作业 HW06 实验报告

姓名：闫浩扬 学号：2253156 日期：2023 年 12 月 28 日

1. 涉及数据结构和相关背景

本次作业涉及排序的相关数据结构和算法。排序作为各类数据结构的相应的运算的一种，在很多领域中都有广泛的应用。主要的排序方法有插入排序、交换排序、选择排序、二路归并排序、基数排序、外部排序等各类排序方法。堆排序、快速排序和归并排序是重难点，应深入掌握各种排序算法的思想、排序过程(能动手模拟)和特征(初态的影响、复杂度、稳定性、适用性等)。基本排序 $O(n^2)$ 优化排序 $O(nlogn)$ 特殊排序

(一) 相关概念

内部排序：若待排序记录都在内存中，称为内部排序。关注时间复杂度、空间复杂度。

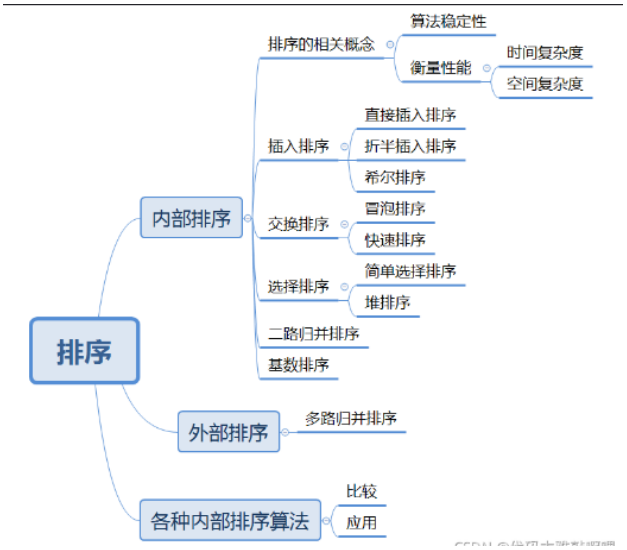
外部排序：若待排序记录一部分在内存，一部分在外存，则称为外部排序。外部排序时，要将数据分批调入内存来排序，中间结果还要及时放入外存，显然外部排序要复杂得多。关注 时间复杂度、空间复杂度、磁盘读写次数。

算法性能衡量：

① 时间效率：反映出排序速度（比较次数与移动次数）；

② 空间效率：反应出占内存辅助空间的大小；

③ 稳定性为 A 和 B 的关键字相等，排序后 A、B 的先后次序保持不变，则称这种排序算法是稳定的。



(二) 插入排序

1) 直接插入排序（基于顺序查找）

算法思想：整个排序过程为 n-1 趟插入，即先将序列中第 1 个记录看成是一个有序子序列，然后从第 2 个记录开始，逐个进行插入，直至整个序列有序。

算法分析：设对象个数为 n，则执行 n-1 趟，比较次数和移动次数与初始排列有关。

①最好情况下，每趟只需比较 1 次，不移动，总比较次数为 n-1；

②最坏情况下：第 i 趟比较 i 次，移动 i+1 次；

比较次数 $\sum_{i=2}^{n} i = \frac{(n+2)(n-1)}{2}$

移动次数 $\sum_{i=2}^{n} (i+1) = \frac{(n+4)(n-1)}{2}$

③若出现各种可能排列的概率相同，则可取最好情况和最坏情况的平均情况，时间复杂度为 O(n2)，空间复杂度为 O(1)，是一种稳定的排序方法。

2) 折半插入排序（基于折半查找）

算法思想：假设待排序的记录存放在数组 r[1 … n]中，r[1]是一个有序序列。

①循环 n-1 次,每次用折半查找法，查找 r[i] (i=2,…，n)在已排好序的序列 r[1..i-1]中的插入位置；②然后将 r[i]插入表长为 i-1 的有序序列 r[1 …i-1], 直到将 r[n]插入表长为 n-1 的有序序列 r[1…n-1] , 最后得到一个表长为 n 的有序序列。

算法分析：①折半插入排序的时间复杂度仍为 O(n2)；折半插入排序所需附加存储空间和直接插入排序相同，只需要一个记录的辅助空间，所以空间复杂度为 O(1)；②折半插入算法是一种稳定的排序算法。

算法特点：①因为要进行折半查找， 所以只能用于顺序结构，不能用于链式结构;②适合初始记录无序、n 较大时的情况。

3) 希尔排序（基于逐趟缩小增量）

算法思想：先将整个待排记录序列分割成若干子序列,分别进行直接插入排序，待整个序列中的记录"基本有序"时，再对全体记录进行一次直接插入排序。

算法技巧：子序列的构成不是简单地"逐段分割"，将相隔某个增量 dk 的记录组成一个子序列，让增量 dk 逐趟缩短（例如依次取 5,3,1），直到 dk＝1 为止。

算法分析：时间复杂度是 n 和 d 的函数，按照经验而言，时间复杂度一般为

$$O(n^{1.25}) \sim O(1.6n^{1.25})$$

空间复杂度为 o(1)；希尔排序是一种不稳定的排序方法。

(三) 交换排序

1) 冒泡排序

算法思想：每趟不断将记录两两比较，并按"前小后大" 规则交换

算法分析：设对象个数为 n，比较次数和移动次数与初始排列有关。

①最好情况下：只需 1 趟排序，比较次数为 n-1，不移动；

②最坏情况下：需 n-1 趟排序，第 i 趟比较 n-i 次，移动 3(n-i)次

$$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}(n^2-n)$$

$$3\sum_{i=1}^{n}(n-i)=\frac{3}{2}(n^2-n)$$

③时间复杂度为 o(n2)，空间复杂度为 o(1)；

④冒泡排序是一种稳定的排序方法。

2) 快速排序

算法思想：

①任取一个元素（如第一个）为中心；

②所有比它小的元素一律前放，比它大的元素一律后放，形成左右两个子表；

③对各子表重新选择中心元素并依此规则调整，直到每个子表的元素只剩一个。

算法分析：

①最好情况为划分后，左侧右侧子序列的长度相同；

②最坏情况为从小到大排好序，递归树成为单支树，每次划分只得到一个比上一次少一个对象的子序列，必须经过 n-1 趟才能把所有对象定位，而且第 i 趟需要经过 n-i 次关键码比较才能找到第 i 个对象的安放位置。

$$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}n(n-1)\approx\frac{n^2}{2}$$

③时间效率： 因每趟确定的元素呈指数增加，因此时间复杂度为$O(nlogn)$；空间效率：因递归要用到栈空间，因此空间复杂度为O（$logn$）。

④因可选任一元素为支点，所以快速排序算法为不稳定算法。

(四) 选择排序

1) 简单选择排序

算法思想：①设待排序的记录存放在数组 r[l… n]中。第一趟从 r[l]开始，通过 n-1 次比较，从 n 个记录中选出关键字最小的记录，记为 r[k], 交换 r[l]和 r[k]。②第二趟从 r[2]开始，通过 n-2 次比较，从 n-1 个记录中选出关键字最小的记录，记为 r[k],交换 r[2]和 r[k] 。③依次类推，第 l 趟从 r[i]开始，通过 n-i 次比较，从 n-i+l 个记录中选出关键字最小的记录，记为 r[k], 交换 r[i]和 r[k]。④经过 n-1 趟，排序完成。

算法分析： ①移动次数，在最好情况下为 0 次，最坏情况下为 3（n-1）次；

② 比较次数为 $\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}(n^2-n)$

③简单排序的时间复杂度：O(n²)，空间复杂度：O(1)；

④简单排序算法是稳定的排序算法。

2) 堆排序

算法思想： ①将无序序列建成一个堆，输出堆顶的最小（大）值，使剩余的 n-1 个元素又调整成一个堆，则可得到 n 个元素的次小值；②重复执行，得到一个有序序列。

算法分析：①堆排序的时间效率：O(nlog2n) ,空间效率：O(1);②堆排序算法因对元素的不断交换，因此该算法是不稳定的；③堆排序算法适用于 n 较大的情况。

(五) 归并排序

算法思想：初始序列看成 n 个有序子序列，每个子序列长度为 1：

①两两合并，得到⌊n/2⌋个长度为 2 或 1 的有序子序列；

②再两两合并，重复直至得到一个长度为 n 的有序序列为止。

算法分析：①二路归并排序短发的时间效率为$O(nlogn)$ ，空间效率为$O(n)$；

②该算法在比较过程中，遇到相同元素是不进行交换的，因此该算法是稳定的。

(六) 外部排序

排序方法	平均时间	比较次数		移动次数		稳定性	附加存储（空间复杂度）
		最好	最差	最好	最差		
直接插入	n2	n	n2	0	n2	√	1
折半插入	n2	nlog2n		0	n2	√	1
希尔排序	n1.3			0		×	1
冒泡排序	n2	n	n2	0	n2	√	1
快速排序	nlog2n	nlog2n	n2	nlog2n	n2	×	log2n
简单排序	n2	n2		0	n	√	1
堆排序	nlog2n	nlog2n		nlog2n		×	1
归并排序	nlog2n	nlog2n		nlog2n		√	n
基数排序	d(n+rd)					√	n+rd

2. 实验内容

2.1 求逆序对

2.1.1 问题描述

对于一个长度为 N 的整数序列 A，满足 i < j 且 Ai > Aj 的数对(i,j)称为整数序列 A 的一个逆序。请求出整数序列 A 的所有逆序对个数。

2.1.2 基本要求

输入：输入包含多组测试数据，每组测试数据有两行；第一行为整数 N(1 <= N <= 20000)，当输入 0 时结束；第二行为 N 个整数，表示长为 N 的整数序列

输出：每组数据对应一行，输出逆序对的个数

2.1.3 数据结构设计

这个问题第一选择可以使用冒泡排序，但是冒泡排序时间复杂度为 O(n^2),如果数据过大会非常容易超时。

其实可以使用归并排序来解决，归并排序是一种分治算法，它的基本思想是将原始数组分成若干个子数组，分别进行排序，然后再将这些子数组合并成一个有序数组。在归并排序的过程中，我们可以统计逆序对的个数。当合并两个有序数组时，如果左边的数组中的某个元素大于右边数组中的某个元素，那么左边数组中这个元素后面的所有元素都与右边数组中的这个元素构成逆序对。

而且归并排序有一个结论：如果在归并排序过程中，出现 $a[i]>a[j]$ ，那么就会产生 $mid-i+1$ 个逆序对。因为我们做归并排序是用到了分治的思想，最后的操作其实就是递归

回溯，从小到大地合并，所以这个时候，我们的两个子序列（即 $l-mid$ ， $mid+1-r$ 其实都是已经排好序的），这个时候，出现了一个不和谐的 a[i] ，说明从这个数一直到 $a[mid]$ 的所有数都是不和谐的。我们直接累加就好。

```
vector<int> a;      // 输入整数序列
long long sum;      // 逆序对的计数
vector<int> temp;   // 归并排序中的临时数组
```

2.1.4 功能说明（函数、类）

设计思想：使用归并排序的思想，通过分治法将问题分解为子问题，逐步求解并合并，同时在合并的过程中计算逆序对的个数。这样可以有效地降低时间复杂度，并通过临时数组暂存数据，控制空间复杂度。

例如：现在有两个部分：
[1 3 5 7 9] [2 4 6 8 10]
现在进行合并，对两个部分的第一个数进行比较。
因为 1 < 2，所以将 1 放入临时数组（也就是答案数组），这时临时数组只有一个元素
[1]
然后因为 3 > 2，所以将 2 放入临时数组，这时临时数组有两个元素
[2]
然后因为 3 < 4，所以将 3 放入临时数组，这是临时数组有三个元素
[123]
.....
如此反复，最后得到的临时数组有十个元素
[1 2 3 4 5 6 7 8 9 10]
在第三步之后，我们需要比较 4 和 5 的大小，因为 4 小于 5 ，又因为左半部分数组是有序（递增）的，所以左半部分剩下的所有元素均大于 4 ，也就是说左半部分剩下的所有数均可以和 4 构成逆序对，左半部分剩下元素 5 7 9 三个元素，所以对 4 而言，有三个逆序对（5，4）（7，4）（9，4），其余同理.....

所以按照上述过程构造 Merge 函数如下：

```
/* description: 合并两个有序数组，同时计算逆序对的个数
 * param1      left: 待合并数组的左边界
 * param2      mid: 待合并数组的中间位置
 * param3      right: 待合并数组的右边界
 * return     无返回值
 */
void merge(int left, int mid, int right) {
    int p1 = left, p2 = mid + 1, i = left;
    temp.resize(right - left + 1);
    // 合并两个有序数组，并计算逆序对的个数
    while (p1 <= mid && p2 <= right) {
        if (a[p1] <= a[p2])
            temp[i++ - left] = a[p1++];
        else {
            temp[i++ - left] = a[p2++];
            sum += mid - p1 + 1;  // 计算逆序对
        }
    }
    // 处理剩余元素
    while (p1 <= mid) temp[i++ - left] = a[p1++];
    while (p2 <= right) temp[i++ - left] = a[p2++];
    // 将临时数组复制回原数组
    for (i = left; i <= right; i++) a[i] = temp[i - left];
}
```

时间复杂度：
在合并的过程中，每个元素最多被复制两次（从原数组到临时数组，然后从临时数组回到原数组），因此复制的次数与数组的长度成正比。
在每一次递归中，都会进行一次合并操作，归并排序的递归深度是$log\ N$，综合考虑，合并的时间复杂度为$O(N\ log\ N)$
空间复杂度：
在合并的过程中，需要一个临时数组来存储部分排序结果，其大小与待合并的数组长度成正比，即$O(N)$。递归调用的过程中，每次调用都会创建一个新的临时数组。归并排序的递归深度是$log\ N$。综合考虑，空间复杂度为$O(N\ log\ N)$。
合并操作的时间复杂度主要影响整体算法的性能，而空间复杂度主要受递归调用深度的影响。在这个归并排序的实现中，空间复杂度比较高，因为在每次递归调用时都需要创建一个新的临时数组。

```
/* description: 递归调用，分治法实现归并排序
 * param1      left: 待排序数组的左边界
 * param2      right: 待排序数组的右边界
 * return     无返回值
 */
void find_pairs(int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2; //划分子序列
        find_pairs(left, mid); //递归调用，处理子片段
        find_pairs(mid + 1, right);
        merge(left, mid, right);//合并
    }
}
```

时间复杂度：
在每一次递归中，都会进行一次分割操作，将数组分为两部分。归并排序的递归深度是$log\ N$，综合考虑，递归调用的时间复杂度为$O(N\ log\ N)$。
空间复杂度：
递归调用的过程中，每次调用都会创建一个新的临时数组。归并排序的递归深度是$log\ N$。综合考虑，空间复杂度为$O(N\ log\ N)$。

2.1.5 调试分析（遇到的问题和解决方法）

主要利用了归并排序的特点来求解逆序对，将归并排序合并过程的分治思想用在了求解逆序对的数量上，将整体的时间复杂度控制在$O(nlogn)$相较于冒泡排序有较快速提升。

在实现过程中，最开始使用了全局数组，后来换成了 vector 发现超时，推测原来是将 tmp 数组声明放在了循环中，导致每次都会生成一个新的临时数组，这样占用空间且时间复杂度高。

2.1.6 总结和体会

这道题体会出了不同的排序方法有着不同的优点和适用场景，对于逆序对的求解，本题使用了经典的分治思想，时间复杂度控制在$O(nlogn)$，另一种方法是利用树状数组维护逆序对的数量。

2.2 最大数

2.2.1 问题描述

给定一组非负整数 nums，重新排列每个数的顺序（每个数不可拆分）使之组成一个最大的整数。

2.2.2 基本要求

注意：输出结果可能非常大，所以你需要返回一个字符串而不是整数。输入包含两行 第一行包含一个整数 n，表示组数 nums 的长度，第二行包含 n 个整数 nums[i].

2.2.3 数据结构设计

解题思路：

给定一组非负整数 nums，要求重新排列这些数的顺序，使其组成一个最大的整数，其中每个数不可拆分，为了构建最大数字，我们希望越高位的数字越大越好。为了实现这个目标，我们采用以下步骤：

①比较每个数的最高位，按最高位从小到大的顺序排列这些数。这一步可以通过将每个数转换为字符串，然后比较字符串的最高位实现。

②在比较最高位时，如果遇到最高位数字相同的情况，采用自定义比较方法：比较 字符串 1 + 字符串 2 与 字符串 2 + 字符串 1 的大小。选择较小的一组作为排序结果。

③如果最高位没有相同数字，那么根据最高位的排序就已经排好了。

另外，输出结果可能非常大，所以你需要返回一个字符串而不是整数。先把整数转换成字符串，然后在比较 a+b 和 b+a，如果 a+b>=b+a，就把 a 排在 b 的前面，反之则把 a 排在 b 的后面。

因为要实现字符串大小的比较，所以可以使用自定义比较器 cmp，用于对转化成字符串的整数进行比较。这个 cmp 结构体被用作 sort 函数的比较器，以确保排序的结果按照最高位从大到小排列。

```
struct {
    bool operator()(string a, string b) const {
         string s1 = a + b;
        string s2 = b + a;
        // 将两个字符串进行拼接并比较，返回拼接后的结果较大的那个。
        return s1 > s2;
          }
}cmp;
```

然后将最大数运算函数 largestNumber 放在 Solution 类中，在其中进行最大数的组合运算。

```
class Solution {
public:
    string largestNumber(std::vector<int>& nums) { } //最大数计算函数
};
```

2.2.4 功能说明（函数、类）

```
    /*
    * description: 重新排列输入向量中的数，以形成最大的整数。
    *
    * param1   nums: 一个非负整数向量。
    * return   string: 最大整数的字符串表示。
    */
    string largestNumber(std::vector<int>& nums) {
        判空
        vector<string> str; // 将每个整数转换为字符串并存储在一个向量中。
        for (int i = 0; i < nums.size(); i++)
            str.push_back(to_string(nums[i]));
        // 使用自定义比较器 cmp 对字符串向量进行排序，实现最大整数的拼接。
        sort(str.begin(), str.end(), cmp);
        // 将排序后的字符串按顺序拼接成最大整数的字符串表示。
        string ans = "";
        for (int i = 0; i < str.size(); i++)
            ans += str[i];
        // 检查最终结果是否以 '0' 开头，如果是，则返回 "0"，否则返回拼接后的字符串。
        if (ans[0] == '0')
            return "0";
        else
            return ans;
    }
```

设计思想：使用自定义比较器实现字符串的排序，按照题目规定的规则排列成最大的整数。将整数转换为字符串，然后根据自定义比较器排序，拼接排序后的字符串，得到最大整数。

时间复杂度：排序算法的时间复杂度为 O(n log n)，其中 n 是输入的整数数量。

空间复杂度：额外使用了一个字符串向量来存储转换后的整数，空间复杂度为 O(n)。

2.2.5 调试分析（遇到的问题和解决方法）

在调试过程中首先将首位直接进行排序，没有考虑到首位相同的情况以及不同位数的情况，在比较最高位时，如果遇到最高位数字相同的情况，采用自定义比较方法：比较 字符串 1 + 字符串 2 与 字符串 2 + 字符串 1 的大小。选择较小的一组作为排序结果。修正后将所有情况都分类考虑了。

2.2.6 总结和体会

数组的排序总是可以从数组中每两个元素之间的排序考虑起，如本题可以考虑到相邻两个元素之间如何进行排序的问题，将问题规模最小化。通过自定义比较器巧妙地解决了排序问题，易于理解和维护。时间复杂度和空间复杂度均在合理范围内，算法效率较高。

2.3 三数之和

2.3.1 问题描述

给你一个整数数组 nums ，判断是否存在三元组 [nums[i], nums[j], nums[k]] 满足 i != j、i != k 且 j != k ，同时还满足 nums[i] + nums[j] + nums[k] == 0 。请你返回所有和为 0 且不重复的三元组，每个三元组占一行。

2.3.2 基本要求

注意：答案中不可以包含重复的三元组。按三元组中最小元素从小到大的顺序依次打印每一组三元组。

2.3.3 数据结构设计

根据题意，我们要找到满足$nums[i] + nums[j] + nums[k] == 0$的三元组，那么如果 3 个数之和等于 0，那么这三个数有以下两种情况：①3 个数字的值都是 0；②3 个数字中有正数也有负数；

如果只寻找满足条件的三元组其实是很容易的，我们可以分步操作，先计算两数之和，在判断是否有第三个数加上满足和为零，记录满足条件的三元组，但是这样会产生很多重复的三元组，所以需要对三元组进行去重，一般的方法处理会显得非常繁杂，不好操作。这里使用先排序再使用双指针的方法是比较容易去除重复三元组的。

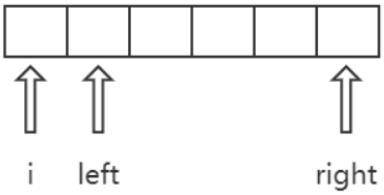
在分析完后，为了便于进行遍历计算，先将 nums 中的数字进行排序操作。然后我们通过指针 i 去遍历整个排序后的数组，与此同时，我们创建两个指针 left 和 right，left 指向 i+1 的位置，right 执行数组的末尾即 n-1 位置。为方便起见，我们将 left 和 right 所指向数字的目标和设为$target$，即 $-nums[i]$这样对于每个循环，i 的元素值是不变的，通过$nums[left] + nums[right]$，我们可以得到总和sum，然后我们将 sum 和 target 进行比较：

如果 sum 等于 target 则满足题目中的条件，将其放入到 result 中，后续作为返回值；

如果 sum 大于 target 说明目标值大了，我们需要向左移动 right 指针，因为这样 right 会变小，整体的 sum 值也会变小更加趋近于 target；

如果 sum 小于 target 我们需要向右移动 left 指针，因为这样 left 会变大，整体的 sum 值也会变大更加趋近于 0；

除了上面的判断逻辑，我们还需要注意去重的操作，也就是说，当我们移动 i 指针、left 指针或 right 指针的时候，如果发现待移动的位置数字与当前数字相同，那么就跳过去继续指向下一个元素，直到找到与当前数字不同的数字为止（当然，要避免数组越界）。在移动 left 指针和 right 指针的过程中，如果不满足 left<right，则结束本轮操作即可。



2.3.4 功能说明（函数、类）

```
/*
* description: 实现一个函数，找到数组中所有和为零的三元组，每个三元组包含不同的元素。
```

```
*
* param1   nums: 整数数组。
* return   vector<vector<int>>: 包含所有和为零的三元组的二维数组。
*/
vector<vector<int>> threeSum(vector<int>& nums) {
    vector<vector<int>> result; //储存结果
    int n = nums.size();
    sort(nums.begin(), nums.end());// 对输入数组进行排序，为后续双指针法做准备。

    for (int i = 0; i < n - 2; i++) {
        // 跳过重复元素，避免重复的三元组。
        if (i > 0 && nums[i] == nums[i - 1])
            continue;

        int target = -nums[i]; //left 和 right 之和的目标值
        int left = i + 1, right = n - 1; //初始化左右指针位置

        // 双指针法，在已排序的数组中查找两个元素使其和为目标值。
        while (left < right) {
            int sum = nums[left] + nums[right];

            if (sum == target) {
                // 找到一组解，添加到结果集中。
                result.push_back({ nums[i], nums[left], nums[right] });

                // 跳过重复元素，避免重复的三元组。
                while (left < right && nums[left] == nums[left + 1]) {
                    left++;
                }
                while (left < right && nums[right] == nums[right - 1]) {
                    right--;
                }

                // 移动指针。
                left++;
                right--;
            }
            else if (sum < target) {
                // 和小于目标值，左指针右移。
                left++;
            }
            else {
                // 和大于目标值，右指针左移。
                right--;
            }
        }
    }

    return result;
}
```

设计思想：使用双指针法，在排序后的数组中寻找和为零的三元组。首先对数组进行排序，便于后续去重，然后对于每一个 nums[i]使用两个指针（left 指针和 right 指针）分别从数组的两端向中间移动，寻找满足条件的三元组。
时间复杂度：排序数组的时间复杂度为 O(n log n)，遍历数组的时间复杂度为 O(n)，双指针法的时间复杂度为 O(n)。总体时间复杂度为 O(n log n) + O(n) * O(n) = O(n^2)。
空间复杂度：结果集的空间复杂度为 O(1)，因为只是返回结果而不使用额外的空间。排序算法的空间复杂度为 O(log n)。总体空间复杂度为 O(log n)。
特色：双指针法的巧妙运用，能够在排序后的数组中以线性时间内找到合为 0 的三元组，大大提高算法效率。同时排序后再用双指针法还可以巧妙地排除重复三元组，按照一定顺序输出。

2.3.4 调试分析（遇到的问题和解决方法）

巧妙运用了双指针法，通过使用双指针法，在排序后的数组中以线性时间内找到和为零的三元组，大大提高了算法的效率。对重复元素的处理，在遍历过程排序后数组的过程中，使用 if 语句判断 left 和 right 当前元素和他们下一个元素是否相同，如果相同则跳过的方法处理了重复元素的情况，避免了重复的三元组被计入结果集。

2.3.6 总结和体会

这道题对我有很大的启发，体会到了双指针法的优点，第一次学习双指针法是跑步套圈的问题，这次是两个指针从两端向中间逼近，通过指定中心元素 nums[i]，以及在逼近过程中，判断指针的下一个元素和当前元素是否相同，排除掉重复的三元组，这种方法十分巧妙。

3. 实验总结

这次实验学习了排序的相关算法，同时也在做题过程中理解了不同排序算法的适用情形和效率，有的算法不稳定，比如快速排序，在最坏情况可能还不如其他排序方法。另外第一题，使用冒泡排序，时间复杂度为 $O(n^2)$时间复杂度较高导致超时，换成效率更高的归并排序 O(nlogn).

这次实验是数据结构这门课的最后一次作业，这学期学到了很多有趣的知识，了解了各种各样的数据结构和算法，让我领略到算法之美，在之后我也会再深入学习其他有趣的数据结构与算法，将他们应用于实际问题中去。