有穷性，是指算法必须能在执行有限个步骤之后终止。

- 确切性（Definiteness）

算法的每一步骤必须有确切的定义。

- 输入项（Input）

一个算法有 0 个或多个输入，以刻画运算对象的初始情况。所谓 0 个输入，是指算法本身定出了初始条件。

- 输出项（Output）

一个算法有一个或多个输出，以反映对输入数据加工后的结果。

- 可行性（Effectiveness）

算法中执行的任何计算步骤都是可以被分解为基本的可执行的操作步骤，即每个计算步骤都可以在有限时间内完成（也称之为有效性）。

1.3 算法复杂度

基于算法的有穷性，我们可以知道算法运行消耗的时间不能是无限的。而对于一个问题的处理，可能有多个不同的算法，它们消耗的时间一般是不同的；运行过程中占用的空间资源也是不同的。

这就涉及到对算法的性能考察。主要有两方面：时间和空间。在计算机算法理论中，用时间复杂度和空间复杂度来分别从这两方面衡量算法的性能。

1.3.1 时间复杂度（Time Complexity）

算法的时间复杂度，是指执行算法所需要的计算工作量。

一般来说，计算机算法是问题规模 n 的函数 f(n)，算法的时间复杂度也因此记做：T(n)=O(f(n))。

问题的规模 n 越大，算法执行的时间的增长率与 f(n) 的增长率正相关，称作渐进时间复杂度（Asymptotic Time Complexity）。

1.3.2 空间复杂度

算法的空间复杂度，是指算法需要消耗的内存空间。有时候做递归调用，还需要考虑调用栈所占用的空间。

其计算和表示方法与时间复杂度类似，一般都用复杂度的渐近性来表示。同时间复杂度

相比，空间复杂度的分析要简单得多。

所以，我们一般对程序复杂度的分析，重点都会放在时间复杂度上。

1.3.3 时间复杂度的计算

要想衡量代码的“工作量”，我们需要将每一行代码，拆解成计算机能执行一条条“基本指令”。这样代码的执行时间，就可以用“基本指令”的数量来表示了。

真实的计算机系统里，基本指令包括：

算术指令（加减乘除、取余、向上向下取整）、数据移动指令（装载、存储、赋值）、控制指令（条件或无条件跳转，子程序调用和返回）。

我们来看一些具体的代码，分析一下它们的时间复杂度：

- int a = 1；

  简单赋值操作，运行时间 1（1 个单位）

- if (a > 1) {}

  简单判断操作、条件跳转，运行时间 1

- for (int i = 0; i < N; i++) { System.out.println(i); }

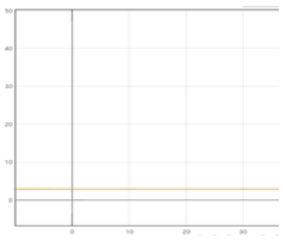
  有循环，运行时间 1（i 赋初值）+ N+1（判断）+N（打印）+N（i 自增）= 3N + 2

1.3.4 复杂度的大 O 表示法

比起代码具体运行的时间，我们更关心的是，当它的输入规模增长的时候，它的执行时间我们是否还能够接受。

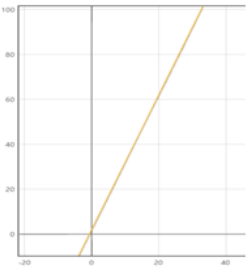
不同的算法，运行时间随着输入规模 n 的增长速度是不同的。我们可以把代码执行时间，表示成输入规模 n 的函数 T(n)。

```
int a = 1;
a = 2;
a = 3;
```



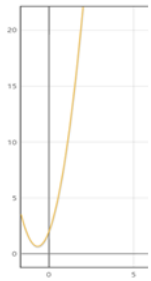
T(n) = 3

```
for (int i = 0; i < n; i++) {
    System.out.println("test");
}
```



T(n) = 3n + 2

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
        System.out.println("test");
}
```



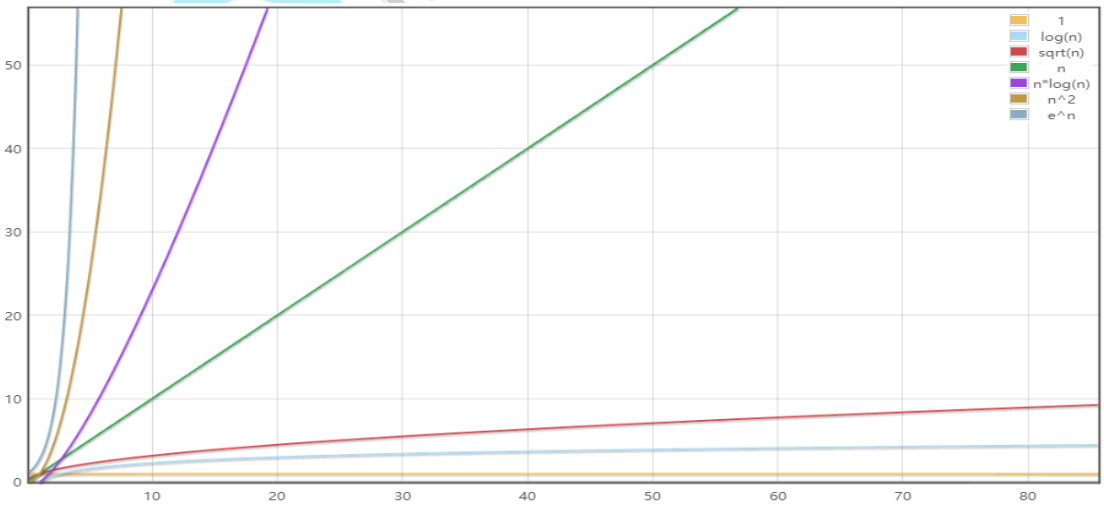
T(n) = $3n^2 + 4n + 2$

在算法分析中，一般用大 O 符号来表示函数的渐进上界。对于给定的函数 g(n)，我们用 O(g(n))来表示以下函数的集合：

- O(g(n)) = { f(n): 存在正常量 c 和 n_0，使对所有 $n \geq n_0$，有 $0 \leq f(n) \leq cg(n)$ }

这表示，当数据量达到一定程度时，g(n) 的增长速度不会超过 O(g(n))限定的范围。也就是说，大 O 表示了函数的“阶数”，阶数越高，增长趋势越大，后期增长越快。

下图画出了常见的算法复杂度：



1.4 算法的分类

可以用两种不同的原则，来对算法做一个分类整理：

- 按照应用的目的来划分

  搜索算法、排序算法、字符串算法、图算法、最优化算法、数学（数论）算法

- 按照具体实现的策略划分

  暴力法、增量法、分治法、贪心、动态规划、回溯、分支限界法

1.5 经典算法

在实际应用中，有一些经典算法和策略，都可以作为解决问题的思路：

- 二分查找
- 快速排序、归并排序
- KMP 算法
- 快慢指针（双指针法）
- 普利姆（Prim）和 克鲁斯卡尔（Kruskal）算法
- 迪克斯特拉（Dijkstra）算法
- 其它优化算法：模拟退火、蚁群、遗传算法

接下来，我们就不同的数据结构和算法策略进行分类，用不同的算法在各章节中解决某一类问题。

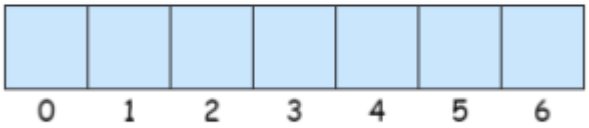
第二章 数组问题讲解

在程序设计中，为了处理方便，常常需要把具有相同类型的若干元素按有序的形式组织起来，这种形式就是数组（Array）。

数组是程序中最常见、也是最基本的数据结构。在很多算法问题中，都少不了数组的处理和转换。

对数组进行处理需要注意以下特点：

- 首先，数组会利用 索引 来记录每个元素在数组中的位置，且在大多数编程语言中，索引是从 0 算起的。我们可以根据数组中的索引，快速访问数组中的元素。事实上，这里的索引其实就是内存地址。



- 其次，作为线性表的实现方式之一，数组中的元素在内存中是**连续**存储的，且每个元素占用相同大小的内存。

接下来，我们就以 LeetCode 上一些数组相关的题目为例，来学习解决数组问题的算法。

2.1 两数之和（#1）

2.2.1 题目说明

给定一个整数数组 nums 和一个目标值 target，请你在该数组中找出和为目标值的那两个整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。

示例:

```
给定 nums = [2, 7, 11, 15], target = 9
因为 nums[0] + nums[1] = 2 + 7 = 9
所以返回 [0, 1]
```

2.2.2 方法一：暴力法

看到一道算法题，首先考虑暴力解法，再进行优化。

暴力法其实非常简单：把所有数、两两组合在一起，计算它们的和，如果是 target，就输出。

我们可以在代码中实现一下：

```
public int[] twoSum(int[] nums, int target) {
    for( int i = 0; i < nums.length; i++ ){
        for( int j = i + 1; j < nums.length; j++ ){
            if( nums[i] + nums[j] == target )
                return new int[] {i, j};
        }
    }
    throw new IllegalArgumentException("No two sum solution");
}
```

复杂度分析

- 时间复杂度：O(n^2)，对于每个元素，我们试图通过遍历数组的其余部分来寻找它所对应的目标元素，这将耗费 O(n)。
- 空间复杂度：O(1)。

2.2.3 方法二：两遍哈希表

为了对运行时间复杂度进行优化，我们需要一种更有效的方法来检查数组中是否存在目标元素。如果存在，我们需要找出它的索引。这可以使用哈希表来实现。

具体实现方法，最简单就是使用两次迭代。

在第一次迭代中，我们将每个元素的值和它的索引添加到表中；然后，在第二次迭代中，我们将检查每个元素所对应的目标元素 (target-nums[i]) 是否存在于表中。

代码如下：

```
public int[] twoSum(int nums[], int target) {
    Map<Integer, Integer> map = new HashMap<>();

    // 遍历数组，全部保存到 hashmap 中
    for(int i = 0; i < nums.length; i++){
        map.put(nums[i], i);
    }

    // 遍历数组，挨个查找对应的“那个数”在不在 map 中
    for( int i = 0; i < nums.length; i++ ){
        int thatNum = target - nums[i];
        if( map.containsKey(thatNum) && map.get(thatNum) != i )
            return new int[] {i, map.get(thatNum)};
    }
    throw new IllegalArgumentException("No two sum solution");
}
```

复杂度分析

- 时间复杂度：O(N)，我们把包含有 N 个元素的列表遍历两次。由于哈希表将查找时间缩短到 O(1)，所以时间复杂度为 O(N)。
- 空间复杂度：O(N)，所需的额外空间取决于哈希表中存储的元素数量，该表中存储了 N 个元素。

2.2.4 方法三：一遍哈希表

在上述算法中，我们对哈希表进行了两次扫描，这其实是不必要的。在进行迭代并将元素插入到表中的同时，我们可以直接检查表中是否已经存在当前元素所对应的目标元素。如果它存在，那我们已经找到了对应解，并立即将其返回。这样，只需要扫描一次哈希表，就可以完成算法了。

代码如下：

```
public int[] twoSum(int nums[], int target) {
    Map<Integer, Integer> map = new HashMap<>();

    for( int i = 0; i < nums.length; i++ ){
        int thatNum = target - nums[i];
        if( map.containsKey(thatNum) && map.get(thatNum) != i )
            return new int[] {map.get(thatNum), i};
        map.put(nums[i], i);
    }
    throw new IllegalArgumentException("No two sum solution");
}
```

复杂度分析

- 时间复杂度：O(N)，我们只遍历了包含有 N 个元素的列表一次。在表中进行的每次查找只花费 O(1) 的时间。其实这个过程中，我们也借鉴了动态规划的思想、把子问题解保存起来，后面用到就直接查询。
- 空间复杂度：O(N)，所需的额外空间取决于哈希表中存储的元素数量，该表最多需要存储 N 个元素。

2.2 三数之和（#15）

2.2.1 题目说明

给定一个包含 n 个整数的数组 nums，判断 nums 中是否存在三个元素 a，b，c ，使得 a + b + c = 0 ？找出所有满足条件且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例:

```
给定数组 nums = [-1, 0, 1, 2, -1, -4],
满足要求的三元组集合为:
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

2.2.2 分析

这个问题比起两数之和来，显然要复杂了一些，而且由于结果可能有多种情况，还要考虑去重，整体难度提升了不少。

最后的返回，就不再是一个简单的数组了，而是“数组的数组”，每一组解都是一个数组，最终有多组解都要返回。

2.2.3 方法一：暴力法

最简单的办法，当然还是暴力法。基本思路是，每个人都先去找到另一个人，然后再一起逐个去找第三个人。

很容易想到，实现起来就是三重循环：这个时间复杂度是 O(n^3)。

代码如下：

```
public List<List<Integer>> threeSum(int[] nums) {
    int n = nums.length;
    List<List<Integer>> resultList = new ArrayList<>();
    // 三重循环，遍历所有的三数组合
    for( int i = 0; i < n - 2; i++ ){
        for( int j = i + 1; j < n - 1; j++ ){
            for( int k = j + 1; k < n; k++ ){
```

```
                if( nums[i] + nums[j] + nums[k] == 0 ){
                    resultList.add(
                        Arrays.asList(nums[i], nums[j], nums[k]));
                }
            }
        }
    }
    return resultList;
}
```

运行一下，我们会发现，这个结果其实是不正确的没有去重，同样的三元组在结果中无法排除。比如-1，0，1 会出现两次。而且时间复杂度非常高，是 N^3。

所以接下来，我们就要做一些改进，试图降低时间复杂度，而且解决去重问题。

2.2.4 暴力法的改进：结果去重

要做去重，自然首先想到的，就是把结果保存到一张 hash 表里。仿照两数之和，直接存到 HashMap 里查找，代码如下：

```
public List<List<Integer>> threeSum(int[] nums) {
    int n = nums.length;
    List<List<Integer>> result = new ArrayList<>();

    Map<Integer, List<Integer>> hashMap = new HashMap<>();

    // 遍历数组，寻找每个元素的 thatNum
    for( int i = 0; i < n; i++ ){
        int thatNum = 0 - nums[i];
```

```
        if( hashMap.containsKey(thatNum) ){
            List<Integer> tempList = new ArrayList<>(
                  hashMap.get(thatNum));
            tempList.add(nums[i]);
            result.add(tempList);
            continue;
        }
        for( int j = 0; j < i; j++ ){
            int newKey = nums[i] + nums[j];
            if( ! hashMap.containsKey(newKey) ){
                List<Integer> tempList = new ArrayList<>();
                tempList.add(nums[j]);
                tempList.add(nums[i]);
                hashMap.put( newKey, tempList );
            }
        }
    }
    return result;
}
```

时间复杂度降为 N^2，空间复杂度 O(N)。

但是，我们加一个输入[0,0,0,0]，会发现 结果不正确。

因为尽管通过 HashMap 存储可以去掉相同二元组的计算结果的值，但没有去掉重复的输出（三元组）。这就导致，0 对应在 HashMap 中有一个值（0，List（0，0））， 第三个 0 来了会输出一次，第四个 0 来了又会输出一次。

如果希望解决这个问题，那就需要继续加入其它的判断来做去重，整个代码复杂度会变得更高。

2.2.5 方法二：双指针法

暴力法搜索时间复杂度为 O(N^3)，要进行优化，可通过双指针动态消去无效解来提高效率。

双指针的思路，又分为左右指针和快慢指针两种。

我们这里用的是左右指针。左右指针，其实借鉴的就是分治的思想，简单来说，就是在数组头尾各放置一个指针，先让头部的指针（左指针）右移，移不动的时候，再让尾部的指针（右指针）左移：最终两个指针相遇，那么搜索就结束了。

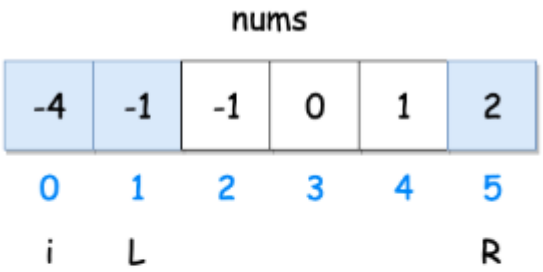
（1）双指针法铺垫： 先将给定 nums 排序，复杂度为 O(NlogN)。

首先，我们可以想到，数字求和，其实跟每个数的大小是有关系的，如果能先将数组排序，那后面肯定会容易很多。

之前我们搜索数组，时间复杂度至少都为 O(N^2)，而如果用快排或者归并，排序的复杂度，是 O(NlogN)，最多也是 O(N^2)。所以增加一步排序，不会导致整体时间复杂度上升。

下面我们通过图解，来看一下具体的操作过程。

（2）初始状态，定义左右指针 L 和 R，并以指针 i 遍历数组元素。



i = 0
L = i + 1, R = nums.length - 1

固定 3 个指针中最左(最小)数字的指针 i，双指针 L，R 分设在数组索引 (i, len(nums)) 两端，所以初始值，i=0；L=i+1；R=nums.length-1

通过 L、R 双指针交替向中间移动，记录对于每个固定指针 i 的所有满足 nums[i] + nums[L] + nums[R] == 0 的 L,R 组合。

两个基本原则：

- 当 nums[i] > 0 时直接 break 跳出：因为 nums[R] >= nums[L] >= nums[i] > 0，即 3 个数字都大于 0 ，在此固定指针 i 之后不可能再找到结果了。
- 当 i > 0 且 nums[i] == nums[i - 1]时，即遇到重复元素时，跳过此元素 nums[i]：因为已经将 nums[i - 1] 的所有组合加入到结果中，本次双指针搜索只会得到重复组合。

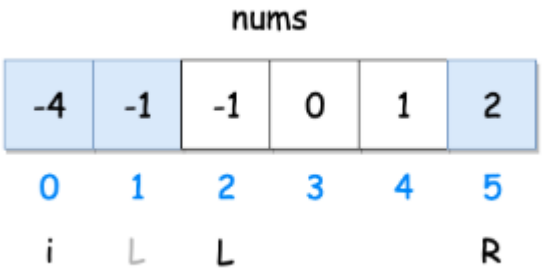
（3）固定 i，判断 sum，然后移动左右指针 L 和 R。

L，R 分设在数组索引 (i, len(nums)) 两端，当 L < R 时循环计算当前三数之和：

sum = nums[i] + nums[L] + nums[R]

并按照以下规则执行双指针移动：

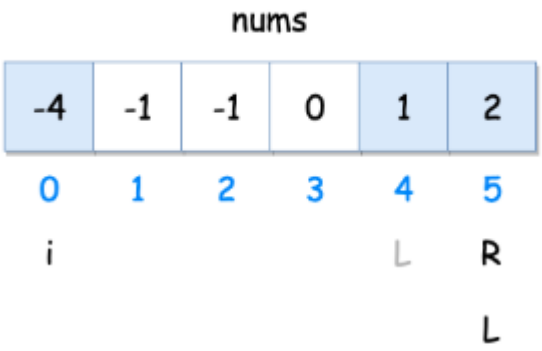
- 当 sum < 0 时，L ++并跳过所有重复的 nums[L]；



sum = nums[i] + nums[L] + nums[R] = -3

∵ sum < 0

∴ L++



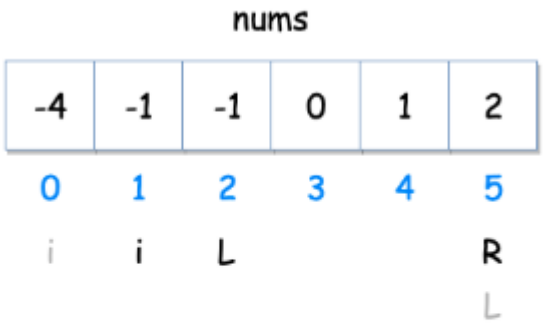
sum = nums[i] + nums[L] + nums[R] = -1

∵ sum < 0

∴ L++

- 由于 sum<0，L 一直右移，直到跟 R 重合。如果依然没有结果，那么 i++，换下一个数考虑。

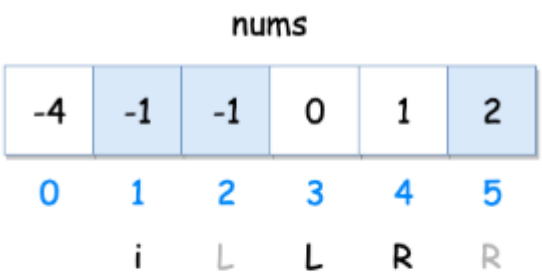
换下一个数，i++，继续移动双指针：



L与R相遇，i++
L = i + 1，R = nums.length - 1

初始同样还是 L=i+1，R=nums.length-1。同样，继续判断 sum。

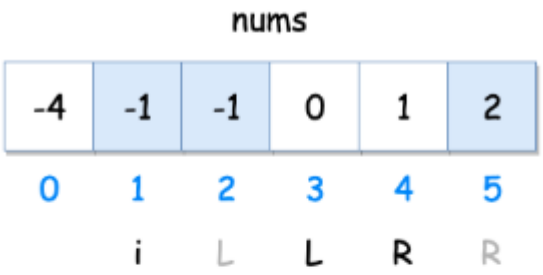
- 找到一组解之后，继续移动 L 和 R，判断 sum，如果小于 0 就右移 L，如果大于 0 就左移 R：



sum = nums[i] + nums[L] + nums[R] = 0
∵ sum == 0
∴ ans = [[-1,-1,2]]
L++
R--

找到一组解[-1,-1,2]，保存，并继续右移 L。判断 sum，如果这时 sum=-1+0+2>0，（R 还没变，还是 5），那么就让 L 停下，开始左移 R。

- 一直移动，又找到一组解



sum = nums[i] + nums[L] + nums[R] = 0

∵ sum == 0

∴ ans = [[-1,-1,2]]

L++

R--

如果又找到 sum=0 的一组解，把当前的[-1,0,1]也保存到结果数组。继续右移 L。

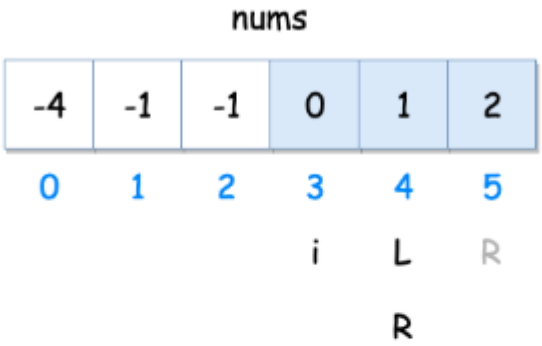
- 如果 L 和 R 相遇或者 L>R，代表当前 i 已经排查完毕，i++；如果 i 指向的数跟 i-1 一样，那么直接继续 i++，考察下一个数；

∵ L > R ∴ i++

∵ nums[i] == nums[i-1] ∴ i++

L = i + 1, R = nums.length - 1

- 这时 i=3，类似地，当 sum > 0 时，R 左移 R -= 1，并跳过所有重复的 nums[R]；

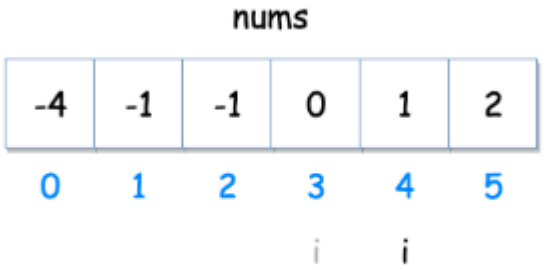


sum = nums[i] + nums[L] + nums[R] = 3

∵ sum > 0

∴ R--

- L 和 R 相遇，结束当前查找，i++。



L与R相遇，i++

∵ nums[i] > 0 ∴ 和不可能为0，结束

ans = [[-1,-1,2] , [-1,0,1]] ☆

当 nums[i] > 0 时直接 break 跳出：过程结束。

所以，最终的结果，就是[-1,-1,2],[-1,0,1]。

代码如下：

```
public List<List<Integer>> threeSum(int[] nums){
    int n = nums.length;
    List<List<Integer>> result = new ArrayList<>();

    // 先对数组进行排序
    Arrays.sort(nums);

    for( int i = 0; i < n; i++ ){
        if( nums[i] > 0 )
            break;
        if( i > 0 && nums[i] == nums[i-1] )
            continue;
```

```
        // 定义左右指针（索引位置）
        int lp = i + 1;
        int rp = n - 1;
        // 只要左右不重叠，就继续移动指针
        while( lp < rp ){
            int sum = nums[i] + nums[lp] + nums[rp];
            if( sum == 0 ){
                result.add(Arrays.asList(nums[i], nums[lp], nums[rp]));
                lp ++;
                rp --;
                while( lp < rp && nums[lp] == nums[lp - 1] )
                    lp ++;
                while( lp < rp && nums[rp] == nums[rp + 1] )
                    rp --;
            }
            else if( sum < 0 )
                lp ++;
            else
                rp --;
        }
    }
    return result;
}
```

复杂度分析：

- 时间复杂度 O(N^2)：其中固定指针 k 循环复杂度 O(N)，双指针 i,j 复杂度 O(N)。比暴力法的 O(n^3)，显然有了很大的改善。
- 空间复杂度 O(1)：指针使用常数大小的额外空间。

尽管时间复杂度依然为 O(n^2)，但是过程中避免了复杂的数据结构，空间复杂度仅为常数级 O(1)，可以说，双指针法是一种很巧妙、很优雅的算法设计。

2.3 下一个排列（#31）

2.3.1 题目说明

实现获取下一个排列的函数，算法需要将给定数字序列重新排列成字典序中下一个更大的排列。

如果不存在下一个更大的排列，则将数字重新排列成最小的排列（即升序排列）。

必须原地修改，只允许使用额外常数空间。

以下是一些例子，输入位于左侧列，其相应输出位于右侧列。

```
1,2,3 → 1,3,2
3,2,1 → 1,2,3
1,1,5 → 1,5,1
```

2.3.2 方法一：暴力法

最简单的想法就是暴力枚举，我们找出由给定数组的元素形成的列表的每个可能的排列，并找出比给定的排列更大的排列。

但是这个方法要求我们找出所有可能的排列，这需要很长时间，实施起来也很复杂。因此，这种算法不能满足要求。 我们跳过它的实现，直接采用正确的方法。

复杂度分析

时间复杂度：O(n!)，可能的排列总计有 n! 个。

空间复杂度：O(n)，因为数组将用于存储排列。

2.3.3 方法二：一遍扫描

首先，我们观察到对于任何给定序列的**降序排列**，就不会有下一个更大的排列。

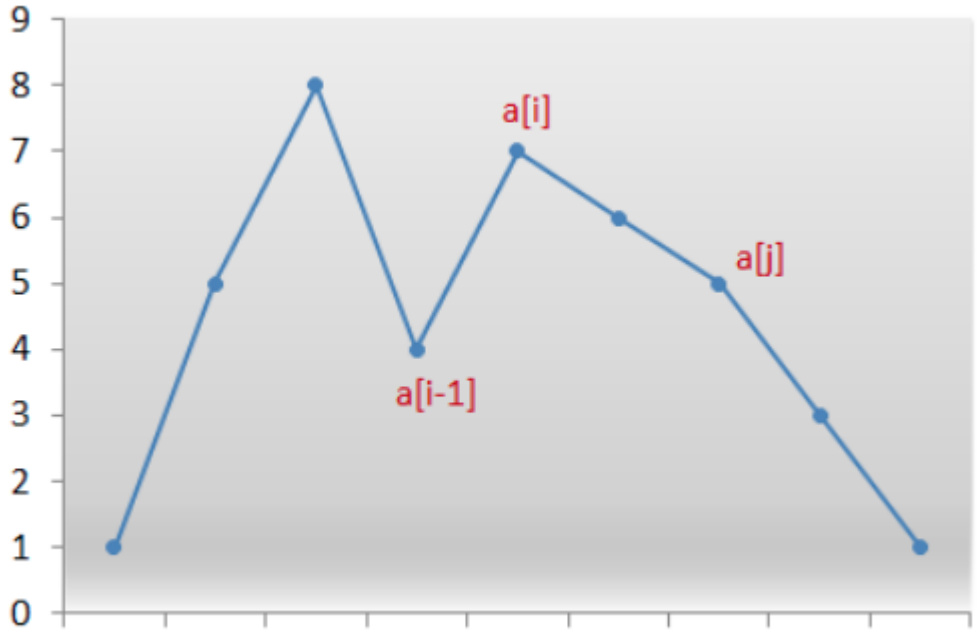
例如，以下数组不可能有下一个排列：

```
[9, 5, 4, 3, 1]
```

这时应该直接返回升序排列。

所以对于一般的情况，如果有一个“升序子序列”，那么就一定可以找到它的下一个排列。具体来说，需要从右边找到第一对两个连续的数字 a[i] 和 a[i-1]，它们满足 a[i]>a[i-1]。

所以一个思路是，找到最后一个的“正序”排列的子序列，把它改成下一个排列就行了。



不过具体操作会发现，如果正序子序列后没数了，那么子序列的“下一个”一定就是整个序列的“下一个”，这样做没问题；但如果后面还有逆序排列的数，这样就不对了。比如

```
[1,3,8,7,6,2]
```

最后的正序子序列是[1,3,8]，但显然不能直接换成[1,8,3]就完事了；而是应该考虑把 3 换成后面比 3 大、但比 8 小的数，而且要选最小的那个（6）。接下来，还要让 6 之后的所有数，做一个升序排列，得到结果：

```
[1,6,2,3,7,8]
```

代码实现如下：

```
public void nextPermutation(int[] nums) {
    int k = nums.length - 2;
    while( k >= 0 && nums[k] >= nums[k+1] )
        k--;
    // 如果全部降序，以最小序列输出
    if( k < 0 ){
        Arrays.sort(nums);
        return;
    }

    int i = k + 2;
    while( i < nums.length && nums[i] > nums[k] )
        i++;

    // 交换 nums[k]和找到的 nums[i-1]
    int temp = nums[k];
    nums[k] = nums[i-1];
    nums[i-1] = temp;

    // k 以后剩余的部分反转成升序
    int start = k + 1, end = nums.length - 1;
    while( start < end ){
        int reTemp = nums[start];
        nums[start] = nums[end];
        nums[end] = reTemp;
        start++;
```

```
        end--;
    }
}
```

复杂度分析

时间复杂度：O(N)，其中 NN 为给定序列的长度。我们至多只需要扫描两次序列，以及进行一次反转操作。

空间复杂度：O(1)，只需要常数的空间存放若干变量。

2.4 旋转图像（#48）

2.4.1 题目说明

给定一个 n × n 的二维矩阵表示一个图像。

将图像顺时针旋转 90 度。

说明：

你必须在原地旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要使用另一个矩阵来旋转图像。

```
示例 1:
给定 matrix =
[
  [1,2,3],
  [4,5,6],
  [7,8,9]
],
原地旋转输入矩阵，使其变为:
[
  [7,4,1],
  [8,5,2],
  [9,6,3]
```

```
]
示例 2:
给定 matrix =
[
  [ 5, 1, 9,11],
  [ 2, 4, 8,10],
  [13, 3, 6, 7],
  [15,14,12,16]
],
原地旋转输入矩阵，使其变为:
[
  [15,13, 2, 5],
  [14, 3, 4, 1],
  [12, 6, 8, 9],
  [16, 7,10,11]
]
```

2.4.2 分析

旋转图像，这个应用在图片处理的过程中，非常常见。我们知道对于计算机而言，图像，其实就是一组像素点的集合（所谓点阵），所以图像旋转的问题，本质上就是一个二维数组的旋转问题。

2.4.3 方法一：数学方法（转置再翻转）

我们可以利用矩阵的特性。所谓顺时针旋转，其实就是先转置矩阵，然后翻转每一行。

代码如下：

```
public void rotate(int[][] matrix) {
    int n = matrix.length;

    // 转置矩阵
    for (int i = 0; i < n; i++)
        for (int j = i; j < n; j++) {
            int tmp = matrix[i][j];
            matrix[i][j] = matrix[j][i];
            matrix[j][i] = tmp;
        }
    // 翻转行
    for( int i = 0; i < n; i++ ){
        for( int j = 0; j < n/2; j++ ){
            int tmp = matrix[i][j];
            matrix[i][j] = matrix[i][n-j-1];
            matrix[i][n-j-1] = tmp;
        }
    }
}
```

复杂度分析

- 时间复杂度：O(N^2)

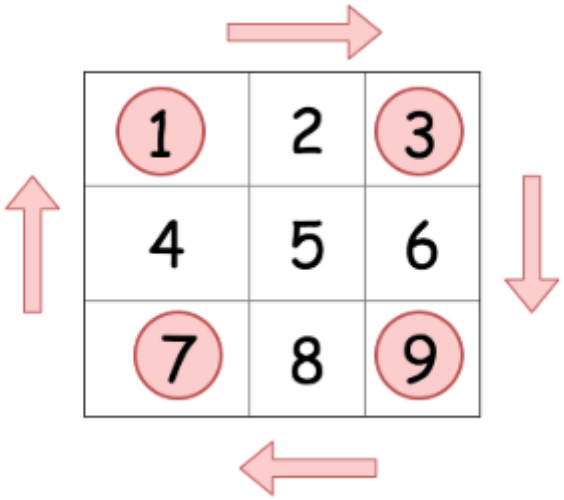
这个简单的方法已经能达到最优的时间复杂度 O(N^2) ，因为既然是旋转，那么每个点都应该遍历到，N^2 的复杂度不可避免。

- 空间复杂度：O(1)。旋转操作是原地完成的，只耗费常数空间。

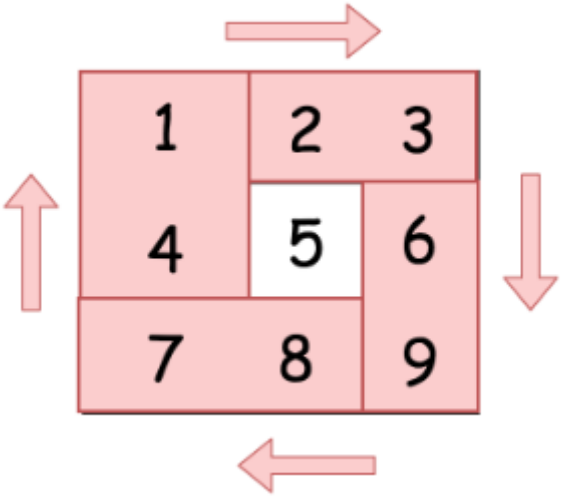
2.4.4 方法二：分治（分为四部分旋转）

方法 1 使用了两次矩阵操作，能不能只使用一次操作的方法完成旋转呢？

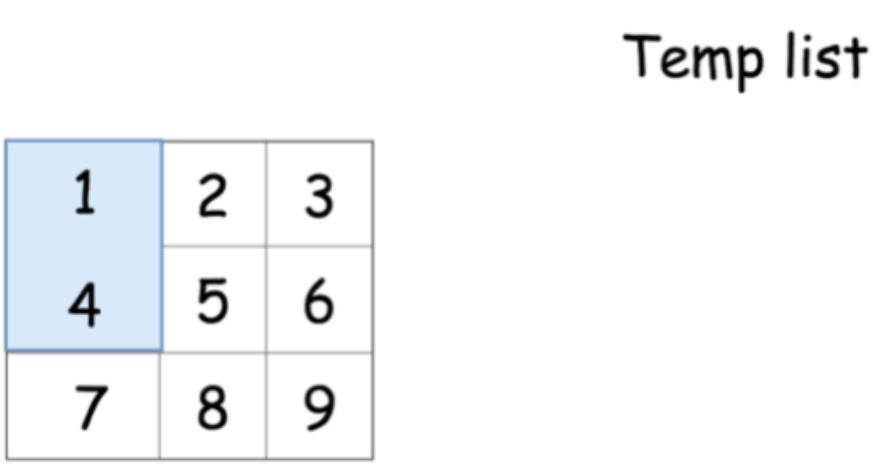
为了实现这一点，我们来研究每个元素在旋转的过程中如何移动。



这提供给我们了一个思路，可以将给定的矩阵分成四个矩形并且将原问题划归为旋转这些矩形的问题。这其实就是分治的思想。



具体解法也很直接，可以在每一个矩形中遍历元素，并且在长度为 4 的临时列表中移动它们。



代码如下：

```
public void rotate(int[][] matrix) {
    int n = matrix.length;

    for (int i = 0; i < n / 2 + n % 2; i++) {
        for (int j = 0; j < n / 2; j++) {
            int[] tmp = new int[4];
            int row = i;
            int col = j;
            for (int k = 0; k < 4; k++) {
                tmp[k] = matrix[row][col];
                // 定位下一个数
                int x = row;
                row = col;
                col = n - 1 - x;
            }

            for (int k = 0; k < 4; k++) {
                matrix[row][col] = tmp[(k + 3) % 4];
```

```
                int x = row;
                row = col;
                col = n - 1 - x;
            }
        }
    }
}
```

复杂度分析

- 时间复杂度：O(N^2) 是两重循环的复杂度。
- 空间复杂度：O(1) 由于我们在一次循环中的操作是“就地”完成的，并且我们只用了长度为 4 的临时列表做辅助。

2.4.5 方法三：分治法改进（单次循环内完成旋转）

大家可能也发现了，我们其实没有必要分成 4 个矩阵来旋转。这四个矩阵的对应关系，其实是一目了然的，我们完全可以在一次循环内，把所有元素都旋转到位。

因为旋转的时候，是上下、左右分别对称的，所以我们遍历元素的时候，只要遍历一半行、一半列就可以了（1/4 元素）。

代码如下：

```
public void rotate(int[][] matrix) {
    int n = matrix.length;

    // 不区分子矩阵，直接遍历每一个元素
    for( int i = 0; i < (n + 1)/2; i++ ){
        for( int j = 0; j < n/2; j++ ){
            int temp = matrix[i][j];
```

```
            matrix[i][j] = matrix[n-j-1][i];
            matrix[n-j-1][i] = matrix[n-i-1][n-j-1];
            matrix[n-i-1][n-j-1] = matrix[j][n-i-1];
            matrix[j][n-i-1] = temp;
        }
    }
}
```

复杂度分析

- 时间复杂度：O(N^2)，是两重循环的复杂度。
- 空间复杂度：O(1)。我们在一次循环中的操作是“就地”完成的。

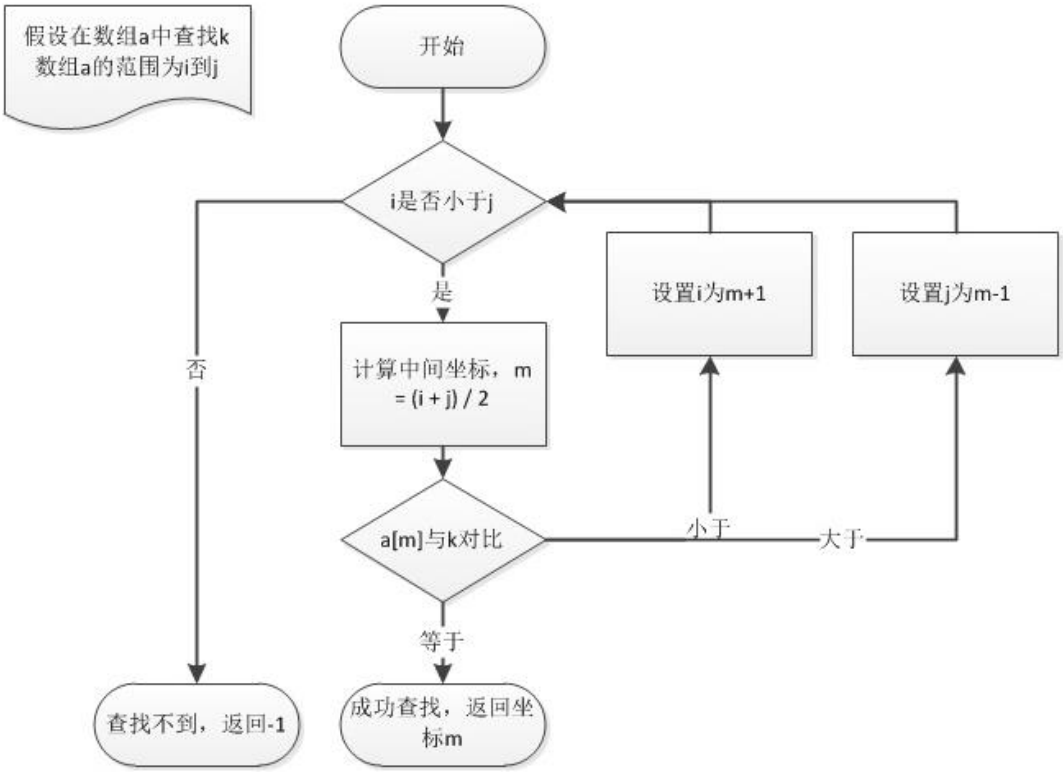
第三章 二分查找相关问题讲解

3.1 二分查找

二分查找也称折半查找（Binary Search），它是一种效率较高的查找方法，前提是数据结构必须先**排好序**，可以在**对数**时间复杂度内完成查找。

二分查找事实上采用的就是一种分治策略，它充分利用了元素间的次序关系，可在最坏的情况下用 O(log n)完成搜索任务。

它的基本思想是：假设数组元素呈升序排列，将 n 个元素分成个数大致相同的两半，取 a[n/2]与欲查找的 x 作比较，如果 x=a[n/2]则找到 x，算法终止；如 果 x<a[n/2]，则我们只要在数组 a 的左半部继续搜索 x；如果 x>a[n/2]，则我们只要在数组 a 的右 半部继续搜索 x。



二分查找问题也是面试中经常考到的问题，虽然它的思想很简单，但写好二分查找算法并不是一件容易的事情。

接下来，我们首先用代码实现一个对 int 数组的二分查找。

```
public class BinarySearch {
    public static int binarySearch(int[] a, int key){
        int low = 0;
        int high = a.length - 1;
        if ( key < a[low] || key > a[high] )
            return -1;

        while ( low <= high){
            int mid = ( low + high ) / 2;
            if( a[mid] < key)
                low = mid + 1;
            else if( a[mid] > key )
                high = mid - 1;
            else
                return mid;    // 查找成功
        }
        // 未能找到
        return -1;
    }
}
```

当然，我们也可以用递归的方式实现：

```
public static int binarySearch(int[] a, int key, int fromIndex, int toIndex){
    if ( key < a[fromIndex] || key > a[toIndex] || fromIndex > toIndex)
        return -1;
```

```
    int mid = ( fromIndex + toIndex ) / 2;


    if ( a[mid] < key )
        return binarySearch(a, key, mid + 1, toIndex);
    else if ( a[mid] > key )
        return binarySearch(a, key, fromIndex, mid - 1);
    else
        return mid;
}
```

我们总结一下二分查找：

- 优点是比较次数少，查找速度快，平均性能好；
- 缺点是要求待查表为有序表，且插入删除困难。

因此，二分查找方法适用于不经常变动而查找频繁的有序列表。使用条件：查找序列是顺序结构，有序。

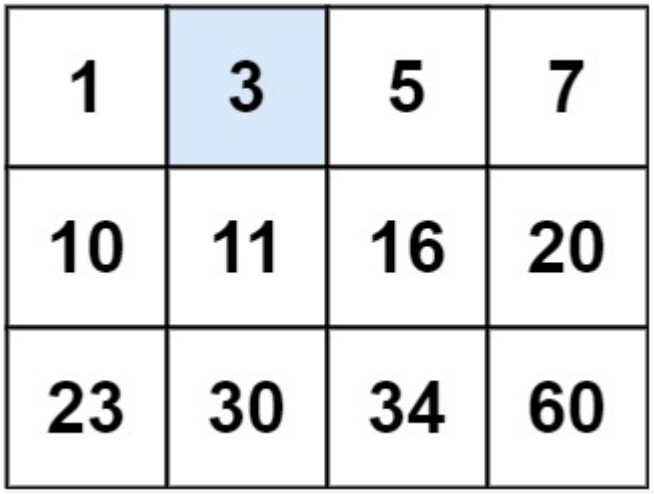
3.2 搜索二维矩阵（#74）

3.2.1 题目说明

编写一个高效的算法来判断 m x n 矩阵中，是否存在一个目标值。该矩阵具有如下特性：

- 每行中的整数从左到右按升序排列。
- 每行的第一个整数大于前一行的最后一个整数。

示例 1：




```
输入：matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,50]], target = 3
输出：true
```

示例 2：

```
输入：matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,50]], target = 13
输出：false
```

示例 3：

```
输入：matrix = [], target = 0
输出：false
```

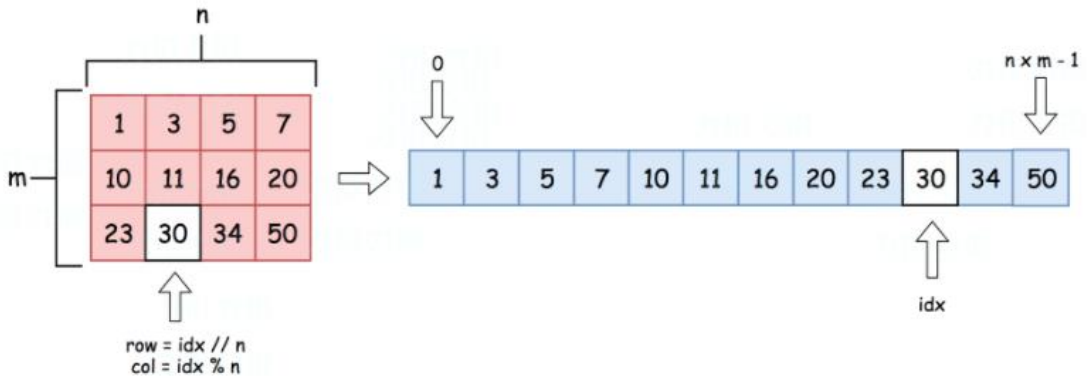
提示：

- m == matrix.length
- n == matrix[i].length
- 0 <= m, n <= 100
- -104 <= matrix[i][j], target <= 104

3.2.2 分析

既然这是一个查找元素的问题，并且数组已经排好序，我们自然可以想到用二分查找是一个高效的查找方式。

输入的 m x n 矩阵可以视为长度为 m x n 的有序数组：



行列坐标为(row, col)的元素，展开之后索引下标为 idx = row * n + col；反过来，对于一维下标为 idx 的元素，对应二维数组中的坐标就应该是：

row = idx / n; col = idx % n;

3.2.3 实现：二分查找

代码实现如下：

```
public class SearchMatrix {

    public boolean searchMatrix(int[][] matrix, int target){

        int m = matrix.length;

        if (m == 0) return false;

        int n = matrix[0].length;

        int left = 0;

        int right = m * n - 1;


        // 二分查找，定义左右指针

        while ( left <= right ){

            int midIdx = (left + right) / 2;

            int midElement = matrix[midIdx/n][midIdx%n];

            if ( midElement < target )
```

```
                left = midIdx + 1;
            else if ( midElement > target )
                right = midIdx - 1;
            else
                return true;    // 找到target
        }

        return false;
    }
}
```

复杂度分析

- 时间复杂度：由于是标准的二分查找，时间复杂度为 O(log(m n))。
- 空间复杂度：没有用到额外的空间，复杂度为 O(1)。

3.3 寻找重复数（#287）

3.3.1 题目说明

给定一个包含 n + 1 个整数的数组 nums，其数字都在 1 到 n 之间（包括 1 和 n），可知至少存在一个重复的整数。假设只有一个重复的整数，找出这个重复的数。

示例 1:

```
输入: [1,3,4,2,2]
输出: 2
```

示例 2:

```
输入: [3,1,3,4,2]
输出: 3
```

说明:

- 不能更改原数组（假设数组是只读的）。

- 只能使用额外的 O(1) 的空间。
- 时间复杂度小于 O(n2) 。
- 数组中只有一个重复的数字，但它可能不止重复出现一次。

3.3.2 分析

怎样证明 nums 中存在至少一个重复值？其实很简单，这是“抽屉原理”（或者叫“鸽子洞原理”）的简单应用。

这里，nums 中的每个数字（n+1 个）都是一个物品，nums 中可以出现的每个不同的数字（n 个）都是一个 “抽屉”。把 n+1 个物品放入 n 个抽屉中，必然至少会有一个抽屉放了 2 个或者 2 个以上的物品。所以这意味着 nums 中至少有一个数是重复的。

3.3.3 方法一：保存元素法（存入 HashMap）

首先我们想到，最简单的办法就是，遍历整个数组，挨个统计每个数字出现的次数。

用一个 HashMap 保存每个数字对应的 count 数量，就可以直观地判断出是否重复了。

代码如下：

```
public int findDuplicate1(int[] nums){
    Map<Integer, Integer> countMap = new HashMap<>();
    for( Integer num : nums ){
        if( countMap.get(num) == null )
            countMap.put(num, 1);
        else
            return num;
    }
    return -1;
}
```

3.3.4 方法二：保存元素法改进（存入 Set）

当然我们应该还能想到，其实没必要用 HashMap，直接保存到一个 Set 里，就知道这个元素到底有没有了。

```
public int findDuplicate2(int[] nums){
    Set<Integer> set = new HashSet<>();
    for( Integer num : nums ){
        if( set.contains(num) )
            return num;
        else
            set.add(num);
    }
    return -1;
}
```

复杂度分析

时间复杂度：O(n)，我们只对数组做了一次遍历，在 HashMap 和 HashSet 中查找的复杂度是 O(1)。

空间复杂度：O(n)，我们需要一个 HashMap 或者 HashSet 来做额外存储，最坏情况下，这需要线性的存储空间。

尽管时间复杂度较小，但以上两种保存元素的方法，都用到了额外的存储空间，这个空间复杂度不能让我们满意。

3.3.5 方法三：二分查找

这道题目中数组其实是很特殊的，我们可以从原始的 [1, N] 的自然数序列开始想。现在增加到了 N+1 个数，根据抽屉原理，肯定会有重复数。对于增加重复数的方式，整体应该有两种可能：

- 如果重复数（比如叫做 target）只出现两次，那么其实就是 1~N 所有数都出现了一次，然后再加一个 target；
- 如果重复数 target 出现多次，那在情况 1 的基础上，它每多出现一次，就会导致 1~N 中的其它数少一个。

例如：1~9 之间的 10 个数的数组，重复数是 6：

```
1，2，5，6，6，6，6，6，7，9
```

本来最简单（重复数出现两次，其它 1~9 的数都出现一次）的是

```
1，2，3，4，5，6，6，7，8，9
```

现在没有 3、4 和 8，所以 6 会多出现 3 次。

我们可以发现一个规律：

- 以 target 为界，对于比 target 小的数 i，数组中所有小于等于它的数，最多出现一次（有可能被多出现的 target 占用了），所以总个数不会超过 i。
- 对于比 target 大的数 j，如果每个元素都只出现一次，那么所有小于等于它的元素是 j 个；而现在 target 会重复出现，所以总数一定会大于 j。

用数学化的语言描述就是：

我们把对于 1~N 内的某个数 i，在数组中小于等于它的所有元素的个数，记为 count[i]。

则：当 i 属于[1, target-1]范围内，count[i] <= i；当 i 属于[target, N]范围内，count[i] > i。

所以要找 target，其实就是要找 1~N 中这个分界的数。所以我们可以对 1~N 的 N 个自然数进行二分查找，它们可以看作一个排好序的数组，但不占用额外的空间。

代码实现如下：

```
public int findDuplicate3(int[] nums){
    int l = 1;
    int r = nums.length - 1;
    // 二分查找
    while (l <= r){
        int i = (l + r) / 2;
        // 对当前 i 计算 count[i]
        int count = 0;
        for( int j = 0; j < nums.length; j++ ){
            if (nums[j] <= i)
                count ++;
        }
        // 判断 count[i] 和 i 的大小关系
        if ( count <= i )
            l = i + 1;
        else
            r = i;
        // 找到 target
        if (l == r)
            return l;
    }
    return -1;
}
```

复杂度分析

- 时间复杂度：O(nlog n)，其中 n 为 nums[] 数组的长度。二分查找最多需要 O(logn) 次，而每次判断 count 的时候需要 O(n) 遍历 nums[] 数组求解小于等于 i 的数的个数，因此总时间复杂度为 O(nlogn)。
- 空间复杂度：O(1)。我们只需要常数空间存放若干变量。

3.3.6 方法四：排序法

另一个想法是，我们可以先在原数组上排序。

排序之后，所有重复的数会排在一起；这样，只要我们遍历的时候发现连续两个元素相等，就可以输出结果了。

代码如下：

```
public int findDuplicate3(int[] nums){
    Arrays.sort(nums);
    for( int i = 1; i < nums.length; i++ ){
        if( nums[i] == nums[i-1] )
            return nums[i];
    }
    return -1;
}
```

复杂度分析

- 时间复杂度： O(nlgn)。对数组排序，在 Java 中要花费 O(nlgn) 时间，后续是一个线性扫描，所以总的时间复杂度是 O(nlgn)。
- 空间复杂度： O(1) (or O(n))，在这里，我们对 nums 进行了排序，因此内存大小是固定的。当然，这里的前提是我们可以用常数的空间，在原数组上直接排序。如果我们不能修改输入数组，那么我们必须把 nums 拷贝出来，并进行排序，这需要分配线性的额外空间。

3.3.7 方法五：快慢指针法（循环检测）

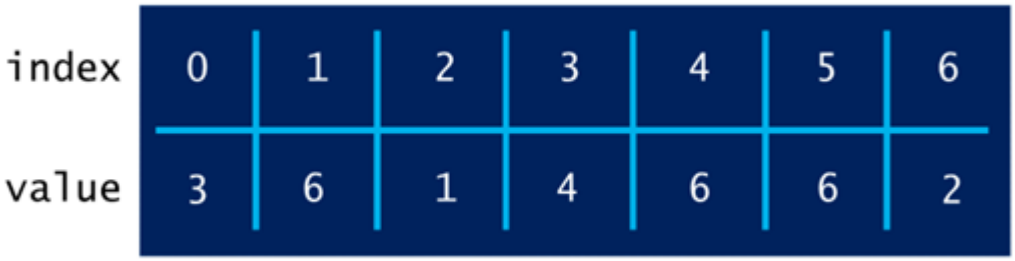
这是一种比较特殊的思路。把 nums 看成是顺序存储的链表，nums 中每个元素的值是下一个链表节点的地址。

那么如果 nums 有重复值，说明链表**存在环**，本问题就转化为了找链表中环的入口节点，因此可以用**快慢指针**解决。

比如数组

```
[3，6，1，4，6，6，2]
```

保存为：



整体思路如下：

- 第一阶段，寻找环中的节点
  a) 初始时，都指向链表第一个节点 nums[0]；
  b) 慢指针每次走一步，快指针走两步；
  c) 如果有环，那么快指针一定会再次追上慢指针；相遇时，相遇节点必在环中
- 第二阶段，寻找环的入口节点（重复的地址值）
  d) 重新定义两个指针，让 before，after 分别指向链表开始节点，相遇节点
  e) before 与 after 相遇时，相遇点就是环的入口节点

第二次相遇时，应该有：

```
慢指针总路程 = 环外 0 到入口 + 环内入口到相遇点 (可能还有 + 环内 m 圈)
快指针总路程 = 环外 0 到入口 + 环内入口到相遇点 + 环内 n 圈
```

并且，快指针总路程是慢指针的 2 倍。所以：

环内 n-m 圈 = 环外 0 到入口 + 环内入口到相遇点。

把环内项移到同一边，就有：

环内相遇点到入口 + 环内 n-m-1 圈 = 环外 0 到入口

这就很清楚了：从环外 0 开始，和从相遇点开始，走同样多的步数之后，一定可以在入口处相遇。所以第二阶段的相遇点，就是环的入口，也就是重复的元素。

代码如下：

```
public int findDuplicate(int[] nums){
    // 定义快慢指针
    int fast = 0, low = 0;
    // 第一阶段：寻找链表中的环
    do {
        // 快指针一次走两步，慢指针一次走一步
        low = nums[low];
        fast = nums[nums[fast]];
    } while( fast != low);

    // 第二阶段：寻找环在链上的入口节点
    int ptr1 = 0, ptr2 = low;
    while( ptr1 != ptr2 ){
        ptr1 = nums[ptr1];
        ptr2 = nums[ptr2];
    }
    return ptr1;
}
```

复杂度分析

- 时间复杂度：O(n)，不管是寻找环上的相遇点，还是环的入口，访问次数都不会超过数组长度。
- 空间复杂度：O(1)，我们只需要定义几个指针就可以了。

通过快慢指针循环检测这样的巧妙方法，实现了在不额外使用内存空间的前提下，满足线性时间复杂度 O(n)。

第四章 字符串问题讲解

字符串（String）是由零个或多个字符组成的有限序列，它是编程语言中表示文本的数据类型。

字符串与数组有很多相似之处，比如可以使用索引（下标）来得到一个字符。字符串，一般可以认为就是一个字符数组（char array）。不过字符串有其鲜明的特点，它的结构相对简单，但规模可能是非常庞大的。

在编程语言中，字符串往往由特定字符集内有限的字符组合而成。在 Java 中字符串属于对象，Java 提供了 String 类来创建和操作字符串。

4.1 字符串相加（#415）

4.1.1 题目说明

给定两个字符串形式的非负整数 num1 和 num2，计算它们的和。

提示：

1. num1 和 num2 的长度都小于 5100
2. num1 和 num2 都只包含数字 0-9
3. num1 和 num2 都不包含任何前导零
4. 你不能使用任何内建 BigInteger 库， 也不能直接将输入的字符串转换为整数形式

4.1.2 分析

这里不允许直接将输入字符串转为整数，那自然想到应该把字符串按每个字符 char 一一拆开，相当于遍历整数上的每一个数位，然后通过“乘 10 叠加”的方式，就可以整合起来了。这相当于算术中的“竖式加法”。

另外题目要求不能使用 BigInteger 的内建库，这其实就是让我们自己实现一个大整数相

加的功能。

4.1.3 代码实现

```
public class AddStrings {

    public String addStrings(String num1, String num2) {

        StringBuffer result = new StringBuffer();

        int i = num1.length() - 1;

        int j = num2.length() - 1;

        int carry = 0;

        while ( i >= 0 || j >= 0 || carry != 0 ){

            int x = i >= 0 ? num1.charAt(i) - '0' : 0;

            int y = j >= 0 ? num2.charAt(j) - '0' : 0;

            int sum = x + y + carry;

            result.append(sum % 10);

            carry = sum / 10;


            i--;

            j--;

        }

        return result.reverse().toString();

    }

}
```

4.1.4 复杂度分析

时间复杂度：O(max(len1,len2))，其中 len1 =num1.length，len2 =num2.length。竖式加法的次数取决于较大数的位数。

空间复杂度：O(n)。解法中使用到了 StringBuffer，所以空间复杂度为 O(n)。

4.2 字符串相乘（#43）

4.2.1 题目说明

给定两个以字符串形式表示的非负整数 num1 和 num2，返回 num1 和 num2 的乘积，它们的乘积也表示为字符串形式。

示例 1:

```
输入: num1 = "2", num2 = "3"
输出: "6"
```

示例 2:

```
输入: num1 = "123", num2 = "456"
输出: "56088"
```

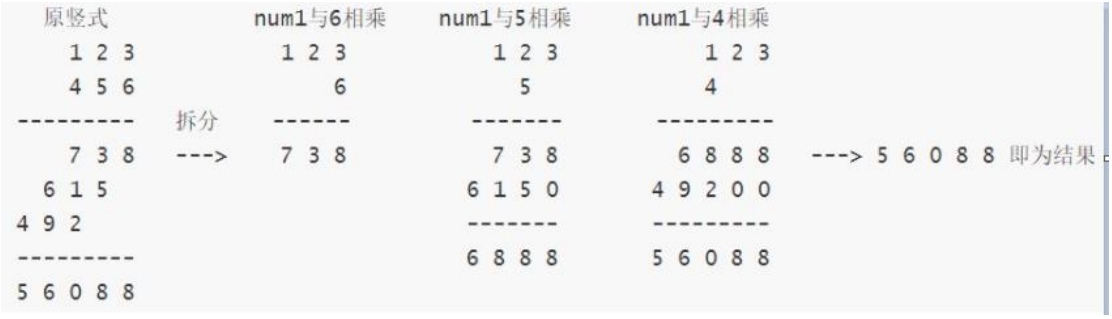
说明：

1. num1 和 num2 的长度小于 110。
2. num1 和 num2 只包含数字 0-9。
3. num1 和 num2 均不以零开头，除非是数字 0 本身。
4. 不能使用任何标准库的大数类型（比如 BigInteger）或直接将输入转换为整数来处理。

4.2.2 分析

跟“字符串相加”类似，这里我们要处理的，也是大整数的相乘问题。

思路也可以非常类似：我们借鉴数学中“竖式乘法”的规则，用 num1 分别去乘 num2 的每一位数字，最后再用 AddStrings 将乘出的结果全部叠加起来就可以了。



4.2.3 具体代码实现

```
public class MultiplyStrings {

    public String multiply(String num1, String num2) {

        if ( num1.equals("0") || num2.equals("0") ) return "0";

        String result = "0";


        // 遍历num2 的每个数位，逐个与num1 相乘

        for ( int i = num2.length() - 1; i >= 0; i-- ){

            int carry = 0;

            StringBuffer curRes = new StringBuffer();

            for ( int j = 0; j < num2.length() - 1 - i; j++ ){

                curRes.append("0");

            }


            int y = num2.charAt(i) - '0';


            // 遍历num1 的每一位数，跟y 相乘

            for ( int j = num1.length() - 1; j >= 0; j-- ){

                int x = num1.charAt(j) - '0';

                int product = x * y + carry;

                curRes.append(product % 10);

                carry = product / 10;
```

```
            }
            if (carry != 0) curRes.append(carry);
            AddStrings addStrs = new AddStrings();
            result = addStrs.addStrings(result,
curRes.reverse().toString());
        }
        return result;
    }
}
```

4.2.4 复杂度分析

时间复杂度：O(mn+n^2)，其中 m 和 n 分别是 num1 和 num2 的长度。

做计算的时候，外层需要从右往左遍历 num2，而对于 num2 的每一位，都需要和 num1 的每一位计算乘积，因此计算乘积的总次数是 mn。字符串相加操作共有 n 次，每次相加的字符串长度最长为 m+n，因此字符串相加的时间复杂度是 O(mn+n^2)。总时间复杂度是 O(mn+n^2)。

空间复杂度：O(m+n)。空间复杂度取决于存储中间状态的字符串，由于乘积的最大长度为 m+n，因此存储中间状态的字符串的长度不会超过 m+n。

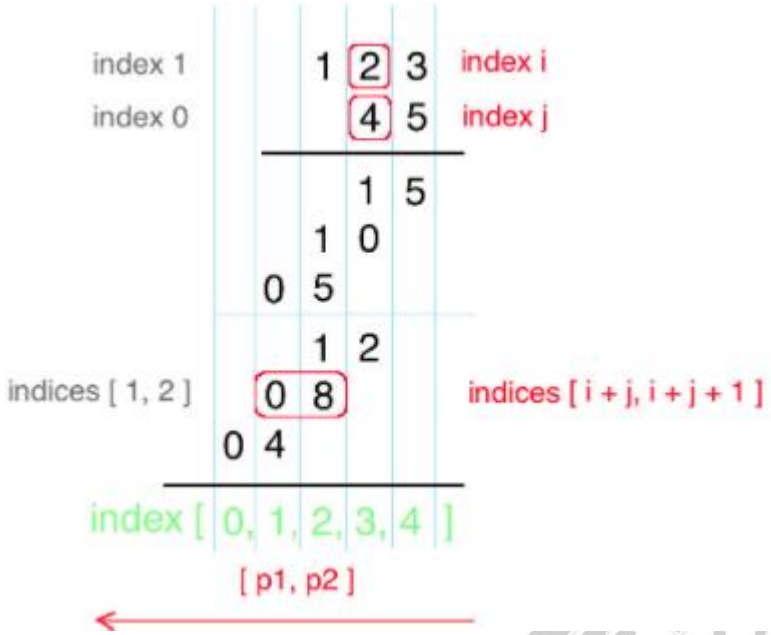
4.2.5 算法优化

我们看到计算过程中，用到了太多的字符串相加操作，调用 addStrings 方法时又需要遍历字符串的每一位，这个过程显得有些繁琐。能不能用其它的方法进行简化呢？

我们发现，m 位数乘以 n 位数，结果最多就是 m+n 位；所以我们可以用一个 m+n 长度的数组来保存计算结果。

而且，某两个数位相乘，num1[i] x num2[j] 的结果(定义为两位数，一位数的话前面补

0)，其第一位位于 result[i+j]，第二位位于 result[i+j+1]。



根据上面的思路，我们可以遍历 num1 和 num2 中的每一位数，相乘后叠加到 result 的对应位上就可以了。

代码实现如下：

```
public String multiply(String num1, String num2) {
    if ( num1.equals("0") || num2.equals("0") ) return "0";
    int[] resultArr = new int[num1.length() + num2.length()];

    // 遍历 num1 和 num2，逐位计算乘积，填入结果数组中
    for ( int i = num1.length() - 1; i >= 0; i-- ){
        int x = num1.charAt(i) - '0';
        for ( int j = num2.length() - 1; j >= 0; j-- ){
            int y = num2.charAt(j) - '0';
            int product = x * y;
            int temp = resultArr[i+j+1] + product;
```

```
            resultArr[i+j+1] = temp % 10;
            resultArr[i+j] += temp / 10;
        }
    }
    StringBuffer result = new StringBuffer();
    int start = resultArr[0] == 0 ? 1 : 0;
    for ( int i = start; i < resultArr.length; i++ ){
        result.append(resultArr[i]);
    }
    return result.toString();
}
```

复杂度分析

时间复杂度：O(mn)，其中 m 和 n 分别是 num1 和 num2 的长度。需要计算 num1 的每一位和 num2 的每一位的乘积。

空间复杂度：O(m+n)，需要创建一个长度为 m+n 的数组存储乘积。

4.3 去除重复字母（#316）

4.3.1 题目说明

给你一个字符串 s ，请你去除字符串中重复的字母，使得每个字母只出现一次。需保证 返回结果的字典序最小（要求不能打乱其他字符的相对位置）。

示例 1：

```
输入：s = "bcabc"
输出："abc"
```

示例 2：

```
输入：s = "cbacdcbc"
```

输出："acdb"

提示：

- 1 <= s.length <= 104
- s 由小写英文字母组成

4.3.2 分析

首先要知道什么叫 “字典序”。

字符串之间比较跟数字之间比较是不太一样的：字符串比较，是从头往后一个字符一个字符比较的，哪个字符串大取决于两个字符串中**第一个对应不相等**的字符。

所以，任意一个以 a 开头的字符串都大于任意一个以 b 开头的字符串。

为了得到最小字典序的结果，解题过程中，我们可以将最小的字符尽可能的放在前面，把前面出现的重复字母全部删除。这其实就是一个**贪心策略**。

4.3.3 方法一：贪心策略（逐个字符处理）

这种想法就是典型的贪心策略了：我们每次都找到当前能够移到最左边的、最小的字母。这就是我们得到结果的第一个字母，它之前的所有重复字母会被删掉；然后我们从它以后的字符串中，使用相同的逻辑，继续寻找第二个最小的字母。

所以，我们在代码实现上，可以使用递归。

代码如下：

```
public class RemoveDuplicateLetters {
    public String removeDuplicateLetters(String s) {
        if (s.length() == 0) return "";
        int position = 0;
        for (int i = 0; i < s.length(); i++){
            if (s.charAt(i) < s.charAt(position)){
                boolean isReplaceable = true;
                for (int j = position; j < i; j++){
```

```
                    boolean isDuplicated = false;
                    for (int k = i; k < s.length(); k++){
                        if (s.charAt(k) == s.charAt(j)){
                            isDuplicated = true;
                            break;
                        }
                    }
                    isReplaceable = isReplaceable && isDuplicated;
                }

                if (isReplaceable){
                    position = i;
                }
            }
        }
        return s.charAt(position)
                    + removeDuplicateLetters0(
                          s.substring(position+1)
                          .replaceAll("" + s.charAt(position), ""));
    }
}
```

复杂度分析

- 时间复杂度：O(N^3)，因为用到了三重循环，最坏情况下时间复杂度达到了 N^3。（超出运行时间限制）
- 空间复杂度：O(N)，每次给字符串切片都会创建一个新的字符串（字符串不可变），切片的数量受常数限制，最终复杂度为 O(N) * C = O(N)。

4.4.4 方法二：贪心策略改进

我们发现，对于“是否重复出现”的判断，每次都要偏离当前字母之后的所有字符，这显然做了很多重复工作。

优化的方法，我们可以用一个 count 数组，保存所有 26 个字母在 s 中出现的频次。当我们遍历字符串时，每遇到一个字母，就让它对应的 count 减一；当当前字母对应的 count 减为 0 时，说明之后不会再重复出现了，因此即使有更小的字母也不能替代它，我们直接就可以把它作为最左侧字母输出了。

代码实现：

```
public String removeDuplicateLetters(String s) {
        if (s.length() == 0) return "";
        int position = 0;
        // 定义一个count 数组，用来保存26 个字母在s 中出现的频次
        int[] count = new int[26];
        for (int i = 0; i < s.length(); i++){
            count[s.charAt(i) - 'a'] ++;
        }
        for (int i = 0; i < s.length(); i++){
            if (s.charAt(i) < s.charAt(position)) {
                position = i;
            }
            if (--count[s.charAt(i) - 'a'] == 0){
                break;
            }
        }
        return s.charAt(position) +
                removeDuplicateLetters(s.substring(position+1)
```

```
                        .replaceAll("" + s.charAt(position), ""));
}
```

复杂度分析

- 时间复杂度：O(N)。每次递归调用占用 O(N) 时间。递归调用的次数受常数限制(只有 26 个字母)，最终复杂度为 O(N) * C = O(N)。

- 空间复杂度：O(N)，每次给字符串切片都会创建一个新的字符串（字符串不可变），切片的数量受常数限制，最终复杂度为 O(N) * C = O(N)。

4.4.5 方法三：贪心策略（用栈实现）

上面的方法由于递归的时候，用到了字符串切片的方法，导致必须要有线性的空间复杂度，而且运行时间也并不短。那还没有别的优化方法呢？

这就需要结合其它的数据结构了。我们可以用栈来存储最终返回的字符串。

代码实现如下：

```
public String removeDuplicateLetters(String s) {
    Stack<Character> stack = new Stack<>();
    HashSet<Character> seen = new HashSet<>();
    HashMap<Character, Integer> last_occur = new HashMap<>();
    for (int i = 0; i < s.length(); i++){
        last_occur.put(s.charAt(i), i);
    }
    // 遍历字符串，判断是否入栈
    for (int i = 0; i < s.length(); i++){
        char c = s.charAt(i);
```

```
        if (!seen.contains(c)){
            while (!stack.isEmpty() &&
                    c < stack.peek() &&
                    last_occur.get(stack.peek()) > i){
                seen.remove(stack.pop());
            }
            seen.add(c);
            stack.push(c);
        }
    }
    // 将栈中元素保存成字符串输出
    StringBuilder sb = new StringBuilder(stack.size());
    for (Character c: stack){
        sb.append(c.charValue());
    }
    return sb.toString();
}
```

复杂度分析

时间复杂度：O(N)。虽然看起来是双重循环，但内循环的次数受栈中剩余字符总数的限制，因为栈中的元素不重复，不会超出字母表大小，因此最终复杂度仍为 O(N)。

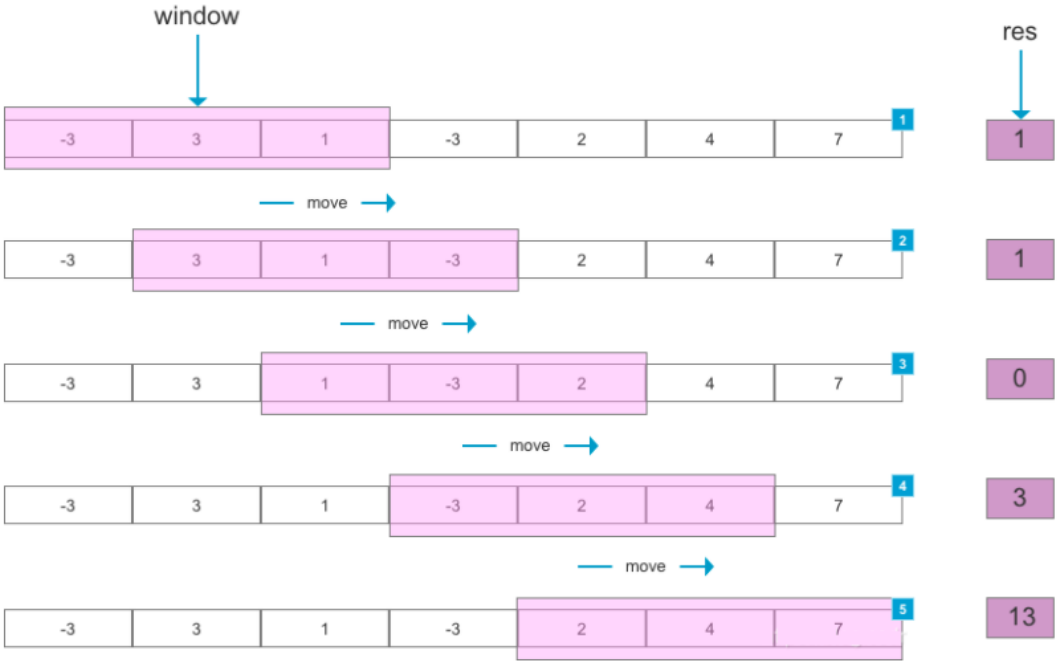
空间复杂度：O(1)。看上去空间复杂度像是 O(N)，但实际上并不是。首先，seen 中字符不重复，其大小会受字母表大小的限制，所以是 O(1)。其次，只有 stack 中不存在的元素才会被压入，因此 stack 中的元素也唯一。所以最终空间复杂度为 O(1)。

第五章 滑动窗口相关问题讲解

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，它的原理与网络传输 TCP 协议中的滑动窗口协议（Sliding Window Protocol）基本一致。

这种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。**滑动窗口主要应用在数组和字符串上。**

例如，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，可以得到一组结果 res。



因为滑动窗口是靠窗口起始、结束两个位置来表示的，所以滑动窗口也可以看作特殊的“双指针”。

5.1 滑动窗口最大值（#239）

5.1.1 题目说明

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 1:

```
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
```

示例 2：

```
输入：nums = [1], k = 1
输出：[1]
```

示例 3：

```
输入：nums = [1,-1], k = 1
输出：[1,-1]
```

示例 4：

```
输入：nums = [9,11], k = 2
输出：[11]
```

示例 5：

```
输入：nums = [4,-2], k = 2
输出：[4]
```

提示：

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

5.1.2 分析

这是一个典型的滑动窗口的问题。由于滑动窗口的大小 k 被限定在[1, nums.length]，所以我们可以直接推出窗口的个数为 nums.length - k + 1。

5.1.3 方法一：暴力法

最简单直接的方法，就是遍历每个滑动窗口，找到每个窗口的最大值。

代码如下：

```
public class SlidingWindowMaximum {
    public int[] maxSlidingWindow(int[] nums, int k) {
        int[] result = new int[nums.length - k + 1];
        // 遍历数组
        for(int i = 0; i <= nums.length - k; i++){
            int max = nums[i];
            for(int j = i; j < i + k; j++){
                if( nums[j] > max ){
                    max = nums[j];
```

```
                }
            }
            result[i] = max;
        }
        return result;
    }
}
```

复杂度分析

时间复杂度： O(Nk)，双重循环，外层遍历数组循环 N 次，内层遍历窗口循环 k 次，所以整体就是 O(N) * O(k) = O(Nk)，表现较差。在 leetcode 上提交，会发现超出了题目要求的运行时间限制。

空间复杂度：O(N-k)，输出数组用到了 N-k+1 的空间。

5.1.4 方法二：使用堆

如何优化时间复杂度呢？可以使用堆。

构建一个大顶堆（Max Heap），那么堆顶元素 heap[0] 永远是最大的。每次移动窗口的时候，我们只要维护这个堆、在里面插入删除元素，然后返回堆顶元素 heap[0]就可以了。

在代码中，我们可以用一个优先队列（Priority Queue）来实现大顶堆。

代码如下：

```
public int[] maxSlidingWindow(int[] nums, int k) {

    int[] result = new int[nums.length - k + 1];

    // 用优先队列定义一个大顶堆
```

```
    PriorityQueue<Integer> maxHeap = new PriorityQueue<>(k, new
Comparator<Integer>() {
        @Override
        public int compare(Integer o1, Integer o2) {
            return o2 - o1;
        }
    });
    for (int i = 0; i < k; i++) {
        maxHeap.add(nums[i]);
    }
    result[0] = maxHeap.peek();
    // 遍历数组
    for(int i = 1; i <= nums.length - k; i++){
        maxHeap.remove(nums[i - 1]);
        maxHeap.add(nums[i + k - 1]);
        result[i] = maxHeap.peek();
    }
    return result;
}
```

复杂度分析

时间复杂度： O(Nlog(k))，在大小为 k 的堆中插入一个元素只需要消耗 log(k) 时间，

因此这样改进后，算法的时间复杂度为 O(Nlog(k))。但提交依然会超出时间限制。

空间复杂度：O(N)，输出数组用到了 O(N-k+1)的空间，大顶堆用了 O(k)。

5.1.5 方法三：使用双向队列

使用堆看起来不错，但离题目要求还有差距。能否得到 O(N) 的算法呢？

我们发现，窗口在滑动过程中，其实数据发生的变化很小：只有第一个元素被删除、后面又新增一个元素，中间的大量元素是不变的。也就是说，前后两个窗口中，有大量数据是**重叠** 的。

[1, 3, -1,] -3, 5, 3, 6, 7

1, [3, -1, -3,] 5, 3, 6, 7

1, 3, [-1, -3, 5,] 3, 6, 7

自然想到，其实可以使用一个 **队列** 来保存窗口数据：窗口每次滑动，我们就让后面的一个元素（-3）进队，并且让第一个元素（1）出队。进出队列的操作，只要耗费常数时间。

这种场景，可以使用 **双向队列**（也叫双端队列 Dequeue），该数据结构可以从两端以常数时间压入/弹出元素。

在构建双向队列的时候，可以采用删除队尾更小元素的策略，所以，得到的其实就是一个 **从大到小排序** 的队列。

这样存储的元素，可以认为是遵循“更新更大”原则的。

具体代码如下：

```
public int[] maxSlidingWindow(int[] nums, int k) {
    if (k == 1) return nums;
```

```
    int[] result = new int[nums.length - k + 1];
    ArrayDeque<Integer> deque = new ArrayDeque<>();
    // 初始化双向队列
    for (int i = 0; i < k; i++){
        while (!deque.isEmpty() && nums[i] > nums[deque.getLast()]){
            deque.removeLast();
        }
        deque.addLast(i);
    }

    result[0] = nums[deque.getFirst()];

    // 遍历数组
    for(int i = k; i < nums.length; i++){
        if (!deque.isEmpty() && deque.getFirst() == i - k){
            deque.removeFirst();
        }
        while (!deque.isEmpty() && nums[i] > nums[deque.getLast()]){
            deque.removeLast();
        }
        deque.addLast(i);
        result[i - k + 1] = nums[deque.getFirst()];
    }

    return result;
}
```

复杂度分析

时间复杂度：O(N)，每个元素被处理两次：其索引被添加到双向队列中，以及被双向队

列删除。

空间复杂度： O(N)，输出数组使用了 O(N−k+1) 空间，双向队列使用了 O(k)。

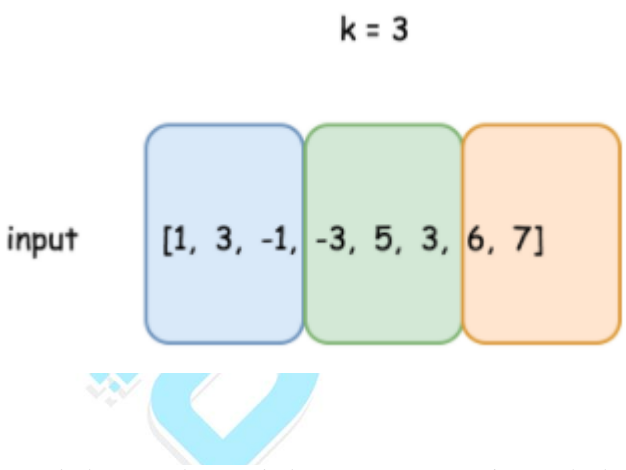
5.1.6 方法四：左右扫描

上面的算法，时间复杂度已经可以满足要求了，空间复杂度也不高；但是用到了双向队列这样的高级数据结构，具体算法也有些繁琐。

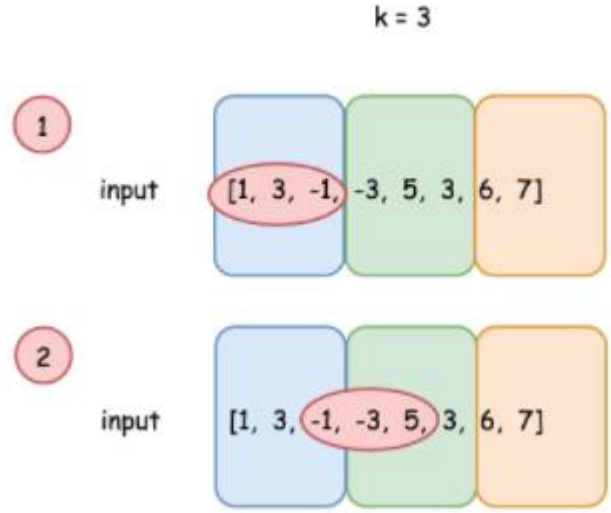
这里再介绍另一种巧妙的算法：

算法的主要思想，是将输入数组分割成有 k 个元素的块，然后分别从左右两个方向进行扫描统计块内的最大值，最后进行合并。这里有一些借鉴了分治和动态规划的思想。

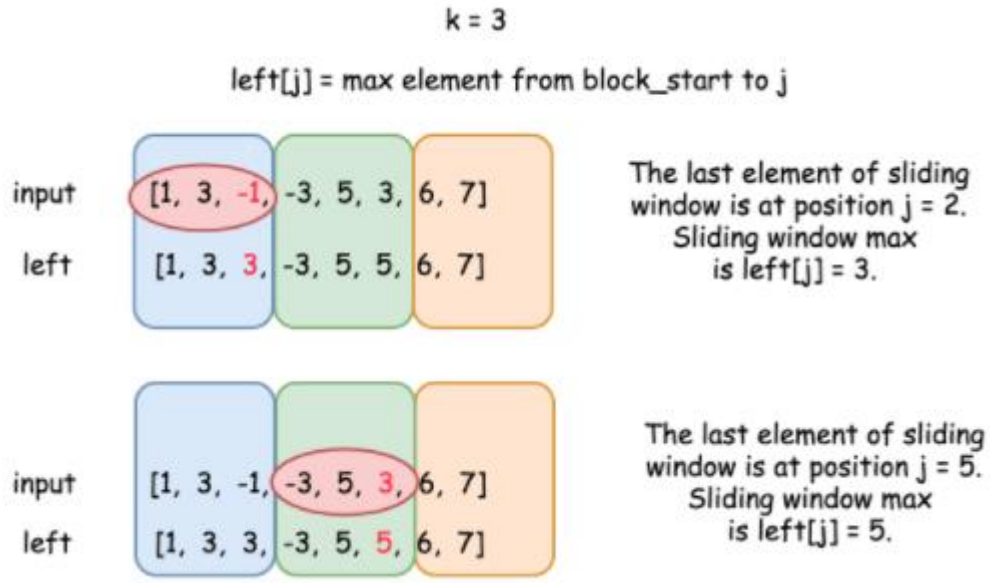
分块的时候，如果 n % k != 0，则最后一块的元素个数可能更少。



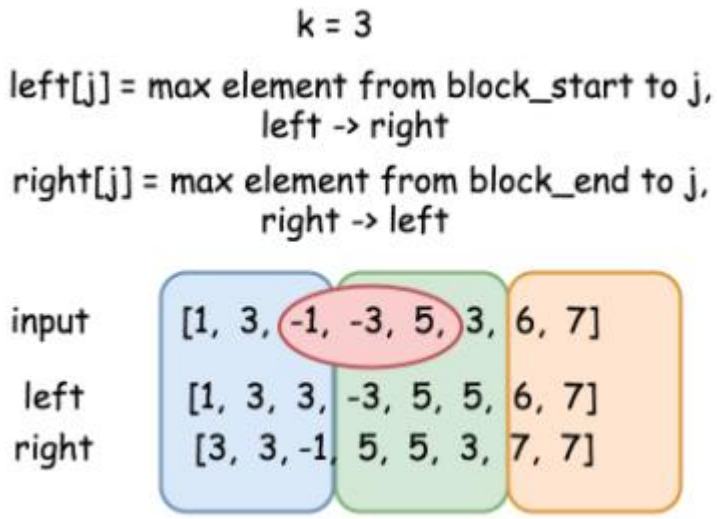
开头元素为 i ，结尾元素为 j 的当前滑动窗口可能在一个块内，也可能在两个块中。



情况 1 比较简单。 建立数组 `left`， 其中 `left[j]` 是从块的开始到下标 `j` 最大的元素，方向 左->右。

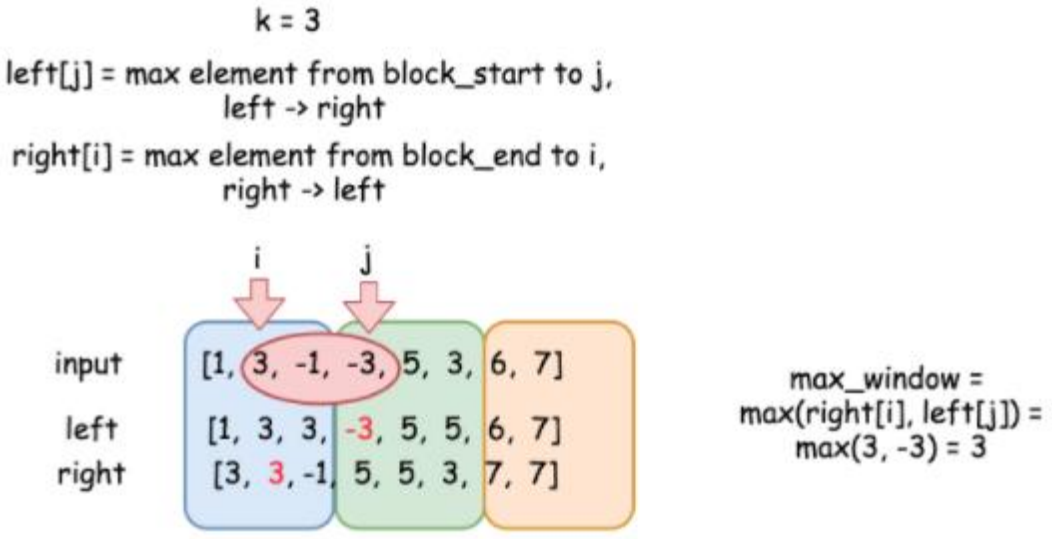


为了处理更复杂的情况 2，我们需要数组 right，其中 right[j] 是从块的结尾到下标 j 最大的元素，方向 右->左。right 数组和 left 除了方向不同以外基本一致。

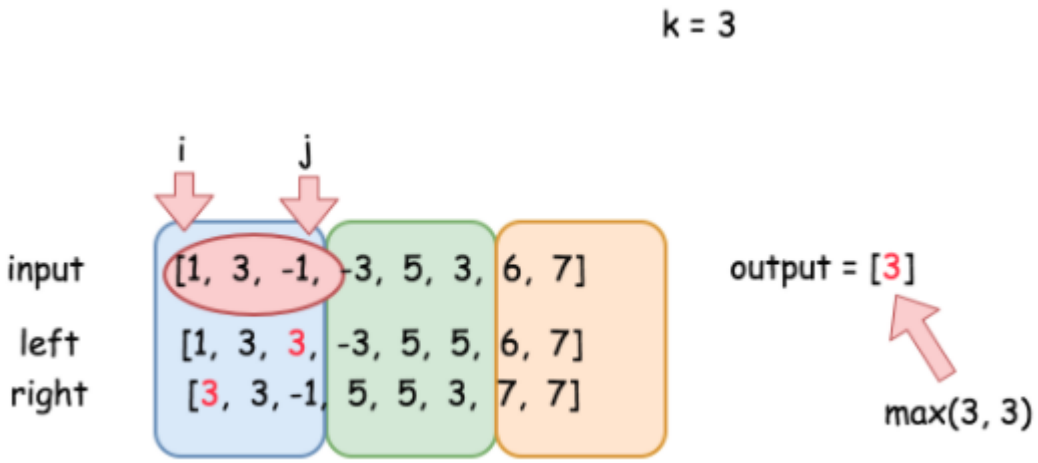


两数组合在一起，就可以提供相邻两个块内元素的全部信息。

现在我们考虑从下标 i 到下标 j 的滑动窗口。可以发现，这个窗口其实可以堪称两部分：以两块的边界（比如叫做 m）为界，从 i 到 m 属于第一个块，这部分的最大值，应该从右往左，看 right[i]；而从 m 到 j 属于第二个块，这部分的最大值应该从左往右，看 left[j]。因此合并起来，整个滑动窗口中的最大元素为 max(right[i], left[j])。



同样，如果是第一种情形，都在一个块内，用上面的公式也是正确的（这时 right[i] = left[j]，都是块内最大值）。



这个算法时间复杂度同样是 O(N)，优点是不需要使用除数组之外的任何数据结构。

代码如下：

```
public int[] maxSlidingWindow(int[] nums, int k) {
    int n = nums.length;
    int[] result = new int[n - k + 1];
    // 定义存放块内最大值的数组 left、right
    int[] left = new int[n];
    int[] right = new int[n];
    // 遍历数组，左右双扫描
    for (int i = 0; i < n; i++){
        if (i % k == 0){
            left[i] = nums[i];
        } else {
            left[i] = Math.max(left[i-1], nums[i]);
        }
        int j = n - i - 1;
        if (j % k == k - 1  || j == n-1){
```

```
            right[j] = nums[j];
        } else {
            right[j] = Math.max(right[j+1], nums[j]);
        }
    }
    for (int i = 0; i < n - k + 1; i++){
        result[i] = Math.max(right[i], left[i + k - 1]);
    }
    return result;
}
```

复杂度分析

时间复杂度：O(N)，我们对长度为 N 的数组处理了 3 次（实际代码中只有两个循环，左右扫描是对称的，我们在一次遍历中同时处理了）。因为避免了出队入队的操作，所以这个算法在实际运行中，耗费时间要明显少于之前的算法。

空间复杂度：O(N)，用于存储长度为 N 的 left 和 right 数组，以及长度为 N - k + 1 的输出数组。

5.2 最小覆盖子串（#76）

5.2.1 题目说明

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"

提示：

- 1 <= s.length, t.length <= 105
- s 和 t 由英文字母组成

进阶：你能设计一个在 o(n) 时间内解决此问题的算法吗？

5.2.2 分析

所谓“子串”，指的是字符串中连续的一部分字符集合。这就要求我们考察的所有字符，应该在同一个“窗口”内，这样的问题非常适合用滑动窗口的思路来解决。

而所谓的“最小子串”，当然就是符合要求的、长度最小的子串了。

另外还有一个小细节：需要找出“包含 T 所有字符的最小子串”，那 T 中的字符会不会有重复呢？给出的示例“ABC”没有重复，但提交代码之后会发现，测试用例是包含有重复字符的 T 的，比如“ABBC”在这种情况下，重复出现的字符“B”在子串中也要重复出现才可以。

5.2.3 方法一：暴力法

最简单直接的方法，就是直接枚举出当前字符串所有的子串，然后一一进行比对，选出覆盖 T 中所有字符的最小的那个。进一步我们发现，其实只需要枚举长度大于等于 T 的子串就可以了。

这里的核心问题是，怎样判断一个子串中包含了 T 中的所有字符？

如果 T 中没有重复，那这个非常简单，只要再遍历一遍 T，依次检查每个字符是否包含

就可以了；但现在 T 中字符可能重复，如果一个字符“A”重复出现 3 次，那我们寻找的子串中也必须有 3 个“A”。

所以我们发现，只要统计出 T 每个字符出现的次数，然后在子串中进行比对就可以。这可以用一个 HashMap 来进行存储，当然也可以更简单的只用一个数组来存。

子串 S 符合要求的条件是：统计 T 中每个字符出现的次数，全部小于等于在 S 中出现次数。

代码如下：

```
public class MinimumWindowSubstring {
    public String minWindow(String s, String t) {
        String minSubString = "";
        HashMap<Character, Integer> tCharFrequency = new HashMap<>();
        // 统计 t 中的字符频次
        for (int i = 0; i < t.length(); i++){
            char c = t.charAt(i);
            int count = tCharFrequency.getOrDefault(c, 0);
            tCharFrequency.put(c, count + 1);
        }
        // 遍历每个字符
        for (int i = 0; i < s.length(); i++){
            for (int j = i + t.length(); j <= s.length(); j++){
                HashMap<Character, Integer> subStrCharFrequency = new
HashMap<>();
```

```
                for (int k = i; k < j; k++){
                    char c = s.charAt(k);
                    int count = subStrCharFrequency.getOrDefault(c, 0);
                    subStrCharFrequency.put(c, count + 1);
                }
                if (check(tCharFrequency, subStrCharFrequency) && (j - i < minSubString.length() || minSubString.equals("") )){
                    minSubString = s.substring(i, j);
                }
            }
        }
        return minSubString;
    }
    // 定义一个方法，用来检查子串是否符合要求
    public boolean check( HashMap<Character, Integer> tFreq, HashMap<Character, Integer> subStrFreq ){
        for (char c: tFreq.keySet()) {
            if (subStrFreq.getOrDefault(c, 0) < tFreq.get(c)){
                return false;
            }
        }
        return true;
    }
}
```

复杂度分析

时间复杂度：O(|s|^3)，事实上，应该写作 O(|s|^3+|t|)，这里|s|表示字符串 s 的长度，|t|表示 t 的长度。我们枚举 s 所有的子串，之后又要对每一个子串统计字符频次，所以是三

重循环，耗费 O(|s|^3)。另外还需要遍历 t 统计字符频次，耗费 O(|t|)。t 的长度本身要小于 s，而且本题的应用场景一般情况是关键字的全文搜索，t 相当于关键字，长度应该远小于 s，所以可以忽略不计。

空间复杂度：O(C)，这里 C 表示字符集的大小。我们用到了 HashMap 来存储 S 和 T 的字符频次，而每张哈希表中存储的键值对不会超过字符集的大小。

5.2.4 方法二：滑动窗口

暴力法的缺点是显而易见的：时间复杂度过大，超出了运行时间限制。在哪些方面可以优化呢？

仔细观察可以发现，我们在暴力求解的时候，做了很多无用的比对：对于字符串“ADOBECODEBANC”，当找到一个符合条件的子串“ADOBEC”后，我们会继续仍以“A”作为起点扩展这个子串，得到一个符合条件的“ADOBECO”——它肯定符合条件，也肯定比之前的子串长，这其实是完全不必要的。

代码实现上，我们可以定义两个指针：指向子串“起始点”的左指针，和指向子串“结束点”的右指针。它们一个固定、另一个移动，彼此交替向右移动，就好像开了一个大小可变的窗口、在不停向右滑动一样，所以这就是非常经典的滑动窗口解决问题的应用场景。所以有时候，滑动窗口也可以归类到双指针法。

代码如下：

```
public String minWindow(String s, String t) {
    String minSubString = "";
    HashMap<Character, Integer> tCharFrequency = new HashMap<>();
    for (int i = 0; i < t.length(); i++){
        char c = t.charAt(i);
        int count = tCharFrequency.getOrDefault(c, 0);
        tCharFrequency.put(c, count + 1);
    }
```

```
    // 设置左右指针
    int lp = 0, rp = t.length();
    while ( rp <= s.length() ){
        HashMap<Character, Integer> subStrCharFrequency = new HashMap<>();
        for (int k = lp; k < rp; k++){
            char c = s.charAt(k);
            int count = subStrCharFrequency.getOrDefault(c, 0);
            subStrCharFrequency.put(c, count + 1);
        }
        if ( check(tCharFrequency, subStrCharFrequency) ) {
            if ( minSubString.equals("") || rp - lp <
minSubString.length() ){
                minSubString = s.substring(lp, rp);
            }
            lp++;
        }
        else
            rp++;
    }
    return minSubString;
}
```

复杂度分析

时间复杂度：O(|s|^2)，尽管运用双指针遍历字符串，可以做到线性时间 O(|s|)，但内部仍需要循环遍历子串，总共消耗 O(|s|)*O(|s|)=O(|s|^2)。

空间复杂度：O(C)，这里 C 表示字符集的大小。同样用到了 HashMap 来存储 S 和 T 的字符频次，而每张哈希表中存储的键值对不会超过字符集的大小。

5.2.5 方法三：滑动窗口优化

这里考虑进一步优化：

我们计算子串 S 的字符频次时，每次都要遍历当前子串，这做了很多重复工作。

其实，每次都只是左指针或右指针做了一次右移，只涉及到**一个字符的增减**。我们不需要重新遍历子串，只要找到移动指针之前的 S 中，这个字符的频次，然后再加一或者减一就可以了。

具体应该分左指针右移和右指针右移两种情况讨论。

- 左指针 i 右移（i –> i+1）。这时子串长度减小，减少的一个字符就是 s[i]，对应频次应该减一。
- 右指针 j 右移（j -> j+1）。这时子串长度增加，增加的一个字符就是 s[j]，对应频次加 1。

代码如下：

```
public String minWindow(String s, String t) {
    String minSubString = "";
    HashMap<Character, Integer> tCharFrequency = new HashMap<>();
    for (int i = 0; i < t.length(); i++){
        char c = t.charAt(i);
        int count = tCharFrequency.getOrDefault(c, 0);
        tCharFrequency.put(c, count + 1);
    }
    HashMap<Character, Integer> subStrCharFrequency = new HashMap<>();
    int lp = 0, rp = 1;
    while ( rp <= s.length() ){
        char newChar = s.charAt(rp - 1);
        if ( tCharFrequency.containsKey(newChar) ){
            subStrCharFrequency.put(newChar,
```

```
subStrCharFrequency.getOrDefault(newChar, 0) + 1);
        }
        while ( check(tCharFrequency, subStrCharFrequency) && lp < rp ){
            if ( minSubString.equals("") || rp - lp <
minSubString.length() ){
                minSubString = s.substring(lp, rp);
            }
            char removedChar = s.charAt(lp);
            if ( tCharFrequency.containsKey(removedChar) ){
                subStrCharFrequency.put(removedChar,
subStrCharFrequency.getOrDefault(removedChar, 0) - 1);
            }
            lp++;
        }
        rp++;
    }
    return minSubString;
}
```

复杂度分析

时间复杂度：O(|s|)。尽管有双重循环，但我们可以发现，内外两重循环其实做的只是分别移动左右指针。最坏情况下左右指针对 s 的每个元素各遍历一遍，哈希表中对 s 中的每个元素各插入、删除一次，对 t 中的元素各插入一次。另外，每次调用 check 方法检查是否可行，会遍历整个 t 的哈希表。

哈希表的大小与字符集的大小有关，设字符集大小为 C，则渐进时间复杂度为 O(C·|s|+|t|)。如果认为 t 的长度远小于 s，可以近似为 O(|s|)。

这种解法实现了线性时间运行。

空间复杂度：O(C)，这里 C 表示字符集的大小。同样用到了 HashMap 来存储 S 和 T 的

字符频次，而每张哈希表中存储的键值对不会超过字符集的大小。

5.2.6 方法四：滑动窗口进一步优化

我们判断S是否满足包含T中所有字符的时候，调用的方法check其实又是一个暴力法：遍历T中所有字符频次，一一比对。上面的复杂度分析也可以看出，遍历s只用了线性时间，但每次都要遍历一遍T的频次哈希表，这就耗费了大量时间。

我们已经知道，每次指针的移动，只涉及到**一个字符的增减**。所以我们其实不需要知道完整的频次 HashMap，只要获取改变的这个字符的频次，然后再和 T 中的频次比较，就可以知道新子串是否符合要求了。

代码如下：

```
public String minWindow(String s, String t) {
    String minSubString = "";
    HashMap<Character, Integer> tCharFrequency = new HashMap<>();
    for (int i = 0; i < t.length(); i++){
        char c = t.charAt(i);
        int count = tCharFrequency.getOrDefault(c, 0);
        tCharFrequency.put(c, count + 1);
    }
    HashMap<Character, Integer> subStrCharFrequency = new HashMap<>();
    int lp = 0, rp = 1;

    int charCount = 0;

    while ( rp <= s.length() ){
        char newChar = s.charAt(rp - 1);
        if ( tCharFrequency.containsKey(newChar) ){
            subStrCharFrequency.put(newChar,
```

```
subStrCharFrequency.getOrDefault(newChar, 0) + 1);
            if ( subStrCharFrequency.get(newChar) <=
tCharFrequency.get(newChar) )
                charCount ++;
        }
        while ( charCount == t.length() && lp < rp ){
            if ( minSubString.equals("") || rp - lp <
minSubString.length() ){
                minSubString = s.substring(lp, rp);
            }
            char removedChar = s.charAt(lp);
            if ( tCharFrequency.containsKey(removedChar) ){
                subStrCharFrequency.put(removedChar,
subStrCharFrequency.getOrDefault(removedChar, 0) - 1);
                if ( subStrCharFrequency.get(removedChar) <
tCharFrequency.get(removedChar) )
                    charCount --;
            }
            lp++;
        }

        rp++;
    }
    return minSubString;
}
```

复杂度分析

时间复杂度：O(|s|)。同样，内外双重循环只是移动左右指针遍历了两遍 s；而且由于引入了 charCount，对子串是否符合条件的判断可以在常数时间内完成，所以整体时间复杂

度为 O(|s|+|t|)，近似为 O(|s|)。

空间复杂度：O(C)，这里 C 表示字符集的大小。同样用到了 HashMap 来存储 S 和 T 的字符频次，而每张哈希表中存储的键值对不会超过字符集的大小。

5.3 找到字符串中所有字母异位词（#438）

5.3.1 题目说明

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。

字符串只包含小写英文字母，并且字符串 s 和 p 的长度都不超过 20100。

说明：

- 字母异位词指字母相同，但排列不同的字符串。
- 不考虑答案输出的顺序。

示例 1:

```
输入:
s: "cbaebabacd" p: "abc"
输出:
[0, 6]
解释:
起始索引等于 0 的子串是 "cba", 它是 "abc" 的字母异位词。
起始索引等于 6 的子串是 "bac", 它是 "abc" 的字母异位词。
```

示例 2:

```
输入:
s: "abab" p: "ab"
输出:
```

```
[0, 1, 2]
解释:
起始索引等于 0 的子串是 "ab", 它是 "ab" 的字母异位词。
起始索引等于 1 的子串是 "ba", 它是 "ab" 的字母异位词。
起始索引等于 2 的子串是 "ab", 它是 "ab" 的字母异位词。
```

5.3.2 分析

“字母异位词”，指“字母相同，但排列不同的字符串”。注意这里所说的“排列不同”，是所有字母异位词彼此之间而言的，并不是说要和目标字符串 p 不同。

另外，我们同样应该考虑，p 中可能有重复字母。

5.3.3 方法一：暴力法

最简单的想法，自然还是暴力法。就是直接遍历 s 中每一个字符，把它当作子串的起始，判断长度为 p.length()的子串是否是 p 的字母异位词就可以了。

考虑到子串和 p 中都可能有重复字母，我们还是用一个额外的数据结构，来保存每个字母的出现频次。由于本题的字符串限定只包含小写字母，所以我们可以简单地用一个长度为 26 的 int 类型数组来表示，每个位置存放的分别是字母 a~z 的出现个数。

代码如下：

```
public class FindAllAnagrams {
    public List<Integer> findAnagrams(String s, String p) {
        int[] pCharCounts = new int[26];
        for ( int i = 0; i < p.length(); i++ ){
            pCharCounts[p.charAt(i) - 'a'] ++;
        }
        ArrayList<Integer> result = new ArrayList<>();
        // 遍历 s
        for ( int i = 0; i <= s.length() - p.length(); i++ ){
```

```
            boolean isMatch = true;
            int[] subStrCharCounts = new int[26];
            for ( int j = i; j < i + p.length(); j++ ){
                subStrCharCounts[s.charAt(j) - 'a'] ++;
                if ( subStrCharCounts[s.charAt(j) - 'a'] >
pCharCounts[s.charAt(j) - 'a'] ) {
                    isMatch = false;
                    break;
                }
            }
            if ( isMatch )
                result.add(i);
        }
        return result;
    }
}
```

复杂度分析

时间复杂度：O(|s| * |p|)，其中|s|表示 s 的长度，|p|表示 p 的长度。时间开销主要来自双层循环，循环的迭代次数分别是(s.length-p.length+1)和 p.length，所以时间复杂度为 O((|s|-|p|+1) * |p|)，去除低阶复杂度，最终的算法复杂度为 O(|s| * |p|)。

空间复杂度：O(1)。需要两个大小为 26 的计数数组，分别保存 p 和当前子串的字母个数。尽管循环迭代过程中在不断申请新的空间，但是上一次申请的数组空间应该可以得到复用，所以实际上一共花费了 2 个数组的空间，因为数组大小是常数，所以空间复杂度为 O(1)。

5.3.4 方法二：滑动窗口（双指针）

暴力法的缺点是显而易见的：时间复杂度较大，运行耗时较长。

我们在暴力求解的时候，其实对于很多字母是做了多次统计的。子串可以看作字符串上开窗截取的结果，自然想到，可以定义左右指针向右移动，实现滑动窗口的作用。在指针移

动的过程中，字符只会被遍历一次，时间复杂度就可以大大降低。

代码如下：

```
public List<Integer> findAnagrams(String s, String p) {
        int[] pCharCounts = new int[26];
        for ( int i = 0; i < p.length(); i++ ){
            pCharCounts[p.charAt(i) - 'a'] ++;
        }
        ArrayList<Integer> result = new ArrayList<>();
        // 定义左右指针
        int lp = 0, rp = 1;
        int[] subStrCharCounts = new int[26];
        while ( rp <= s.length() ){
            char newChar = s.charAt(rp - 1);
            subStrCharCounts[newChar - 'a'] ++;
            while ( subStrCharCounts[newChar - 'a'] > pCharCounts[newChar -
'a'] && lp < rp ){
                char removedChar = s.charAt(lp);
                subStrCharCounts[removedChar - 'a'] --;
                lp ++;
            }
            if ( rp - lp == p.length() )
                result.add(lp);
            rp ++;
        }
        return result;
}
```

复杂度分析

时间复杂度：O(|s|)。 窗口的左右指针最多都到达 s 串结尾，s 串每个字符最多被左右指针都经过一次，所以时间复杂度为 O(|s|)。

空间复杂度：O(1)。只需要两个大小为 26 的计数数组，大小均是确定的常量，所以空间复杂度为 O(1)。