作业 HW6* 实验报告

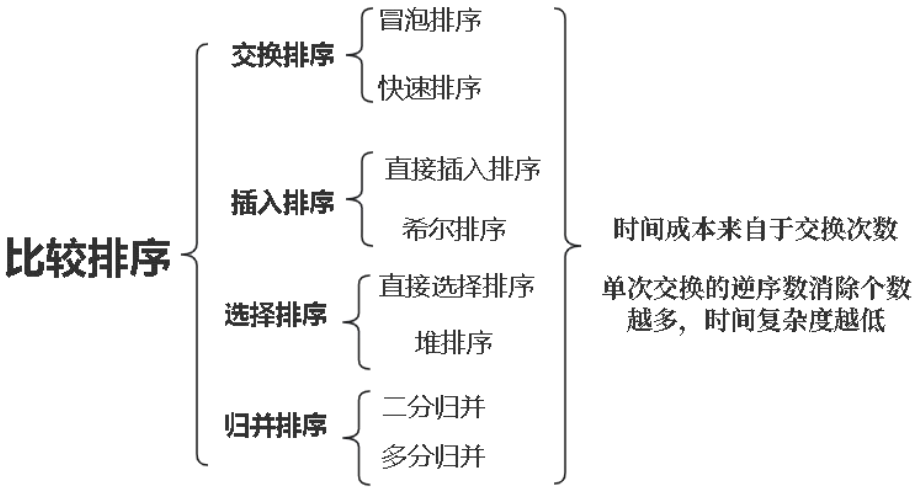
姓名：朱俊泽 学号：2351114 日期：2024 年 12 月 16 日

1. 涉及数据结构和相关背景

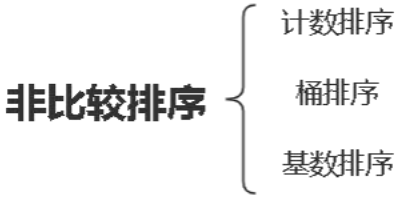
排序问题的核心是比较+交换，各个排序问题都具有不同的时间复杂度与空间复杂度。
从逆序对(Inversion)的角度来看：

排序 ⇔ 消除逆序对数目

其中，逆序对表示在序列中的两个元素x_i与x_j，若满足：$x_i > x_j$且$i < j$，则$< x_i , x_j >$构成一对逆序对。不同的排序算法在不同排序序列中可能发挥的效果不同。事实上，并没有绝对的最优的排序算法，要基于实际任务情况进行选择。常见的比较排序算法可分为以下几类：



理解比较排序的排序效率关键在于用估算每种方法每次进行交换时消除逆序对数平均个数。
除了比较排序，还有常见的非比较排序方法。但是注意，这些非比较排序的方法实质是借助数字数值特征分配对应的顺序存储地址，再根据地址顺序取出数字从而实现排序的。因此非比较排序往往有比较大的局限性，比较排序基本上都只能适用于整数；此外，如计数排序，还会耗用巨大的存储空间，带来超大的空间复杂度。



2. 实验内容

2.1 题目一 排序 （针对本题要的多种排序算法，我只交了一种，剩下的在其他题目报告完毕后再做陈述）

2.1.1 问题描述

2.1.2 基本要求

2.2 题目二 求逆序对数

2.2.1 问题描述

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

2.2.2 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

2.2.3 数据结构设计

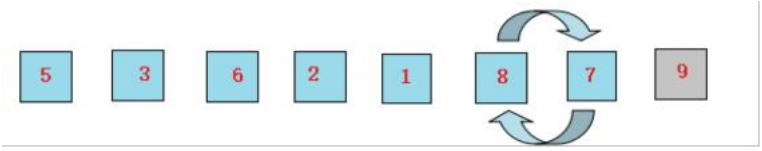
本题涉及的逆序对的概念是理解一切比较排序算法效率优劣的核心。虽然在所有算法中都涉及到逆序对的消除，但是并非所有算法都能比较容易的统计出来每次比较、交换操作所消除的逆序对的数目。

下面对冒泡、归并算法的过程进行分析，来判断他是否能用于求解逆序对数。能否用于统计逆序对的条件如下：

1)所有逆序对均实现交换且仅被交换一次。这一点对于所有排序算法均成立。

2)每次交换时交换的逆序对个数可获知。

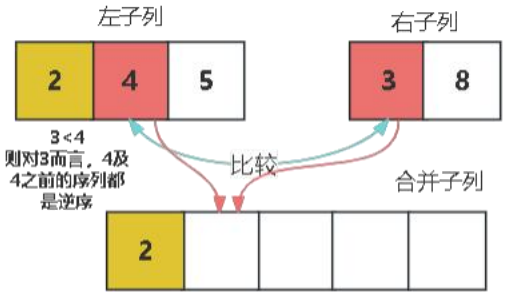
冒泡排序的核心思想是将每一对逆序元素进行交换，如下图：



每次进行一次交换就可以发现一个逆序对。

每次交换时都仅交换了一个逆序对，因此冒泡排序可以用于统计逆序对数。

归并排序。归并排序的比较、交换环节发生在合并的步骤。对于合并过程，实际上是两个从大到小已经排好的顺序序列进行合并。最后我们使用 nlogn 的归并排序

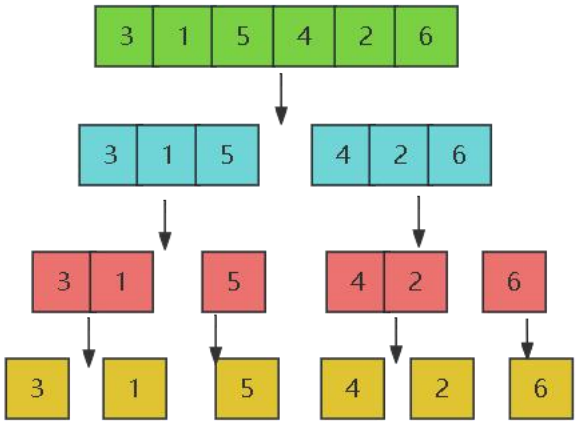


2.2.4 功能说明（函数、类）

归并算法实现分为两部分：第一是二分，第二是合并。

（1）二分

二分的模拟过程如下：



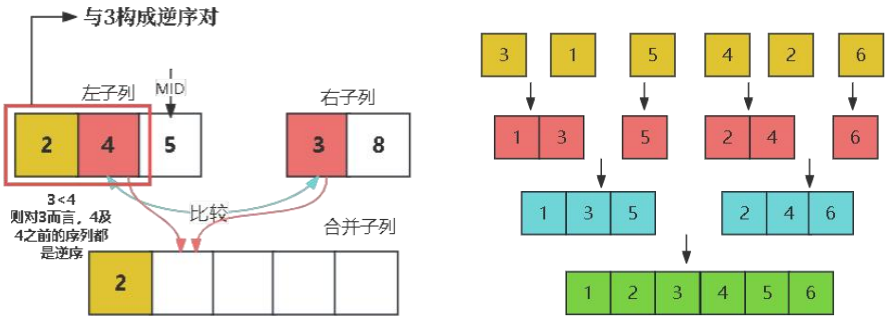
这个过程针对长度为 n 的串的复杂度就是 logn 的复杂度。

```
void MergeSort(int Seq[MAX_LEN],int left,int right,int &num,int temp[MAX_LEN]) {
    if (left < right) {
        int mid = (left + right) / 2;

        MergeSort(Seq, left, mid, num, temp);
        MergeSort(Seq, mid + 1, right, num, temp);
        Merge(Seq, left, right, num, temp);
    }
    return;
}
```

（2）合并

接下来两侧 sort 二分拆开后就要执行 merge 合并之后再进行合并操作：合并的操作需要借助一个中间数列 temp，将合并后的过程暂时保存，合并后将 temp 复制回原数列。合并过程就是依次把两个子列中更小的值放到 temp 中。 注意，在合并的过程中，由于具有默认的左右子列顺序，因此在实际的原序列中，右子列的值的下标都大于左子列的下标:



如图：

```
void Merge(int Seq[MAX_LEN], int left, int right, int &num,int
    int mid = (left + right) / 2;
    int i = left;
    int j = mid  + 1;
    int tmp = 0;
    while (i <= mid &&j <=right) {
        if (Seq[i] <= Seq[j]) {
            temp[tmp++] = Seq[i++];
        }
        else {
            temp[tmp++] = Seq[j++];

            num+=mid+1-i;
        }
    }

    while (i <= mid) {
        temp[tmp++] = Seq[i++];
    }

    while (j <= right) {
        temp[tmp++] = Seq[j++];
    }

    for(int i=0;i<tmp;i++){
        Seq[left+i] = temp[i];
    }
```

2.2.5 调试分析（遇到的问题和解决方法）

2.2.6 总结和体会

在本题虽然考察的是逆序对数的统计，但是注意到逆序对数又是排序中的重要概念。因此要从排序的角度出发来解决此问题。此外，也要注意到，归并排序在实际操作的过程中需要开辟一个新的 temp 数列，因此空间复杂度是 O(n)。

2.3 题目三 三数之和

2.3.1 问题描述

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。

输入描述

2.3.2 基本要求

输入描述

2 行
第 1 行一个整数，表示数组元素个数;
第 2 行输入一组整数，中间以空格分隔。

输出描述

输出所有和为 0 的三元组，每个三元组一行，中间以空格分隔。
对于每一个三元组，你需要按从小到大的顺序依次返回三个元素;
对于所有三元组，你需要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。

注意：答案中不可以包含重复的三元组。

2.3.3 数据结构设计

本题最重要是解决两个问题:
① 如何按照一个合理的顺序遍历所有数列Seq中所有的三数组合
② 如何去除重复解
对于问题①:

这里按照一种从小到大的顺序进行 3 数的遍历。首先选定一个$pivot$，作为第一个基准。这里$pivot$是从第一个数开始，从第一个数一直到倒数第二个;

之后借助双指针i,j移动来寻找剩下两个数，而i则从数列$[pivot,\ right]$的序列的$pivot\ +1$ 开始，依次往右移动;j则从$right$开始依次向左移动。移动条件如下:

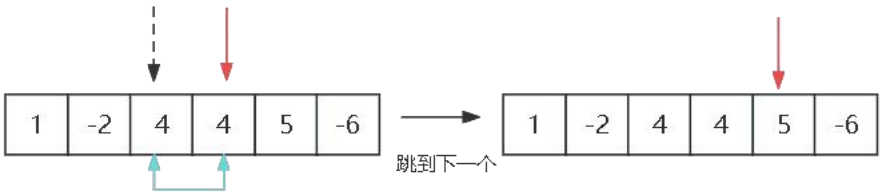
如果出现$Seq[pivot]+\ Seq[i]+\ Seq[j]$==0，则记录三数，并将$j$向左移动一次;

如果出现$Seq[pivot]+\ Seq[i]+\ Seq[j]$>0，说明大了，应当$j$指向的最大的数减小一点，因此将$j$向左移动一次;

如果出现$Seq[pivot]+\ Seq[i]+\ Seq[j]$<0，说明小了，应当$i$指向的最小的数减大一点，因此$i$向右移动一次;

对于问题②:

重复解出现的原因是因为序列中有重复的数，而在已经排序好的序列中，重复的数都是紧邻的，因此这里只需要在每次指针移动的时候判断一下前后位置的元素是否相同，如果相同则跳过到下一个即可，直到前后位置元素不同。



2.3.4 功能说明（函数、类）

对于本题使用一个快速排序:

QuickSort 排序的算法流程如下:

① 选择序列的第一个元素为枢轴元素，并用 middle 指针指向该枢轴元素的位置。以下步骤的目的是将序列调整顺序，直到枢轴元素左侧的元素的值都小于枢轴元素，枢轴元素右侧的所有元素的值都大于枢轴元素。

② 头尾 i，j 指针开始移动。

i)先移动尾指针 j，从尾向头移动，直到指向一个小于枢轴元素的值;

ii)之后开始移动头指针 i，从头向尾移动，直到指向一个大于枢轴元素的值;

iii)执行完成 i)、ii)后，交换 i，j 指针所指向位置的元素;

重复以上 i)、ii)、iii)步骤，直到 i，j 指针重合，交换 i（或 j）指向位置与 middle 指针指向的枢轴元素。在该过程中 iii)步骤可能一直不会执行;

③ 执行完毕①、②后，枢轴元素已经调整到序列的中间部分，其左侧的元素都比枢轴元素更小，其右侧的元素都不枢轴元素更大。这时候把两端的序列也当成新序列，
重新传入 QuickSort（）函数进行排序（分治思想），每次重复执行①、②步骤;

④ 直到最终传入数列的长度为 1，结束排序。

```
void QuickSort(int arr[MAX_SIZE], int low, int high) {
    if (low < high) {
        int pivotIndex = Partition(arr, low, high);
        QuickSort(arr, low, pivotIndex - 1);
        QuickSort(arr, pivotIndex + 1, high);
    }
}
```

然后找符合条件的三个数字

```
void FindTriplets(int arr[MAX_SIZE], int left, int right) {
    int pivot = left;
    int i = pivot;
    int j = right;

    while (pivot < right) {
        if (pivot > 0) {
            while (pivot > left && arr[pivot] == arr[pivot - 1]) {
                pivot++;
            }
        }

        j = right;
        i = pivot + 1;

        while (i < j) {
            int sum = arr[pivot] + arr[i] + arr[j];
            if (sum == 0) {
                SaveTriple(arr[pivot], arr[i], arr[j]);
                j--;
                while (j < right && arr[j] == arr[j + 1]) {
                    j--;
                }
            } else if (sum > 0) {
                j--;
            } else {
                i++;
            }
        }
        pivot++;
    }
}
```

2.3.5 调试分析（遇到的问题和解决方法）

2.3.6 总结和体会

本题在寻找三数的过程中运用了双指针的思想，这在快速排序中也用到了相似的方法。事实上双指针的方法并不仅仅局限于这里，双指针更重要的是实现两个变量的遍历过程。在求解相交链表中也用到过，这也是因为要同时遍历两个链表。应当着重体会这种双指针的作用。

2.4 题目四 最大数

2.4.1 问题描述

题目描述

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。
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。

2.4.2 基本要求

输入描述

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]
对于 100%的数据，1<=nums.size()<=100，0<=nums[i]<=10^9;

输出描述

输出包含一行，为重新排列后得到的数字

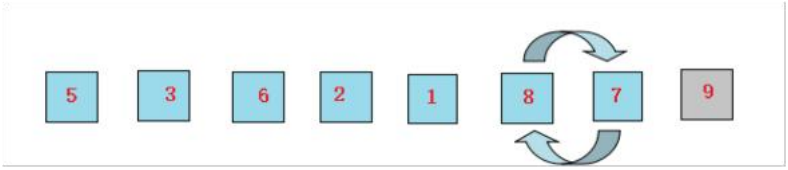
2.4.3 数据结构设计

题目包含一个想法就是比较拼接方法，拼接方法可以从一个最优转化到另一个最优。

1.完全性
我们随机选择的两个元素a,b，一定满足下面条件中的一个
a<=b或者a>=b,也就是说可以进行大小的比较
2.反对称性
两个元素a,b，满足a<=b并且a>=b可以推导出a==b
也就是a,b的前后顺序无所谓，因为排序之后是唯一确定的。
3.传递性
任意三个元素a,b,c
如果a>=b并且b>=c我们可以推导出a>=c;

也就是说，这个题目可以比较做成比较的方法，从每一个局部最优转化出结果最优。我们考虑一个冒泡排序即可：

冒泡排序的核心思想是将每一对逆序元素进行交换，如下图：



2.4.4 功能说明（函数、类）

冒泡排序的实现：但是我们希望求解最优值，那么就是让 a+b 大于 b+a 的往前放。因此实现下列的冒泡排序。

```
private:
    void bubbleSort(vector<string>& numStrs) {    vector 不是模板
        int n = numStrs.size();
        for (int i = 0; i < n - 1; ++i) {
            for (int j = 0; j < n - i - 1; ++j) {
                // 比较 a + b 和 b + a
                if (numStrs[j] + numStrs[j + 1] < numStrs[j + 1] + numStrs[j]) {
                    swap(numStrs[j], numStrs[j + 1]);    未定义标识符 "swap"
                }
            }
        }
    }
};
```

整个 largestNumber 函数实现如下：

```
string largestNumber(vector<int>& nums) {    未定义标识符 "string"
    // 将数字转换为字符串
    vector<string> numStrs;    未定义标识符 "vector"
    for (int num : nums) {
        numStrs.push_back(to_string(num));    未定义标识符 "to_string"
    }

    // 自定义排序规则的冒泡排序
    bubbleSort(numStrs);

    // 拼接结果
    string result;    应输入“;”
    for (const string &s : numStrs) {    "string" 不是类型名
        result += s;    未定义标识符 "result"
    }

    // 特殊情况处理：如果结果是 "000...0"，则返回 "0"
    if (result[0] == '0') {    未定义标识符 "result"
        return "0";
    }

    return result;
}
```

2.4.5 调试分析（遇到的问题和解决方法）

2.4.6 总结和体会

整个题目的难点都在考虑这个题目实际上是考特殊的转换，不过本题目出现在章节“排序”中，也就不难想出 a+b 和 b+a 之间的比较作为排序交换条件，使用冒泡法。

2.3 题目五 序列

2.5.1 问题描述

题目描述：

给定 m 个数字序列，每个序列包含 n 个非负整数。我们从每一个序列中选取一个数字组成一个新的序列，显然一共可以构造出 n^m 个新序列。接下来我们对每一个新的序列中的数字进行求和，一共会得到 n^m 个和，请找出最小的 n 个和

2.5.2 基本要求

输入格式:

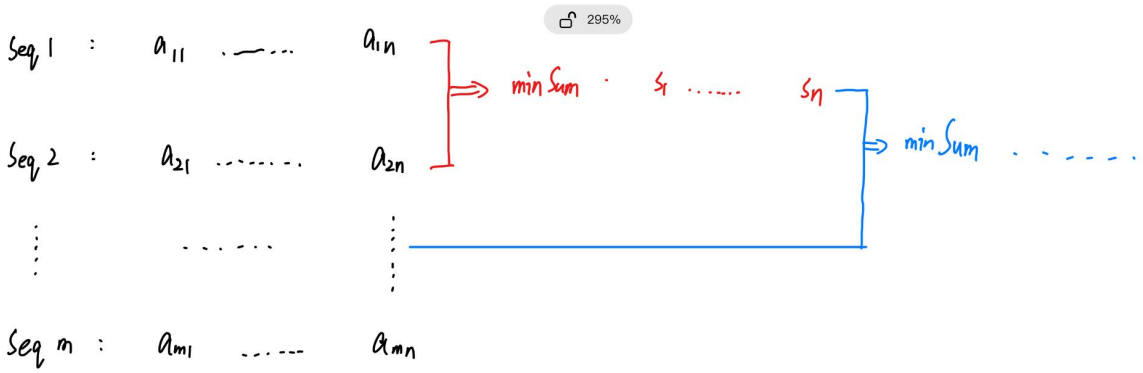
输入的第一行是一个整数 T，表示测试用例的数量，接下来是 T 个测试用例的输入
每个测试用例输入的第一行是两个个正整数 m（0 < m <= 100）和 n(0 < n <= 2000),
然后有 m 行，每行有 n 个数，数字之间用空格分开，表示这 m 个序列
序列中的数字不会大于 10000

输出格式：

对每组测试用例，输出一行用空格隔开的数，表示最小的 n 个和

2.5.3 数据结构设计

对本题来说，有个想法就是是否 m 个序列，可以从两个序列的最小值转化来得到 m 个序列选值的最小值，又是和上题一样的思路：局部最优转移到全局最优。

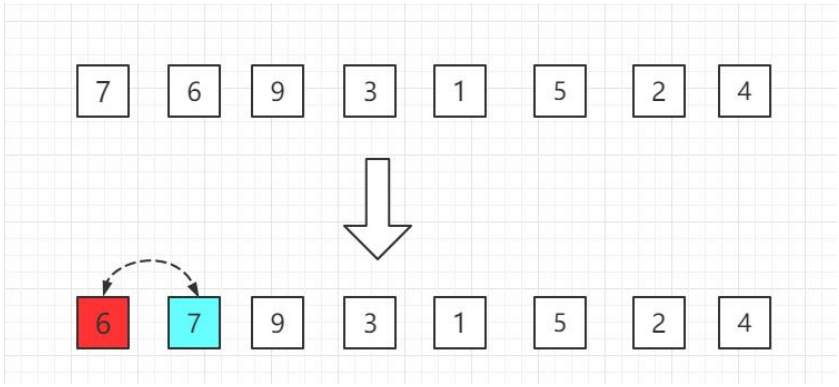


因此可以进行 m 次最小值的计算。

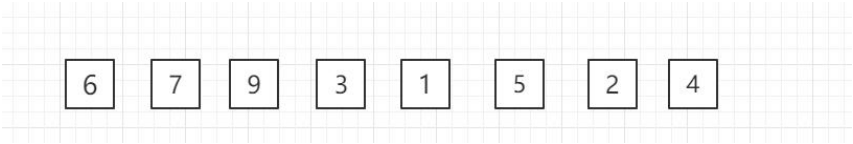
2.5.4 功能说明（函数、类）

可以考虑执行 m 次判断两个相邻序列的最小和问题，那么就要先执行排序，这里选择了插入排序

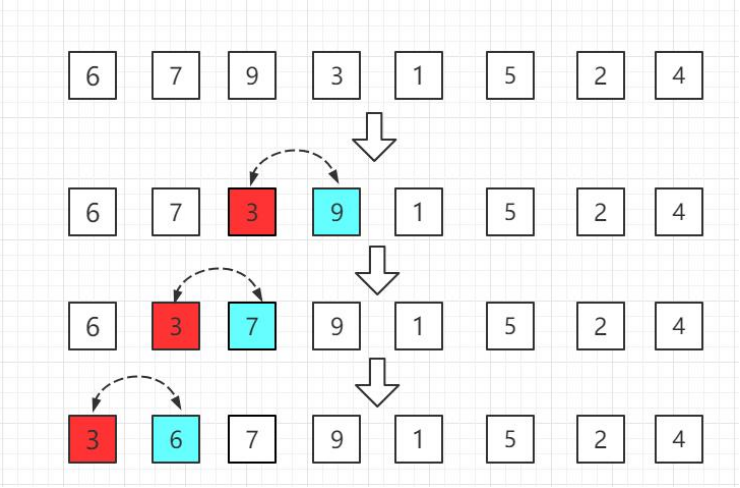
第一轮：从第二位置的 6 开始比较，比前面 7 小，交换位置。



第二轮：第三位置的 9 比前一位置的 7 大，无需交换位置。



第三轮：第四位置的 3 比前一位置的 9 小交换位置，依次往前比较。



第四轮：第五位置的 1 比前一位置的 9 小，交换位置，再依次往前比较

……

就这样依次比较到最后一个元素。

```c
// 插入排序实现
void insertionSort(int* array, int length) {
    for (int i = 1; i < length; ++i) {
        int keyValue = array[i];
        int position = i - 1;

        // 将大于 keyValue 的元素向右移动
        while (position >= 0 && array[position] > keyValue) {
            array[position + 1] = array[position];
            position--;
        }
        array[position + 1] = keyValue;
    }
}
```

对于计算 min 数组的方法，就是从小到大比较两个数组的加和。

```cpp
void maintainSum(int* currentSums, int* newArray, int length) {
    priority_queue<int> minHeap;

    for (int i = 0; i < length; ++i) {
        for (int j = 0; j < length; ++j) {
            int sumValue = currentSums[i] + newArray[j];

            if (minHeap.size() < length)
                minHeap.push(sumValue);
            else if (sumValue < minHeap.top()) {
                minHeap.pop();
                minHeap.push(sumValue);
            } else {
                break;
            }
        }
    }

    for (int k = length - 1; k >= 0; k--) {
        currentSums[k] = minHeap.top();
        minHeap.pop();
    }
}
```

总体逻辑:

先输入一个数组然后插入排序, 之后每一次输入一个数组都把他和当前这个数组做最小序列和的计算，然后传递给下一个数组。

```cpp
// 使用插入排序
insertionSort(currentSums, numElements);

for (int i = 1; i < numSequences; i++) {
    for (int j = 0; j < numElements; j++)
        cin >> newArray[j];

    // 使用插入排序
    insertionSort(newArray, numElements);
    maintainSum(currentSums, newArray, numElements);
}

for (int i = 0; i < numElements; ++i)
    cout << currentSums[i] << " ";

cout << endl;
```

2.5.5 调试分析（遇到的问题和解决方法）

2.5.6 总结和体会

总体逻辑不难想到，但是对传递的证明思路多多学习，此类排序的变式题，排序的拓展题都很重要。

3. 实验总结

本次实验实现了冒泡排序、选择排序、快速排序、归并排序、堆排序、希尔排序和插入排序等经典算法深刻理解了每个算法的优劣和适用场景。

冒泡排序和选择排序虽然简单，但在大规模数据上性能较差，尤其是冒泡排序。插入排序在小规模数据上表现良好，但对于大规模数据效率较低。希尔排序通过引入步长，对插入排序进行改进，具有较好的性能。

快速排序通过分而治之的思想，对于大规模数据具有较高的效率，是一种常用的排序算法。归并排序同样采用分治策略，适用于大规模且链式存储的数据。

堆排序通过二叉堆的数据结构，实现了不稳定排序，对于大规模数据的堆排序具有较好的性能。综合实验体会，理解了排序算法的时间复杂度、空间复杂度和稳定性等特性。在实际应用中需要根据具体场景选择合适的排序算法以达到最佳性能。

此外，本章的题目还考虑了很多的排序的变形题目，通过改变排序的判定条件等。比如第四第五题目。

4. 排序

4.1 快速排序

本章的题目三数之和中有使用，不再赘述。

4.2 归并排序

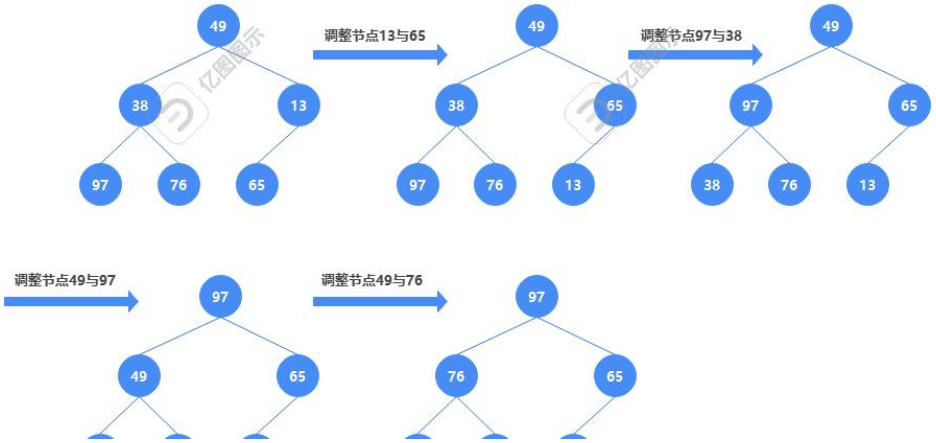
本章的题目求逆序对数中已经使用，不再赘述。

4.3 堆排序

.堆是一种完全二叉树数据结构，分为两种类型：

1. 最大堆：每个节点的值（只有非叶子节点才有子节点）都不小于其子节点的值。根节点是最大值。
2. 最小堆：每个节点的值（只有非叶子节点才有子节点）都不大于其子节点的值。根节点是最小值。

我们从最后一个非叶子节点开始调整。



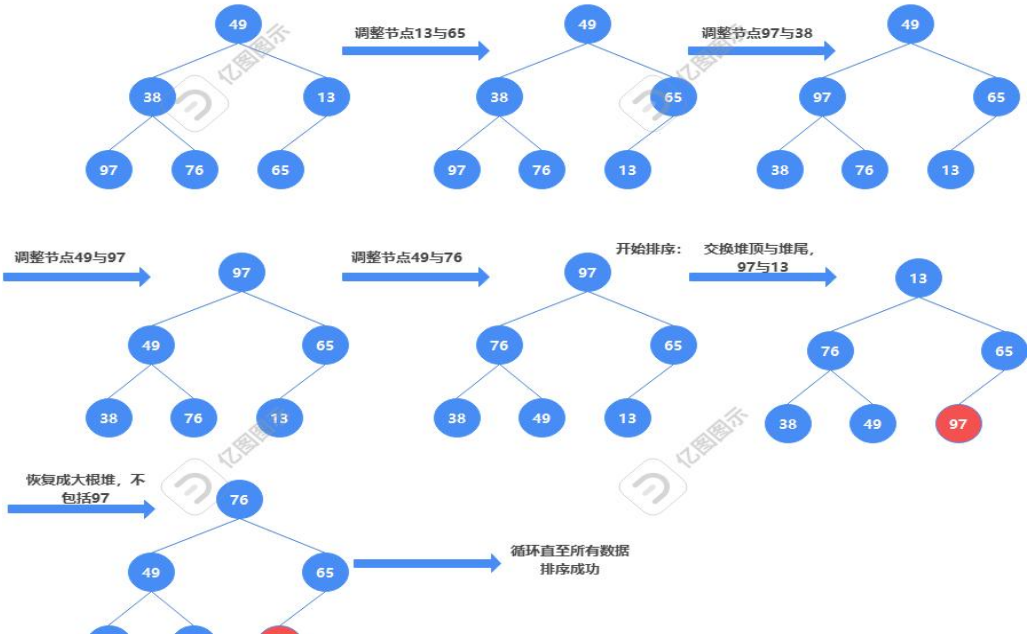
建堆过程解释：

1.最后一个非叶子节点为 13，13 的儿子节点 65，13<65，交换这两个节点。

2.其次非叶子节点为 38，38 的儿子左节点 97 比其儿子右节点 76 大，38<97，交换这两个节点。

3.其次为非叶子节点 49，49 的儿子左节点 97 比其儿子右节点 65 大，49<97，交换这两个节点。

4.非叶子节点 49，49 的儿子左节点 38 比其儿子右节点 76 小，49<76，交换这两个节点。




```
//构造大根堆（通过新插入的数上升）
void heapInsert(std::vector<int>& arr) {
    for (int i = 0; i < arr.size(); i++) {
        //当前插入的索引
        int currentIndex = i;
        //父结点索引
        int fatherIndex = (currentIndex - 1) / 2;
        //如果当前插入的值大于其父结点的值,则交换值，并且将索引指向父结点
        //然后继续和上面的父结点值比较，直到不大于父结点，则退出循环
        while (arr[currentIndex] > arr[fatherIndex]) {
            //交换当前结点与父结点的值
            swap(arr, currentIndex, fatherIndex);
            //将当前索引指向父索引
            currentIndex = fatherIndex;
            //重新计算当前索引的父索引
            fatherIndex = (currentIndex - 1) / 2;
        }
    }
}
```

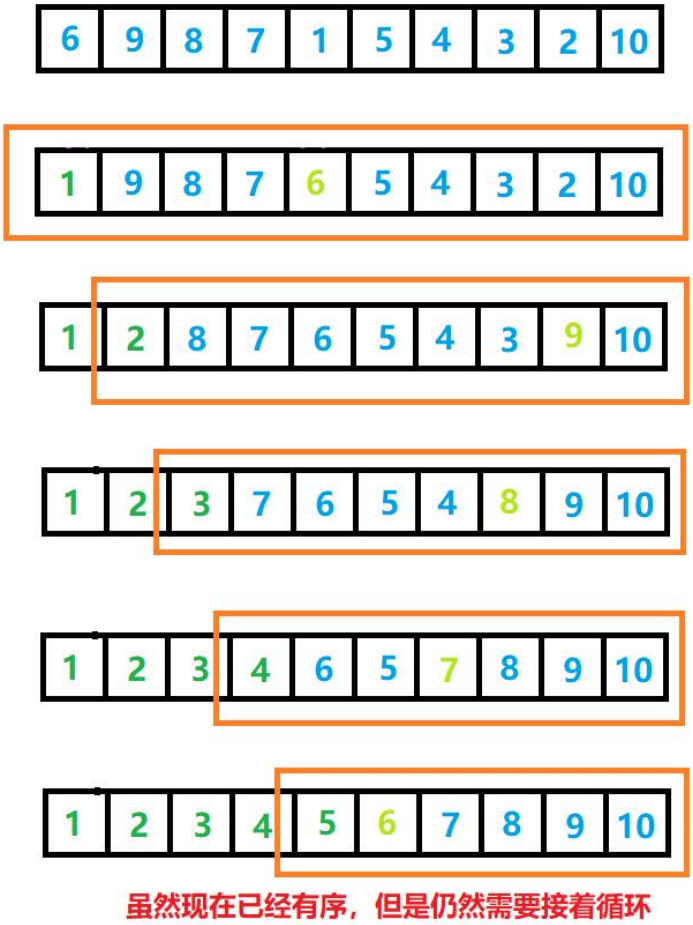
```
//将剩余的数构造成大根堆（通过顶端的数下降）
void heapify(std::vector<int>& arr, int index, int size) {
    int left = 2 * index + 1;
    int right = 2 * index + 2;
    while (left < size) {
        int largestIndex;
        //判断孩子中较大的值的索引（要确保右孩子在size范围之内）
        if (arr[left] < arr[right] && right < size) {
            largestIndex = right;
        }
        else {
            largestIndex = left;
        }
        //比较父结点的值与孩子中较大的值，并确定最大值的索引
        if (arr[index] > arr[largestIndex]) {
            largestIndex = index;
        }
        //如果父结点索引是最大值的索引，那已经是大根堆了，则退出循环
        if (index == largestIndex) {
            break;
        }
        //父结点不是最大值，与孩子中较大的值交换
        swap(arr, largestIndex, index);
        //将索引指向孩子中较大的值的索引
        index = largestIndex;
        //重新计算交换之后的孩子的索引
        left = 2 * index + 1;
        right = 2 * index + 2;
    }
}
```

4.4 选择排序

从数组中选出最小的元素，放在数组的起始位置。然后就相当于起始位置的元素就是正确的位置，也就相当于需要排序的部分减少了一个元素。由此循环就可以实现排序整个数组：

内层循环需要确定出当前待排序部分中最小的元素，并将其下标记录下来。每次内层循环后，将最小元素与待排徐部分的起始元素交换；

外层循环需要缩小待排序部分的大小。从数组的大小开始，直到待排序部分为 0，即整个数组排序完毕：




```cpp
void SelectSort(std::vector<int>& nums) {
    int min = 0;
    int min_index = 0;
    int tmp = 0;
    for (int i = 0; i < nums.size()-1; i++) {
        min = MAX_INT;
        for (int j = i; j < nums.size(); j++) {
            if (nums[j] < min) {
                min = nums[j];
                min_index = j;
            }
        }
        tmp = nums[min_index];
        nums[min_index] = nums[i];
        nums[i] = tmp;
    }
}
```

4.5 冒泡排序

本章的题目求逆序对数中已经讨论，不再赘述。

4.6 插入排序

本章的题目序列中已经讨论，不再赘述。

4.7 希尔排序

希尔排序本质是分段的插入排序，定义了一个 Gap 值，代表分段的个数。Gap=n/2，每一轮 Gap 都要除 2，直到除到 1 为止，代表排列完成所有元素。

```
void ShellSort(std::vector<int>& nums) {
    int length = nums.size();
    int j;
    int t;
    int tmp;
    int cnt = 0;  //记录在每组中进行插入的时候的位置数
    for (int gap = length / 2; gap >= 1; gap=gap / 2) {  //gap代表组的个数
        for (int i = 0; i < gap; i++) {   //对每组内部依次进行希尔排序
            j = i+gap;  //j在一开始指向每组的第二个元素
            cnt = 1;
            while (cnt < length / gap) {     //length/gap代表每组内的元素个数
                t = j;
                //组内通过交换法进行插入排序
                while (t-gap >= 0 && nums[t] < nums[t - gap]) {
                    tmp = nums[t];
                    nums[t] = nums[t - gap];
                    nums[t - gap] = tmp;
                    t = t - gap;
                }
                cnt++;
                j += gap;
            }
        }
    }
}
```