九章算法班2022版直播第2章

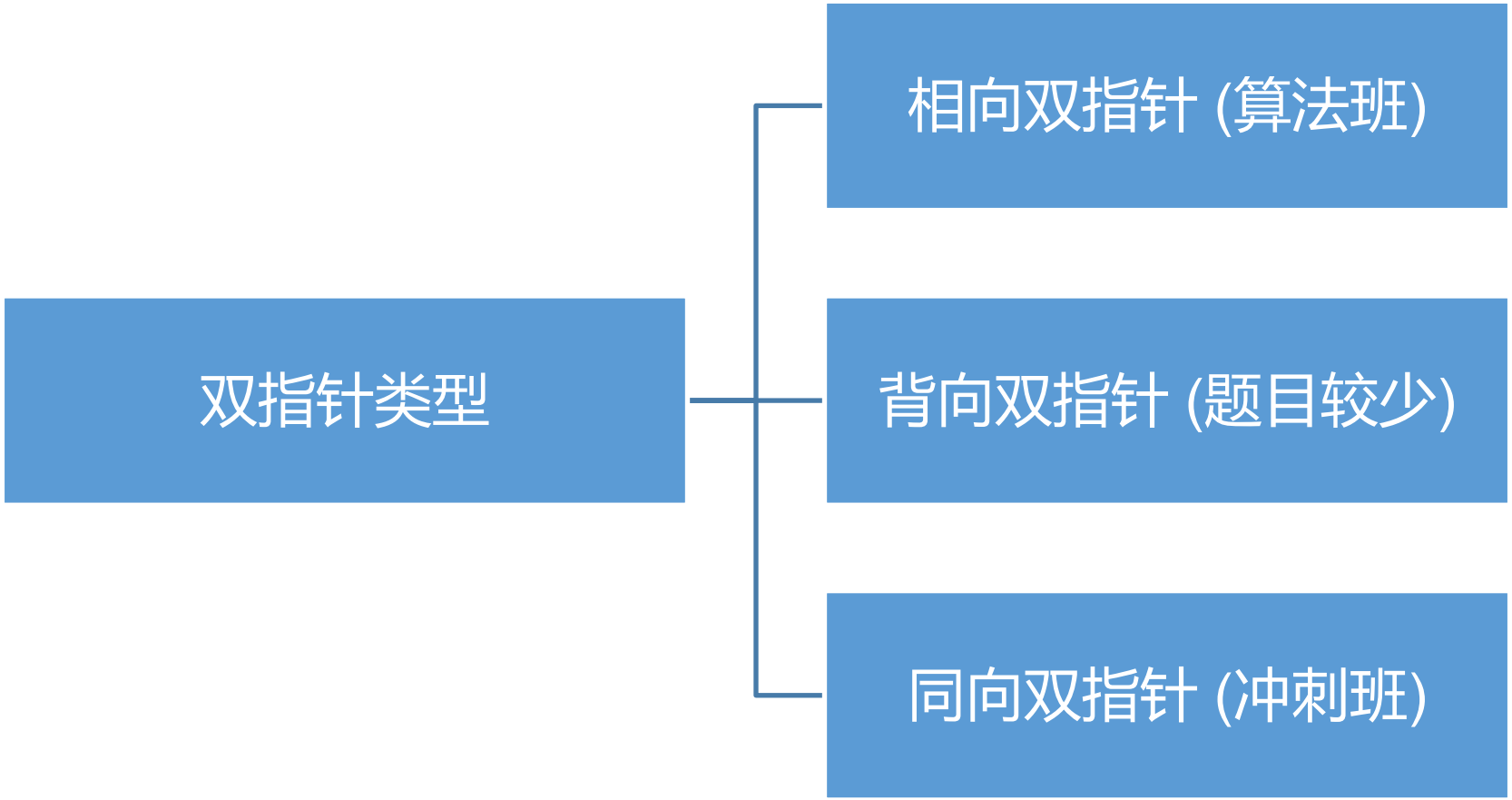
高频算法之王 ——

双指针算法之相向双指针

主讲：夏天

双指针类型题目概述

双指针类型题目概述



相向双指针

两根指针一头一尾，向中间靠拢直到相遇

时间复杂度 O(n)

Two Sum

两数之和

56 Two Sum 两数之和 (经典必会)

Given an array of integers, find two numbers such that they add up to a specific target number.

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are zero-based.

给一个整数数组，找到两个数使得他们的和等于一个给定的数 target。

你需要实现的函数twoSum需要返回这两个数的下标, 并且第一个下标小于第二个下标。注意这里下标的范围是 0 到 n-1。

输入：

numbers = [15, 2, 7, 11]

target = 9

输出：

[1,2]（返回的是下标）

解释:

numbers[1] + numbers[2] = 2 + 7 = 9

在预习内容中我们已经讲解了相向双指针的经典题 Two Sum

在一串**有序**数中，寻找**两个**数字，**和为target**

接下来我们来看这类问题可能的变化

然而，你预习了吗？

复习（review）和预习（preview）

有个P的区别

607 Two Sum III - Data structure design 两数之和 III- 数据结构设计

Design and implement a TwoSum class. It should support the following operations: add and find.

add - Add the number to an internal data structure.

find - Find if there exists any pair of numbers which sum is equal to the value.

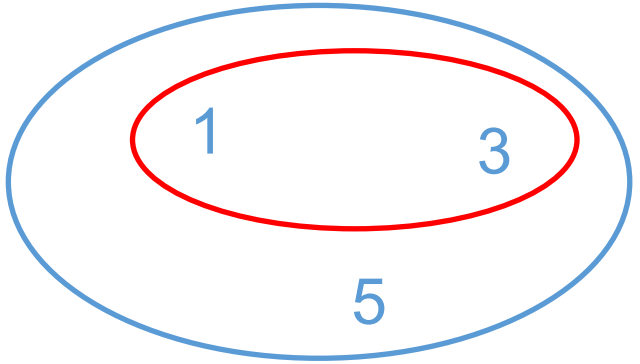
设计并实现一个 TwoSum 类。他需要支持以下操作:add 和 find。

add - 把这个数添加到内部的数据结构。

find - 是否存在任意一对数字之和等于这个值

```python
class TwoSum:
    """
    @param number: An integer
    @return: nothing
    """
    def add(self, number):
        # write your code here

    """
    @param value: An integer
    @return: Find if there exists any pair of numbers which sum is equal to the value.
    """
    def find(self, value):
        # write your code here
```



Example

add(1);

add(3);

add(5);

find(4); // return **true**

find(7); // return **false**

```java
public class TwoSum {
    /**
     * @param number: An integer
     * @return: nothing
     */
    public void add(int number) {
        // write your code here
    }

    /**
     * @param value: An integer
     * @return: Find if there exists any pair of numbers which sum is equal to the value
     */
    public boolean find(int value) {
        // write your code here
    }
}
```

跟面试官核实：

1. 有没有**重复**数字？ 有。特殊情况: 3 + 3 = 6
2. 如果有多组答案，是不是**找到一组**，就可以返回true？ 是
3. 数据是**升序**的吗？ 不是

方法1：暴力解法（brute force）

已有数据：[1， 5， 3， 2， 7， 4]

是否有两数之和为9？

暴力解法（brute force）

add 直接加入数据末端

find 两层for循环枚举所有数对，判断是否符合条件

```
for (从下标0开始的每个数字N1):
    for (N1后面的每个数字N2):
        if (N1 + N2) == target:
            return true
return false
```

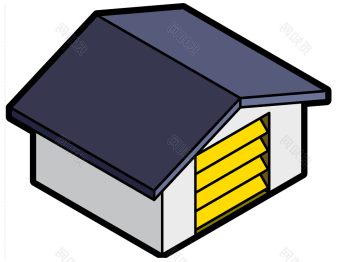
	add	find
时间复杂度	O(1)	O(N^2)
空间复杂度	O(N)，如果插入了N个数据	

数据结构设计题 —— 方案选择

设计一个数据结构存储数据

设计几个方法对数据结构里的数据进行增删查改（CRUD）操作

哪个方案好？	add	find
方案A	O(1) 快	O(N) 慢
方案B	O(logN) 慢	O(logN) 快



要跟面试官进行沟通，根据具体的商业需求（比如，方法被调用的频率）进行设计

方法2: 排序数组 + 双指针，list + two pointers

1 2 3 5 2

1 2 3 2 5

1 2 2 3 5

```python
class TwoSum:
    def __init__(self):
        # list装升序数据，因为twoSum经典算法需要在有序数列中寻找
        self.nums = []

    def add(self, number):
        self.nums.append(number)
        index = len(self.nums) - 1
        # 按照插入排序的方法，加入新数字，保持nums升序
        # 每次将数组最后一个元素作为插入元素，与它前面有序（已排好序）的数组元素依次进行比较，
        # 如果没有在正确的位置，交换两元素，继续下一轮比较
        # 如果在正确的位置，结束
        while index > 0 and self.nums[index - 1] > self.nums[index]:
            # 在list中，交换(index - 1)和index对应的值(这里有换行符\)
            self.nums[index - 1], self.nums[index] = \
            self.nums[index], self.nums[index - 1],
            index -= 1

    # ...后面还有find代码
```

如果想插入2，把整个数组sort一下，不就把2插入到升序数组中了吗？

排序时间复杂度O(NlogN)，比较慢。而插入为O(N)，更加高效

为什么需要确保 index > 0？ 因为需要获得数组中(index – 1)位置的数字

```java
public class TwoSum {
    // list装升序数据
    public List<Integer> nums;
    public TwoSum() {
        nums = new ArrayList<Integer>();
    }

    public void add(int number) {
        nums.add(number);
        int index = nums.size() - 1;
        // 按照插入排序的方法，加入新数字，保持nums升序
        // 每次将数组最后一个元素作为插入元素，与它前面有序（已排好序）的数组元素依次进行比较，
        // 如果没有在正确的位置，交换两元素，继续下一轮比较
        // 如果在正确的位置，结束
        while (index > 0 && nums.get(index - 1) > nums.get(index)) {
            swap(nums, index);
            index--;
        }
    }

    // 在list中，交换(index - 1)和index对应的值
    private void swap(List<Integer> nums, int index) {
        int temp = nums.get(index);
        nums.set(index, nums.get(index - 1));
        nums.set(index - 1, temp);
    }

    // ...后面还有find代码
}
```

add时间复杂度	O(N)

方法2: 排序数组 + 双指针，list + two pointers

为什么可以“草率”地移动指针，淘汰某个数字？

时间的优化在于**不需要枚举所有数对，就可以批量淘汰不符合条件的数对**

```python
    # ...前面还有add代码

    # 经典two sum，在排序数组中用相向双指针寻找和为target的一对数字
    def find(self, targetValue):
        # 左右两颗相向双指针
        left, right = 0, len(self.nums) - 1
        while left < right:
            two_sum = self.nums[left] + self.nums[right]
            # twoSum小于target，左指针向中间移动，寻找更大twoSum
            if two_sum < targetValue:
                left += 1
            # twoSum大于target，右指针向中间移动，寻找更小twoSum
            elif two_sum > targetValue:
                right -= 1
            # 找到，返回true
            else:
                return True
        # 找不到，返回false
        return False
```

```java
    // ...前面还有add代码

    // 经典two sum，在排序数组中用相向双指针寻找和为target的一对数字
    public boolean find(int targetValue) {
        // 左右两颗相向双指针
        int left = 0, right = nums.size() - 1;
        while (left < right) {
            int twoSum = nums.get(left) + nums.get(right);
            // twoSum小于target，左指针向中间移动，寻找更大twoSum
            if (twoSum < targetValue) {
                left++;
            }
            // twoSum大于target，右指针向中间移动，寻找更小twoSum
            else if (twoSum > targetValue) {
                right--;
            }
            // 找到，返回true
            else {
                return true;
            }
        }
        // 找不到，返回false
        return false;
    }
}
```

	add	find
时间复杂度	O(N)	O(N)
空间复杂度	O(N)	

有没有其他方法一边增加数一边保持数组有序？

方法	分析
Binary Search + List/Array Insert	用BinarySearch可以通过O(logN)在有序数列中找到新数据插入位置 但是在List/Array(连续空间)中插入新元素会导致元素整体后移，需要O(N) 比如，[1,3,4,5,6,7] 中插入 2
Binary Search + Linked List Insert	Binary Search 是基于数组的算法，不是基于链表的算法 数组 Array - 连续型存储 - 支持 O(1) Index Access 链表 Linked List - 离散型存储 – 不支持 O(1) Index Access，只支持 O(n) Index Access 链表中插入一个元素虽然是 O(1) 的，但是找到插入位置需要花 O(n) 的时间 (不能用二分)
Heap (ProrityQueue)	堆是一个树状结构，堆内部的元素组织顺序不是有序的 堆可以实现 O(logN) 的插入 但是双指针Two Sum的方法是基于一个 Sorted Array进行的，不能在堆中完成
TreeMap (红黑树)	TreeMap是一个树状结构 (O(N) 时间中序遍历之后可以得到一个有序数组) 双指针Two Sum的方法是基于一个 Sorted Array 进行的，不能在 TreeMap 中完成 TreeMap 可以实现 O(logN) add 和 O(N) find 不如 HashMap (O(1) add 和 O(N) find 的方法好

方法3：哈希表，dict / HashMap 思路解析

已有数据：[2， 3， 5， 3]，是否有两数之和为 8？

暴力解法（brute force）

add 直接加入数据末端

find 两层for循环枚举所有数对，判断是否符合条件

```
for (从下标0开始的每个数字N1):
    for (N1后面的每个数字N2):
        if (N1 + N2) == target:
            return true
return false
```

	add	find
时间复杂度	O(1)	O(N^2)
空间复杂度	O(N)，如果插入了N个数据	

优化解法

把流入的数据用 dict / HashMap 做分类统计，key = 数字，value = 出现次数

Key (数字)	Value (频次)
2	1
3	2
5	1

为什么要记录频次？

for 循环 dict / HashMap 里的每个 num，检查一下 target - num 是不是也在 HashMap 里，这个过程需要总共 O(n * 1) 的时间。

利用dict / HashMap查找 O(1) 的优点，省去了暴力方法的内层循环，优化了时间

	add	find
时间复杂度	O(1)	O(N)
空间复杂度	O(N)，如果插入了N个数据	

方法3：哈希表，dict / HashMap


```python
class TwoSum(object):
    def __init__(self):
        # key是数字，value是数字出现的次数
        self.num_to_cnt_map = {}

    # Add the number to an internal data structure.
    # @param number {int}
    # @return nothing
    def add(self, number):
        # 数字出现次数加1。如果这个数字第一次出现，则之前出现的次数为0
        self.num_to_cnt_map[number] = self.num_to_cnt_map.get(number , 0) + 1


    # Find if there exists any pair of numbers which sum is equal to the value.
    # @param value {int}
    # @return true if can be found or false
    def find(self, value):
        # 遍历map中所有的key
        for num1 in self.num_to_cnt_map:
            # 寻找跟num1匹配的num2
            num2 = value - num1
            # 如果num1和num2都为value的1/2，则num2的值需要在map中出现2次
            num2Cnt = 2 if (num1 == num2) else 1
            # 如果找到，返回true。如果找不到num2，则出现次数为0
            if self.num_to_cnt_map.get(num2, 0) >= num2Cnt:
                return True
        # 找不到任何配对，返回false
        return False
```

```java
public class TwoSum {
    // key是数字，value是数字出现的次数
    private Map<Integer, Integer> numToCntMap;


    public TwoSum() {
        numToCntMap = new HashMap<Integer, Integer>();
    }

    // Add the number to an internal data structure.
    public void add(int number) {
        // 数字出现次数加1。如果这个数字第一次出现，则之前出现的次数为0
        numToCntMap.put(number, numToCntMap.getOrDefault(number, 0) + 1);
    }

    // Find if there exists any pair of numbers which sum is equal to the value.
    public boolean find(int value) {
        // 遍历map中所有的key
        for (Integer num1 : numToCntMap.keySet()) {
            // 寻找跟num1匹配的num2
            int num2 = value - num1;
            // 如果num1和num2都为value的1/2，则num2的值需要在map中出现2次
            int num2Cnt = (num1 == num2) ? 2 : 1;
            // 如果找到，返回true。如果找不到num2，则出现次数为0
            if (numToCntMap.getOrDefault(num2, 0) >= num2Cnt) {
                return true;
            }
        }
        // 找不到任何配对，返回false
        return false;
    }
}
```

	add	find
时间复杂度	O(1)	O(N)
空间复杂度	O(N)	

57 3Sum 三数之和

Given an array *S* of n integers, are there elements *a*, *b*, *c* in *S* such that a + b + c = 0? Find all unique triplets in the array which gives the sum of zero.

Elements in a triplet (a, b, c) must be in non-descending order. (i.e., a ≤ b ≤ c)

The solution set **must not contain duplicate triplets**.

给出一个有n个整数的数组S，在S中找到三个整数a, b, c，找到所有使得 **a + b + c = 0** 的三元组。

在三元组(a, b, c)，要求 a <= b <= c。

结果**不能包含重复的三元组**。

输入：

[-1,0,1,2,-1,-4]

输出：

[[-1, 0, 1],[-1, -1, 2]]

输入：

[2,7,11,15]

输出：

[]

跟面试官核实：

1. 输入是排好序的吗？ 不是

2. 有没有**重复**数字？ 有

特殊情况：

输入：[0, 0, -1, -1, 1, 1]

输出：[0, -1, 1] 没有多个重复结果

代码解析

暴力可行解：三层for循环，找到所有满足条件的三元组，并去重。时间复杂度O(N^3)，空间复杂度O(1)

优化解：如果选定一个点A，那么另外两个点的目标和为-A，转化为two sum问题。 **选定三元组中的最小值，中间值，还是最大值？**

-7 -7 2 2 3 4 5 5

```python
def threeSum(self, nums):
    results = []

    # 特殊情况处理
    if not nums or len(nums) < 3:
        return results

    # 原数据无序，经典two sum需要在有序列中进行
    # nums = sorted(nums)
    nums.sort()


    length = len(nums)
    # 遍历三元组中的最小数
    for i in range(0, length - 2):
        # 如果第当前元素和左边元素一样，跳过
        if i > 0 and nums[i] == nums[i - 1]:
            continue
        # 好的代码，不需要太多注释
        # 这里left和right可以不写，但是写出来不解自明
        left = i + 1
        right = length - 1
        target = -nums[i]
        # 用经典two sum寻找所有和为target的不重复数对
        self.find_two_sum(nums, left, right, target, results)
    return results
```

```java
public List<List<Integer>> threeSum(int[] nums) {
    List<List<Integer>> results = new ArrayList<>();

    // 特殊情况处理
    if (nums == null || nums.length < 3) {
        return results;
    }

    // 原数据无序，经典two sum需要在有序列中进行
    Arrays.sort(nums);

    // 遍历三元组中的最小数
    for (int i = 0; i < nums.length - 2; i++) {
        // 如果第当前元素和左边元素一样，跳过
        if (i > 0 && nums[i] == nums[i - 1]) {
            continue;
        }
        // 好的代码，不需要太多注释
        // 这里left和right可以不写，但是写出来不解自明
        int left = i + 1, right = nums.length - 1;
        int target = -nums[i];
        // 用经典two sum寻找所有和为target的不重复数对
        twoSum(nums, left, right, target, results);
    }

    return results;
}
```

代码解析

带有去重逻辑的相向双指针Two Sum

a + b + c = 0　答案： [-7, 2, 5], [-7, 3, 4]

-7（第一个数）　-7　2　2　3　4　5　5

```python
def find_two_sum(self, nums, left, right, target, results):
    while left < right:
        if nums[left] + nums[right] == target:
            results.append([-target, nums[left], nums[right]])
            right -= 1
            left += 1
            # 如果左指针当前数字跟左边数字相同，左指针向中间移动，跳过重复
            while left < right and nums[left] == nums[left - 1]:
                left += 1
            # 如果右指针当前数字跟右边数字相同，右指针向中间移动，跳过重复
            while left < right and nums[right] == nums[right + 1]:
                right -= 1
        # 当前sum小于target，右移左指针，去找更大的sum
        elif nums[left] + nums[right] > target:
            right -= 1
        # 当前sum大于target，左移右指针，去找更小的sum
        else:
            left += 1
```

去重，跳过重复数字

时间复杂度	O(NlogN + N^2)	O(NlogN) 排序，O(N*N) 进行N次two sum 利用two sum降维优化了时间复杂度
空间复杂度	O(K)	K为解的个数

```java
public void twoSum(int[] nums,
                   int left,
                   int right,
                   int target,
                   List<List<Integer>> results) {
    while (left < right) {
        int sum = nums[left] + nums[right];
        if (sum == target) {
            generateTriplet(nums, results, left, right, target);
            left++;
            right--;
            // 如果左指针当前数字跟左边数字相同，左指针向中间移动，跳过重复
            while (left < right && nums[left] == nums[left - 1]) {
                left++;
            }
            // 如果右指针当前数字跟右边数字相同，右指针向中间移动，跳过重复
            while (left < right && nums[right] == nums[right + 1]) {
                right--;
            }
        }
        // 当前sum小于target，右移左指针，去找更大的sum
        else if (sum < target) {
            left++;
        }
        // 当前sum大于target，左移右指针，去找更小的sum
        else {
            right--;
        }
    }
}

private void generateTriplet(int[] nums, List<List<Integer>> results,
                             int left, int right, int target) {
    ArrayList<Integer> triplet = new ArrayList<>();
    triplet.add(-target);
    triplet.add(nums[left]);
    triplet.add(nums[right]);
    results.add(triplet);
}
```

去重，跳过重复数字

382 Triangle Count

Given an array of integers, how many three numbers can be found in the array, so that we can build an triangle whose three edges length is the three numbers that we find?

给定一个整数数组，在该数组中，寻找三个数，分别代表三角形三条边的长度，问，可以寻找到多少组这样的三个数来组成三角形？

输入：

[4, 3, 6, 7]

输出：

3 （三角形个数）

解释:

(3, 4, 6)

(3, 6, 7)

(4, 6, 7)

跟面试官核实：

1. 输入是否**有序**？ 不是
2. 有没有**重复**数字？ 有
3. 需不需要去掉**重复**答案 不需要

特殊情况：

输入： [4, 4, 4, 4]

输出： 答案是多少？ 4

在四个4中，任何三个元素都可以够成等边三角形，C(3, 4) = 4

如何判断三边是否可以组成三角形？

小边1 + 小边2 > 大边

等价于 小边1 > 大边 - 小边2

代码解析

可行解：三层for循环，找到所有满足条件的三元组。时间复杂度O(N^3)，空间复杂度O(1)

如果选定一个点A，那么另外两个点的目标和为-A，转化为two sum问题。**选定三元组中的最小值，中间值，还是最大值**？

1 2 2 3 3

固定**最大边**（初始值下标为2，保证之前至少有两个边），在最大边之前寻找两条较小边

```python
    def triangleCount(self, S):
        # 特殊情况处理
        if not S or len(S) < 3:
            return 0
        # 经典two sum双指针需要在有序数据上进行
        S.sort()
        ans = 0
        # 遍历最大边，在最大边的左边寻找两个小边
        for i in range(2, len(S)):
            ans += self.get_triangle_count(S, i)
        return ans

```

```java
    public int triangleCount(int[] S) {
        // 特殊情况处理
        if (S == null || S.length < 3) {
            return 0;
        }
        // 经典two sum双指针需要在有序数据上进行
        Arrays.sort(S);
        int cnt = 0;
        // 遍历最大边，在最大边的左边寻找两个小边
        for (int i = 2; i < S.length; i++) {
            cnt += getTriangleCount(S, i);
        }
        return cnt;
    }
```

代码解析

这不是一个经典的Tow Sum问题（A + B = Target），时间复杂度O(N)

而是一个变形（A + B > Target），如果要计算可能性的个数，时间复杂度O(N^2)

但是，**本题是求解的个数，而不是所有解**。可以利用two sum批量求解解的个数（不用一个个枚举），实现时间的优化

小边1 + 小边2 > 大边

1　　2　　2　　3　　3 (大边)

```python
    def get_triangle_count(self, ls, target_index):
        cnt = 0
        # 寻找范围为[0, targetIndex - 1]
        left = 0
        right = target_index - 1
        target_sum = ls[target_index]
        while left < right:
            # sum大于target，可以组成三角形
            if ls[left] + ls[right] > target_sum:
                # 一次求出多个可行解的个数
                cnt += right - left
                # 已经计入右指针所有可能的组合，右指针向中间移动
                right -= 1
            # sum小于target，左指针向右移动，寻找更大sum
            else:
                left += 1
        return cnt
```

时间复杂度	O(NlogN + N^2)	O(NlogN)排序，O(N^N)进行N次two sum
空间复杂度	O(1)	

```java
private int getTriangleCount(int[] arr, int targetIndex) {
    int cnt = 0;
    // 寻找范围为[0, targetIndex - 1]
    int left = 0;
    int right = targetIndex - 1;
    int targetSum = arr[targetIndex];
    int sum = 0;
    while (left < right) {
        sum = arr[left] + arr[right];
        // sum大于target，可以组成三角形
        if (sum > targetSum) {
            // 一次求出多个可行解的个数
            cnt += right - left;
            // 已经计入右指针所有可能的组合，右指针向中间移动
            right--;
        }
        // sum小于target，左指针向右移动，寻找更大sum
        else {
            left++;
        }
    }
    return cnt;
}
```

58 4Sum

Given an array *S* of *n* integers, are there elements *a*, *b*, *c*, and *d* in *S* such that *a* + *b* + *c* + *d* = *target*?

Find all unique quadruplets in the array which gives the sum of target.

Elements in a quadruplet (*a,b,c,d*) must be in **non-descending** order. (ie, *a* ≤ *b* ≤ *c* ≤ *d*)

The solution set must **not contain duplicate quadruplets**.

给一个包含n个数的整数数组S，在S中找到所有使得和为给定整数target的四元组(a, b, c, d)。

四元组(a, b, c, d)中，需要满足**a <= b <= c <= d**。答案中**不可以包含重复的四元组。**

输入:

[1,0,-1,0,-2,2]

sum = 0

输出:

[[-1, 0, 0, 1],

[-2, -1, 1, 2],

[-2, 0, 0, 2]]

输入:

[2,7,11,15]

sum = 3

输出:

[]

没有4数之和为3

跟面试官核实：

1. 输入是否有序？ 不是
2. 有没有**重复**数字？ 有
3. 需不需要去掉重复答案 需要

固定两个点，然后用双指针的做法，扫描一下后续数组，记录答案即可。

时间复杂度	O(NlogN + N^3)	O(NlogN)排序 O(N^3) = O(N*N)选择前两个数 * O(N)寻找后两个数
空间复杂度	O(K)	K为解的个数

976 4Sum II

Given four lists A, B, C, D of integer values, compute how many tuples (i, j, k, l) there are such that A[i] + B[j] + C[k] + D[l] is **zero**.

To make problem a bit easier, all A, B, C, D have same length of N where 0 ≤ N ≤ 500. All integers are in the range of -2^28 to 2^28 - 1 and the result is guaranteed to be at most 2^31 - 1.

给出 A, B, C, D 四个整数列表，计算有多少的tuple (i, j, k, l)满足A[i] + B[j] + C[k] + D[l]为 **0**。

为了简化问题，**A, B, C, D 具有相同的长度**，且长度N满足 0 ≤ N ≤ 500。所有的整数都在范围(-2^28, 2^28 - 1)内以及保证结果最多为2^31 - 1。

输入:

A = [1, 2]

B = [-2,-1]

C = [-1, 2]

D = [0, 2]

输出: 2

解释:

(0, 0, 0, 1) -> A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) + 2 = 0

(1, 1, 0, 0) -> A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) + 0 = 0

跟面试官核实：

1. 输入是否有序？ 不是

2. 有没有**重复**数字？ 有

3. 需不需要去掉重复答案 不需要

特殊情况：

输入：A = [0] , B = [0] , C = [0] , D = [0, 0]

输出： 2

D中两个0可以分别和ABC中的元素构成四元组

代码解析

这道题目是否似曾相识？

607 Two Sum III - Data structure design 两数之和 III- 数据结构设计

A = [1, 2]　　将 a,b 组成的和及其组成方案个数

B = [-2, -2]　　统计在hash里

C = [-1, 2]　　去枚举 c,d 的组合，找 -(c + d) 在

D = [0, 2]　　hash 里的组合数。

AB和的map

-1 ➔ 2

0 ➔ 2

```python
    """
    def fourSumCount(self, A, B, C, D):
        # key为(a + b)之和，value为出现频次
        dictionary = {}
        for a in A:
            for b in B:
                total = a + b
                # 如果dict中没有key sum，返回默认频次0
                dictionary[total] = dictionary.get(total, 0) + 1
        cnt = 0
        for c in C:
            for d in D:
                total = c + d
                # 如果sum出现在AB之和中，累加频次，否则加默认频次0
                cnt += dictionary.get(-total, 0)
        return cnt
```

```java
public int fourSumCount(int[] A, int[] B, int[] C, int[] D) {
    // key为(a + b)之和，value为出现频次
    Map<Integer, Integer> map = new HashMap<>();
    for (int a : A) {
        for (int b : B) {
            int sum = a + b;
            // 如果map中没有key sum，返回默认频次0
            int frqcy = map.getOrDefault(sum, 0);
            map.put(sum, frqcy + 1);
        }
    }
    int cnt = 0;
    for (int c : C) {
        for (int d : D) {
            int sum = c + d;
            // 如果-sum出现在AB之和中，累加频次
            cnt += map.getOrDefault(-sum, 0);
        }
    }
    return cnt;
}
}
```

时间复杂度	O(N^2)	N为单个数组的长度
空间复杂度	O(N^2)	N为单个数组的长度

89 K Sum K数之和

https://www.lintcode.com/problem/k-sum/description 求方案总数（**动态规划**）

https://www.lintcode.com/problem/k-sum-ii/description 求具体方案（**深度优先搜索**）

敬请期待在**动态规划**和**深度优先搜索**中对这两个问题的讲解

相关问题

统计所有和 <= target 的配对数

http://www.lintcode.com/problem/two-sum-less-than-or-equal-to-target/

http://www.jiuzhang.com/solutions/two-sum-less-than-or-equal-to-target/

统计所有和 >= target 的配对数

http://www.lintcode.com/en/problem/two-sum-greater-than-target/

http://www.jiuzhang.com/solutions/two-sum-greater-than-target/

课间休息

Partition

分区算法

31 Partition Array 分割数组 (经典必会)

Given an array nums of integers and an int k, partition the array (i.e move the elements in "nums") such that:

All elements < *k* are moved to the *left*, All elements >= *k* are moved to the *right*

Return the partitioning index, i.e. the first index *i* nums[*i*] >= *k*.

给出一个整数数组 nums 和一个整数 k。划分数组（即移动数组 nums 中的元素），使得：

所有小于k的元素移到左边，所有大于等于k的元素移到右边

返回数组划分的位置，即数组中第一个位置 i，满足 nums[i] 大于等于 k。

输入：

nums = [3,2,2,1]

k = 2

输出：

1 (返回下标)

解释：

The real array is[1,2,2,3].So return 1.

输入：

nums = []

k = 9

输出：

0 (返回下标)

解释：

Empty array, print 0.

分区 Partition

目标：把 < 2 的数移到左边，>= 2的数移到右边

如果**左指针** < 2就一直右移，遇到 >= 2 就停下

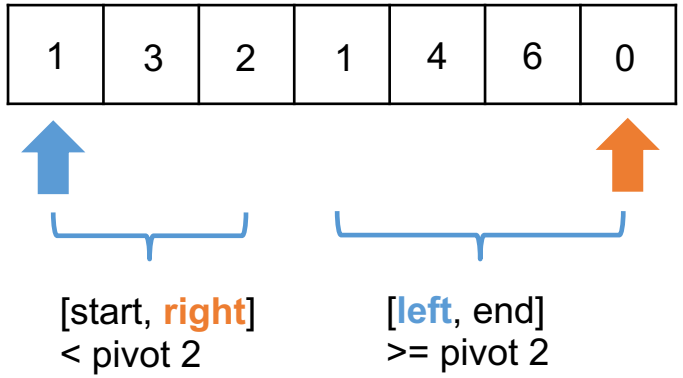
如果**右指针** >= 2就一直左移，遇到 < 2 就停下

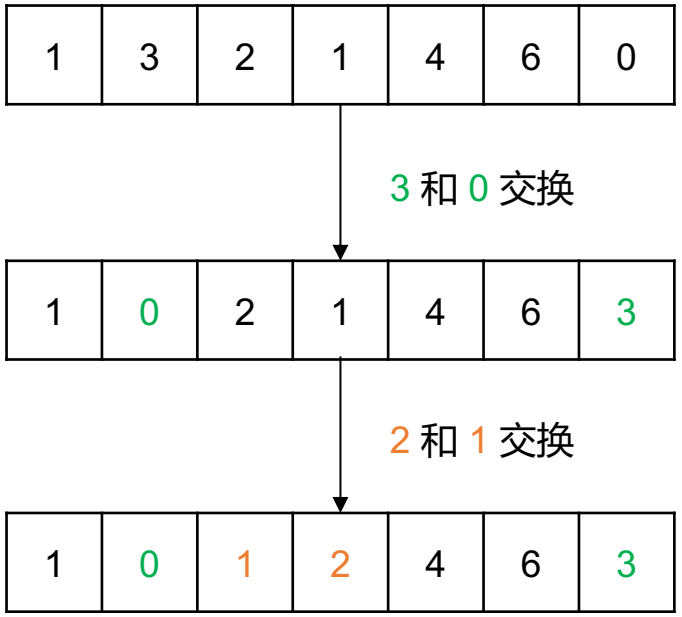
交换两个指针指向的数字，两个指针各向中间移动一步

一直到右指针在左指针左边为止

```
while left <= right:                                    < 2
    while left <= right and nums[left] 应该在左侧:
        left += 1                                       >= 2
    while left <= right and nums[right] 应该在右侧:
        right -= 1

    if left <= right:
        # 找到了一个不该在左侧的和不该在右侧的，交换他们
        nums[left], nums[right] = nums[right], nums[left]
        left += 1
        right -= 1
```





两层while循环时间复杂度是 O(N^2) 吗？

不是O(N^2) ，是O(N)。两个指针遍历了整个数组一次

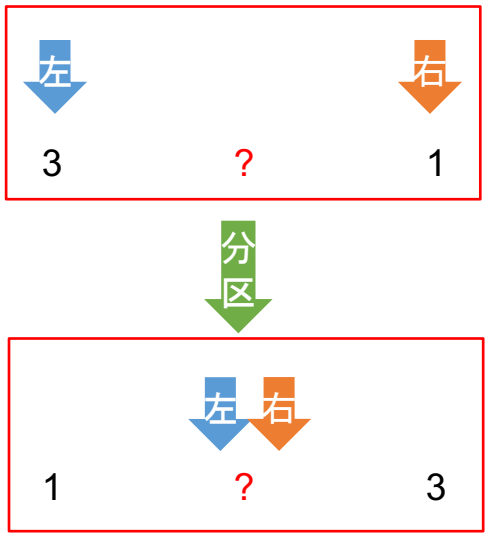
时间复杂度与最内层循环主体的执行次数有关与有多少重循环无关

为什么用 left <= right 而不是 left < right?


```
while left <= right:                                    < 2
    while left <= right and nums[left] 应该在左侧:
        left += 1                                       >= 2
    while left <= right and nums[right] 应该在右侧:
        right -= 1

    if left <= right:
        # 找到了一个不该在左侧的和不该在右侧的，交换他们
        nums[left], nums[right] = nums[right], nums[left]
        left += 1
        right -= 1
```

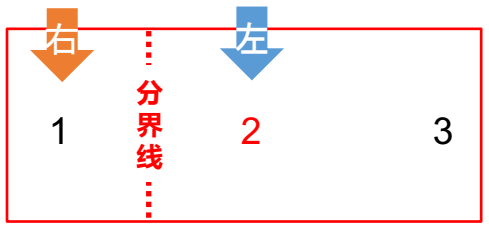
左边<2，右边>=2

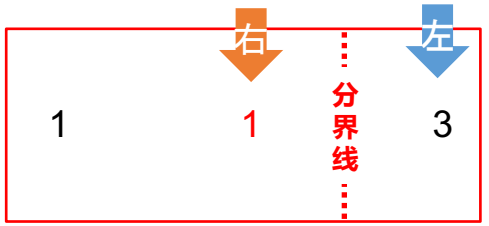


如果用 left < right, while 循环结束在 left == right

此时需要多一次 if 一句判断 nums[left] 到底是 < k 还是 >=k

因此使用 left <= right 可以省去这个判断





代码解析


```python
    def partitionArray(self, nums, k):
        # 特殊情况处理
        if not nums:
            return 0

        # 两端的相向双指针
        left, right = 0, len(nums) - 1

        while left <= right:
            # 左指针寻找一个不属于左边的数字
            while left <= right and nums[left] < k:
                left += 1
            # 右指针寻找一个不属于右边的数字
            while left <= right and nums[right] >= k:
                right -= 1
            if left <= right:
                # 交换左右指针数字，双方都到了正确的一端
                nums[left], nums[right] = nums[right], nums[left]
                left += 1
                right -= 1

        return left
```

```java
    public int partitionArray(int[] nums, int k) {
        // 特殊情况处理
        if (nums == null) {
            return 0;
        }

        // 两端的相向双指针
        int left = 0, right = nums.length - 1;
        while (left <= right) {
            // 左指针寻找一个不属于左边的数字
            while (left <= right && nums[left] < k) {
                left++;
            }
            // 右指针寻找一个不属于右边的数字
            while (left <= right && nums[right] >= k) {
                right--;
            }

            if (left <= right) {
                // 交换左右指针数字，双方都到了正确的一端
                int temp = nums[left];
                nums[left] = nums[right];
                nums[right] = temp;

                left++;
                right--;
            }
        }
        // 左指针位置即为右边partition起点
        return left;
    }
```

时间复杂度	O(N)
空间复杂度	O(1)

148 Sort Colors 颜色分类

Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

给定一个包含红，白，蓝且长度为 n 的数组，将数组元素进行分类使相同颜色的元素相邻，并按照红、白、蓝的顺序进行排序。

我们可以使用整数 0，1 和 2 分别代表红，黄，蓝。

要求：**O(N)的时间复杂度**

输入 : [1, 0, 2, 0, 2]

输出 : [0, 0, 1, 2, 2]

这个题目为什么有可能找到比O(NlogN)快的解法？

题目解析 —— Quick Sort （ Partition ）

array + 固定的有限种元素sort + O(N)时间 → **有限次快速排序分区Quick Sort Partition**

第一次partition：左边<1，右边>=1

第二次partition：左边<2，右边>=2

时间复杂度	O(N)	每次partition一个for循环O(N)，进行两次partition
空间复杂度	O(1)	

```python
def sortColors(self, a):
    self.partition_array(a, 1)
    self.partition_array(a, 2)

def partition_array(self, nums, k):
    # 指向<k区间的最后一个元素
    last_small_pointer = -1
    for i in range(len(nums)):
        # 如果nums[i] < k，需要交换并右移指针
        if nums[i] < k:
            # 指针先右移一位，指向当前元素应该去往的位置
            # 然后进行swap
            last_small_pointer += 1
            nums[last_small_pointer], nums[i] = nums[i], nums[last_small_pointer]

    return last_small_pointer + 1
```

```java
public void sortColors(int[] a) {
    partitionArray(a, 1);
    partitionArray(a, 2);
}

public int partitionArray(int[] nums, int k) {
    // 指向<k区间的最后一个元素
    int lastSmallPointer = -1;
    for (int i = 0; i < nums.length; i++) {
        // 如果nums[i] < k，需要交换并右移指针
        if (nums[i] < k) {
            // 指针先右移一位，指向当前元素应该去往的位置
            // 然后进行swap
            lastSmallPointer++;
            swap(nums, lastSmallPointer, i);
        }
    }
    return lastSmallPointer + 1;
}

private void swap(int[] a, int i, int j) {
    int tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
}
```

143 Sort Colors II 颜色分类 II

Given an array of *n* objects with *k* different colors (numbered from 1 to k), sort them so that objects of the same color are adjacent, with the colors in the order 1, 2, ... k.

A rather straight forward solution is a two-pass algorithm using counting sort. That will cost O(k) extra memory.

Can you do it O(logk) using extra memory?

给定一个有n个对象（包括k种不同的颜色，并按照1到k进行编号）的数组，将对象进行分类使相同颜色的对象相邻，并按照1,2，...k的顺序进行排序。

可直接**使用计数排序算法扫描两遍，但这样会花费O(k)的额外空间。你能否在O(logk)的额外空间的情况下完成**？

输入：

[3,2,2,1,4]

4

输出：

[1,2,2,3,4]

输入：

[2,1,1,2,2]

2

输出：

[1,1,2,2,2]

猜时间复杂度

O(n*k)

O(n^k)

O(nlogk)

O(klogn)

O(nlogn)

K = 1, O(1)

K = 2, O(N)

K = 3, O(N)

K = n, O(NlogN)

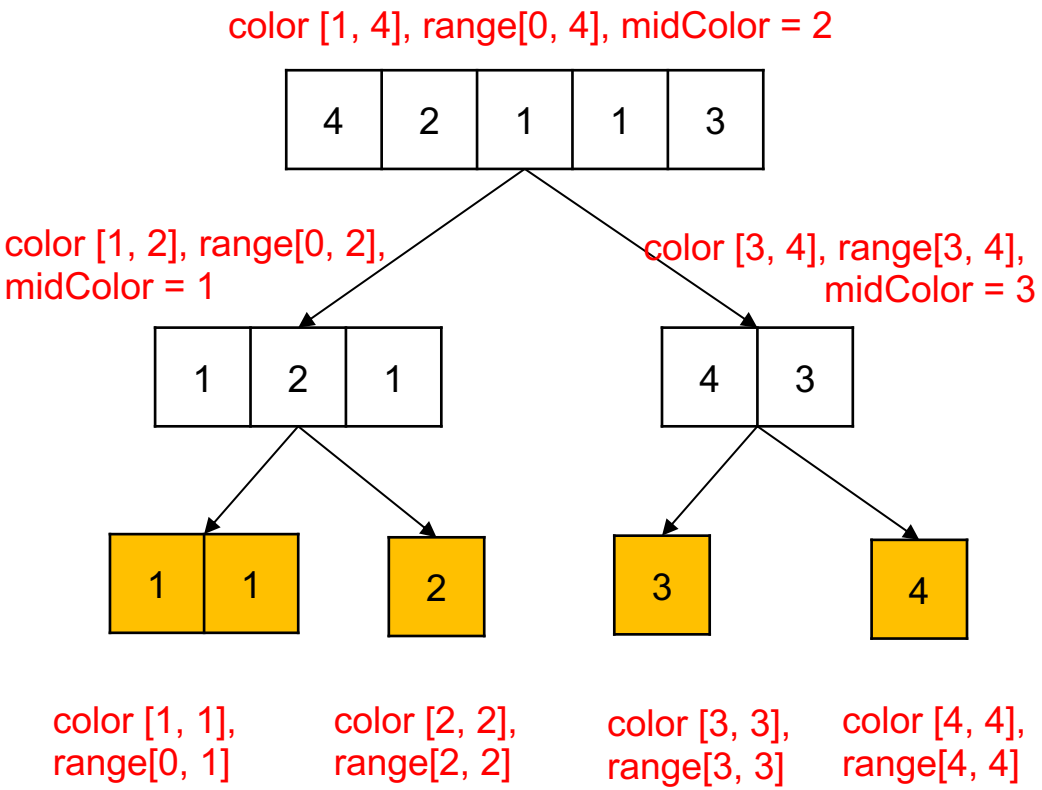
代码解析

类似于快速排序，在一个区间上对k种颜色的元素进行排序

先整体有序，再局部有序

利用分治算法思想，递归的程序设计方式

时间复杂度	O(NlogK)	N为数组长度，K为颜色个数
空间复杂度	O(logK)	K为颜色个数，logK为递归深度




```python
class Solution:
    def sortColors2(self, colors, k):
        # 特殊情况处理
        if not colors or len(colors) < 2:
            return
        self.sort(colors, 1, k, 0, len(colors) - 1)

    # 递归三要素之一：递归的定义
    def sort(self, colors, color_from, color_to, index_from, index_to):
        # 递归三要素之三：递归的出口
        # 如果这个范围内只有一个颜色，无需继续排序
        if color_from == color_to:
            return

        # 递归三要素之二：递归的分解
        #寻找中间色
        mid_color = (color_from + color_to) // 2

        #分区，左边区域小于等于中间色，右边大于中间色
        left, right = index_from, index_to
        while left <= right:
            while left <= right and colors[left] <= mid_color:
                left += 1
            while left <= right and colors[right] > mid_color:
                right -= 1
            if left <= right:
                colors[left], colors[right] = colors[right], colors[left]
                left += 1
                right -= 1
        # 继续在子区域内按照颜色进行排序
        self.sort(colors, color_from, mid_color, index_from, right)
        self.sort(colors, mid_color + 1, color_to, left, index_to)
```

代码解析

```java
public void sortColors2(int[] colors, int k) {
    // 特殊情况处理
    if (colors == null || colors.length < 2) {
        return;
    }
    rainbowSort(colors, 0, colors.length - 1, 1, k);
}
```

时间复杂度	O(NlogK)	N为数组长度，K为颜色个数
空间复杂度	O(logK)	K为颜色个数，logK为递归深度

```java
// 递归三要素之一：递归的定义
public void rainbowSort(int[] colors, int left, int right,
                        int colorFrom, int colorTo) {
    // 递归三要素之三：递归的出口
    // 如果这个范围内只有一个颜色，无需继续排序
    if (colorFrom == colorTo) {
        return;
    }

    // 递归三要素之二：递归的分解
    // 寻找中间色
    int colorMid = (colorFrom + colorTo) / 2;
    // 分区，左边区域小于等于中间色，右边大于中间色
    int l = left, r = right;
    while (l <= r) {
        while (l <= r && colors[l] <= colorMid) {
            l++;
        }
        while (l <= r && colors[r] > colorMid) {
            r--;
        }
        if (l <= r) {
            int temp = colors[l];
            colors[l] = colors[r];
            colors[r] = temp;
            l++;
            r--;
        }
    }
    // 继续在子区域内按照颜色进行排序
    rainbowSort(colors, left, r, colorFrom, colorMid);
    rainbowSort(colors, l, right, colorMid + 1, colorTo);
}
```

其他有趣的排序

烙饼排序 Pancake Sort（有可能会考哦）

https://en.wikipedia.org/wiki/Pancake_sorting

http://www.geeksforgeeks.org/pancake-sorting/

睡眠排序 Sleep Sort

https://rosettacode.org/wiki/Sorting_algorithms/Sleep_sort

面条排序 Spaghetti Sort

https://en.wikipedia.org/wiki/Spaghetti_sort

猴子排序 Bogo Sort

https://en.wikipedia.org/wiki/Bogosort

144 Interleaving Positive and Negative Numbers

Given an array with positive and negative integers. Re-range it to interleaving with positive and negative integers.
You are **not necessary to keep the original order** of positive integers or negative integers.

Do it in-place and without extra memory.

给出一个含有正整数和负整数的数组，重新排列成一个**正负数交错**的数组。

不需要保持正整数或者负整数原来的顺序。

完成题目，且不消耗额外的空间。

跟面试官核实：

1. 输入是否**有序**？ 不是
2. 有没有**重复**数字？ 有，但对本题没影响
3. 数据确保**正负数个数相差不超过1**
4. 不使用额外空间(do it in-place)
5. 不需要保持正整数或者负整数原来的顺序。

输入：

\- - + - + +
[-3, -2, 4, -1, 6, 5]

输出：

\- + - + - +
[-1, 5, -2, 4, -3, 6]

或者

\- + - + - +
[-1, 5, -3, 4, -2, 6]

或者

\+ - + - + -
[5, -1, 4, -3, 6, -2]

或者其他任何满足条件的答案

思路解析

可行解：先全部排序，再正负交错。

但是全部排序有必要吗？

没有必要花O(NlogN)时间全部排序。我们只需要花O(N)时间把正负数字分成左右两区，即可满足题意

已知数据确保正负数个数相差不超过1，正数多还是负数多有关系吗？ **有关系**

左边分区为**负数**，右边分区为**正数**，交换方式可以有多种

负多正少，Left = 1, right = length – 1, 间隔交换

负少正多，left = 0, right = length – 2, 间隔交换

正负相等，left = 0, right = length – 1, 间隔交换

代码解析


```python
def rerange(self, A):
    neg_cnt = self.partition(A)
    pos_cnt = len(A) - neg_cnt
    left = 1 if neg_cnt > pos_cnt else 0
    right = len(A) - (2 if pos_cnt > neg_cnt else 1)
    self.interleave(A, left, right)

def partition(self, A):
    left, right = 0, len(A) - 1
    while left <= right:
        while left <= right and A[left] < 0:
            left += 1
        while left <= right and A[right] > 0:
            right -= 1
        if left <= right:
            A[left], A[right] = A[right], A[left]
            left += 1
            right -= 1
    return left

def interleave(self, A, left, right):
    while left < right:
        A[left], A[right] = A[right], A[left]
        left, right = left + 2, right - 2
```

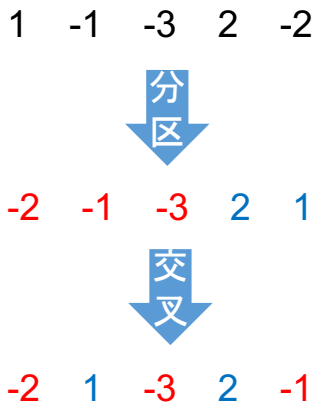
时间复杂度	O(N)	无需全部排序 只需正负分区，然后交错
空间复杂度	O(1)	In place

```java
public void rerange(int[] A) {
    int negCnt = partition(A);
    int posCnt = A.length - negCnt;
    // 根据正负数个数的情况设定两边交换的起点
    int left = negCnt > posCnt ? 1 : 0;
    int right = A.length - (posCnt > negCnt ? 2 : 1);
    interleave(A, left, right);
}

// 左分区为负，右分区为正
private int partition(int[] A) {
    int left = 0, right = A.length - 1;
    while (left <= right) {
        // 在左边找到一个正数
        while (left <= right && A[left] < 0) {
            left++;
        }
        // 在右边找到一个负数
        while (left <= right && A[right] > 0) {
            right--;
        }
        // 交换两边错位的数字，各就其位
        if (left <= right) {
            int temp = A[left];
            A[left] = A[right];
            A[right] = temp;

            left++;
            right--;
        }
    }
    return left;
}
```

```java
// 两端对调，实现正负交替
private void interleave(int[] A, int left, int right) {
    while (left < right) {
        int temp = A[left];
        A[left] = A[right];
        A[right] = temp;
        // 注意是+2，间隔交换
        left += 2;
        right -= 2;
    }
}
```



下节课预习材料

457 Classical Binary Search 经典二分查找问题

熟悉下节课题目的题意

我们一起努力~ϙ(´･ᴗ･`)

做作业

做ladder

群里提问题

看回放

课前预习

课后复习

刷题

互动课