作业 HW6 实验报告

姓名：郑博远 学号：2154312 日期：2022 年 12 月 9 日

1. 涉及数据结构和相关背景

本次作业涉及到的数据结构相关知识为排序。排序 (Sorting) 是计算机程序设计中的一种重要操作,它的功能是将一个数据元素(或记录)的任意序列,重新排列成一个按关键字有序的序列。为了查找方便，通常希望计算机中的表是按关键字有序的。因为有序的顺序表可以采用查找效率较高的折半查找法，而无序的顺序表只能进行顺序查找，因此顺序表有序能够降低平均查找长度。因此,学习和研究各种排序方法是计算机工作者的重要课题之一。

排序的具体定义如下:

假设含 n 个记录的序列为 $\{R_1, R_2, \dots, R_n\}$，其相应的关键字序列为 $\{K_1, K_2, \dots, K_n\}$,需确定 1, 2, …, n 的一种排列 $p_1, p_2, \dots, p_n$ 使得其关键字满足如下的非递减关系 $K_{p_1} \le K_{p_2} \le \cdots \le K_{p_n}$，即使得原序列成为一个按照关键字有序的序列 $\{R_{p_1}, R_{p_2}, \dots, R_{p_n}\}$。这样的操作称为排序。

上述排序定义中的关键字 K，可以是记录 $R_i (i = 1, 2, \dots, n)$ 的主关键字，也可以是记录 R 的次关键字，甚至是若干数据项的组合。若 K 是主关键字,则任何一个记录的无序序列经排序后得到的结果是唯一的;若 K 是次关键字,则排序的结果不唯一，因为待排序的记录序列中可能存在两个或两个以上关键字相等的记录。假设 $K_i = K_j$ $(1 \le i \le n, 1 \le j \le n)$，且在排序前的序列中 R_i 领先于 R_j (即 i > j)。若在排序后的序列中 R_i 仍领先于 R_j，则称所用的排序方法是稳定的；反之，若可能使排序后的序列中 R_j 领先于 R_i，则称所用的排序方法是不稳定的。

由于待排序的记录数量不同,使得排序过程中涉及的存储器不同,可将排序方

法分为两大类：一类是内部排序，指的是待排序记录存放在计算机随机存储器中进行的排序过；另一类是外部排序，指的是待排序记录的数量很大,以致内存一次不能容纳全部记录在排序过程中尚需对外存进行访问的排序过程。本次作业中涉及到的快速排序、归并排序等均为内部排序。

内部排序的方法很多，在不同的排序情况下各有优劣。对内部排序方法进行分类，则大致可分为插入排序、交换排序、选择排序、归并排序和计数排序等五类；如果按内部排序过程中所需的工作量来区分，则可分为 3 类：(1) 简单的排序方法，其时间复杂度为 $O(n^2)$；(2) 先进的排序方法，其时间复杂度为 O(nlogn)；(3) 基数排序，其时间复杂度为 O(d·n)。

在排序的过程中通常需进行下列两种基本操作：(1) 比较两个关键字的大小；(2) 将记录从一个位置移动至另一个位置。前一个操作对大多数排序方法来说都是必要的，而后一个操作可以通过改变记录的存储方式来予以避免。待排序的记录序列可有下列 3 种存储方式：(1) 待排序的一组记录存放在地址连续的一组存储单元上。它类似于线性表的顺序存储结构,在序列中相邻的两个记录R_i和$R_{i+1}(i = 1, 2, \dots, n)$，它们的存储位置也相邻。在这种存储方式中，记录之间的次序关系由其存储位置决定，则实现排序必须借助移动记录；(2) 一组待排序记录存放在静态链表中，记录之间的次序关系由指针指示，则实现排序不需要移动记录，仅需修改指针即可；(3) 待排序记录本身存储在一组地址连续的存储单元内，同时另设一个指示各个记录存储位置的地址向量，在排序过程中不移动记录本身，而移动地址向量中这些记录的“地址”，在排序结束之后再按照地址向量中的值调整记录的存储位置。在第二种存储方式下实现的排序又称（链）表排序,在第三种存储方式下实现的排序又称地址排序。

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 $A_i > A_j$ 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

```
#define MAXN 20005
//int型数组存放输入的数字 并用于排序
//newarr用于暂存merge后的新数组
int arr[MAXN], newarr[MAXN];
```

2.1.4 功能说明（函数、类）

```
/**
 * @brief 将两个有序的序列合并
 * @param arr 存放序列的数组
 * @param l 前一个序列的开始
 * @param m 两序列的分隔下标
 * @param r 后一个序列的结束
*/
void Merge(int arr[], int l, int m, int r)

/**
 * @brief 归并排序
```

```
     * @param arr 排序的数组
     * @param l 此时在排序序列的最左侧下标
     * @param r 此时在排序序列的最右侧下标
    */
    void MergeSort(int arr[], int l, int r)
```

2.2.5 调试分析

1. 在上述的 Merge 函数中，进行已经排序的两个子序列的合并，将二者合并成一个更长的有序序列。需要注意的是，arr 数组存放的是题目中的整个数组，因此此时两子序列合并后的第一个下标并非对应 arr 下标开始处 1，而是应该是下标 l。因此在合并函数中，要从下标 l 处开始合并，否则会出错;

2. 每次归并排序将当前的序列分成两个部分。若令 m = (l + r) / 2，则两部分的下标区间分别为[l, m] 与 [m+1, r]，然后再分别对两个序列进行递归的归并排序，直至序列长度为 1 时停止。起初在程序中将两个区间的下标误当作[l, m] 与 [m, r]，即 m 下标的元素被前后两次排序纳入，导致出错;

3. 在归并排序中，每次合并两个子序列时，若前面序列某元素的值大于后面序列某元素的值，则对前面某元素来说，包含其在内的之后元素都会与后序列内的元素产生逆序数对，可以通过此累计计数来计算所需的值。但是要注意的是，逆序数的定义是满足 i < j 且 $A_i > A_j$ 的数对 (i, j)。因此在两个元素的大小判断上，对两个元素相等的判断要谨慎，否则会出错。起初我将等于关系放入了错误的分支，导致具有等于关系的数对也被错误地统计为逆序对。

2.1.6 总结和体会

本题中所设计到的算法为归并排序，这是一种比较常用的排序算法。它的工作原理是将两个已经排好序的序列合并成一个更大的有序序列。归并排序采用分治的策略，将原序列不断地划分成越来越小的两个子序列，直到每个子序列只有

一个元素为止。然后将这些有序的子序列两两合并，不断合并得到有序序列，最终得到原序列的有序版本。归并排序的时间复杂度为 O(n log n)。

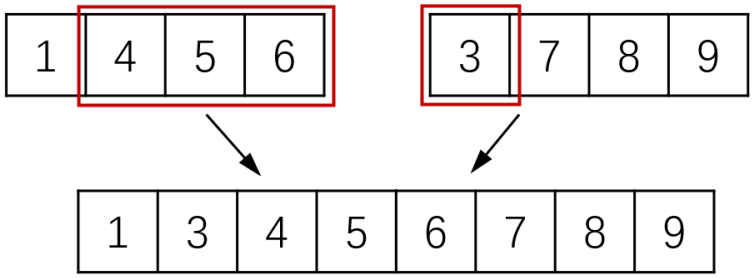

图 1　在归并排序中计算逆序数对个数

在归并排序合并两个有序序列的过程中，用一个指针 k 在生成的结果序列中移动，两个指针分别在两个子序列中移动，不断将两个指针中所指内容值更小的值放入 k 当前所在的下标处，在使得 k 自增继续填充。对于逆序数对的个数计算，采用如下的方法：如上方图 1 中所示，若当前后子序列的指针指向值大于前子序列指针指向的值（在图中对应“3”小于“4”），则由于子序列的有序性，前子序列之后的所有值也均大于当前后子序列的指针所指向元素的值。因此会诞生（前子序列长度-当前前子序列指针下标）对的逆序数对。在过程中用一个变量不断累积计数，即可统计逆序数对个数，对时间复杂度无额外增加。

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。

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。

2.2.2 基本要求

输入：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：输出包含一行，为重新排列后得到的数字。

2.2.3 数据结构设计

```
//将每个整数转为string方便比较 存入vector中
vector<string> numarray;
```

2.2.4 功能说明（函数、类）

```
/**
 * @brief （快速排序）将枢轴前后按大小分割
 * @param L 排序的vector
 * @param low 当前排序的序列起始下标
 * @param high 当前排序的序列结束下标
 * @return 枢轴的下标
*/
int Partition(vector<string>& L, int low, int high)

/**
 * @brief 快速排序
 * @param L 排序的vector
 * @param low 当前排序的序列起始下标
 * @param high 当前排序的序列结束下标
*/
   void Qsort(vector<string>& L, int low, int high)


/**
 * @brief 将数字数组转为string数组
 * @param nums 输入的数组数组
*/
void convert(vector<int>& nums)
```

```
/**
 * @brief 定义string比较大小
 * @param a 字符串a
 * @param b 字符串b
*/
static bool LT(string a, string b)
```

2.2.5 调试分析

经过思考分析后，容易得到整体的思路是通过字典序比较以字符串的方式逐个比较整数，按照从大到小的排列之后按照顺序输出即可。整体上比较顺利，但在比较字符大小方面出现了一个小问题，即若两个字符串长度不等时，直接用 string 原生的小于号重载比较会出错。例如在“3”与“30”的比较中，两种组合“330”和“303”显然前者更优，即“3”应该“大于”“30”从而被放在前面。在几次调试中我不断改进了对字符逐个比较的写法，但始终有逻辑上的错误。最终我发现本质上这里的比较就是比较两个字符串相拼凑后的字符串大小，即比较字符串 a 与 b 拼接后字符串 a+b 与 b+a 的字典序大小。

2.2.6 总结和体会

本题的大体思路是将输入的整数以字典序方式（略有差异）比较进行排序，按照从大到小的排列之后，按照顺序依次输出。需要注意到两个字符串长度不等时，要考虑到细节上的差异，即应该字符串 a 与 b 拼接后字符串 a+b 与 b+a 的字典序大小来确定两个字符串的前后关系。

在本题的排序算法选择时，我选择了快速排序。快速排序是一种高效的排序算法，它的基本思想是通过一趟排序将待排序的记录分割成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序的目的。

快速排序的基本步骤如下：

1. 选择一个基准元素，通常选择第一个元素或者最后一个元素；

2. 通过一趟排序将待排序的记录分割成独立的两部分，其中一部分记录的关键字均比基准元素的关键字小，另一部分的关键字均比基准值大；

3. 分别对这两部分记录继续进行排序，直到整个序列有序。

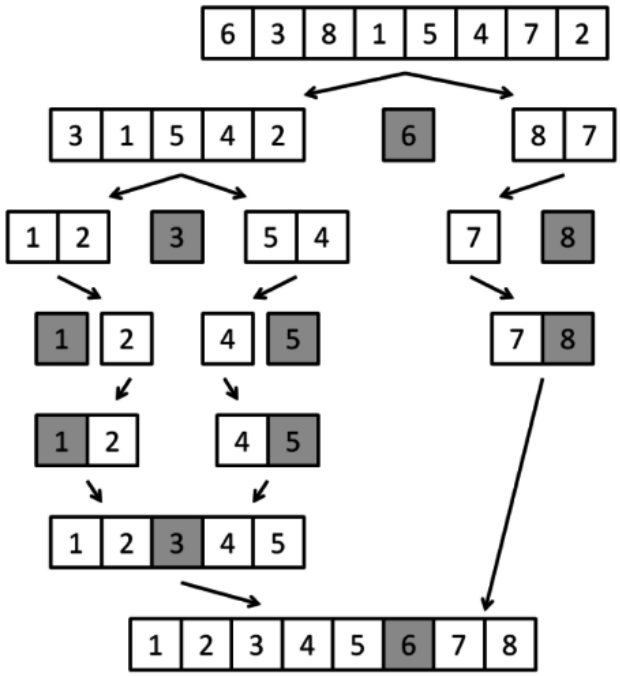

图 2　快速排序算法示意图

快速排序是一种分治的排序算法，在一般情况下它的时间复杂度为 O(nlogn)；但是在最坏状况下（如数列逆序），它的时间复杂度是 $O(n^2)$。这可以通过三数取中法进行优化，具体的算法以及分析我在第三题“排序”中进行了详细实践。

2.3 排序

2.3.1 问题描述

排序算法分为简单排序（时间复杂度为 O(n^2)）和高效排序（时间复杂度为 O(nlog n)）。

本题给定 N 个整数，要求输出从小到大排序后的结果。请用不同的排序算法测试，注意有些算法无法拿到满分，由此同学们可以猜测一下 10 个测试用例的数据特征。

请同学们自己随机生成不同规模的数据（例如 10，100，1K，10K，100K，1M，10K 正序，10K 逆序），用不同的排序算法（快速排序，归并排序，堆排序，选择排序，冒泡排序，直接插入排序，希尔排序）分别对这些数据进行测试，输出运行时间，将结果写在实验报告中，并总结各种排序算法的特点、时间复杂度，以及是否是稳定排序。

2.3.2 基本要求

输入：第 1 行一个正整数 n，表示元素个数

第 2 行 n 个整数，用空格分割。

输出：从小到大排序后的结果，以空格分割。

2.3.3 数据结构设计

```
//存放输入的数用于排序
vector<int> nums(n);
```

2.3.4 功能说明（函数、类）

```
/**
 * @brief 三数取中法找枢轴
 * @param L 排序的vector
 * @param low 当前排序的序列起始下标
 * @param high 当前排序的序列结束下标
*/
void findpivotkey(vector<int>& L, int low, int high){

/**
 * @brief （快速排序）将枢轴前后按大小分割
 * @param L 排序的vector
 * @param low 当前排序的序列起始下标
 * @param high 当前排序的序列结束下标
```

```
 * @return 枢轴的下标
*/
int Partition(vector<int>& L, int low, int high)

/**
 * @brief 快速排序
 * @param L 排序的vector
 * @param low 当前排序的序列起始下标
 * @param high 当前排序的序列结束下标
*/
void Qsort(vector<int>& L, int low, int high)

/**
 * @brief 将两个有序的序列合并
 * @param arr 存放序列的数组
 * @param l 前一个序列的开始
 * @param m 两序列的分隔下标
 * @param r 后一个序列的结束
*/
void Merge(vector<int>& arr, int l, int m, int r)

/**
 * @brief 归并排序
 * @param arr 排序的数组
 * @param l 此时在排序序列的最左侧下标
 * @param r 此时在排序序列的最右侧下标
*/
void MergeSort(vector<int>& arr, int l, int r)


/**
 * @brief 调整s到m的堆使其满足堆的性质
 * @param H 堆
 * @param s 起始下标
 * @param m 终止下标
*/
void HeapAdjust(vector<int>& H, int s, int m)


/**
 * @brief 堆排序
 * @param H 用于排序的数组
*/
void HeapSort(vector<int>& H)

/**
 * @brief 希尔排序
```

```
 * @param nums 排序的数组
*/
void ShellSort(vector<int>& nums)

/**
 * @brief 希尔排序的某一趟
 * @param nums 排序的数组
 * @param dk 增量
*/
void ShellInsert(vector<int>& nums, int dk)
```

2.3.5 调试分析

我在该作业的调试问题主要出现在数组下标的处理问题上。因为输入时数组元素的存储是从 0 下标开始的，但在课本上学习时有时示例算法是从 1 开始（尤其是堆排序这样下标有特殊特征的算法），在不注意时就会出现越界访问的情况。除此之外，困难主要出现在对算法概念以及实现的理解方面，不同算法的实现与时间复杂度在下一部分进行详细地介绍。

2.3.6 总结和体会

1. 首先分析平均时间复杂度为 $O(n^2)$的简单排序，包含直接插入排序、冒泡排序和选择排序，分析如下：

直接插入排序的工作原理是：首先，将第一个元素看作是一个有序序列，把第二个元素到最后一个元素当成是未排序序列；取出未排序序列中的第一个元素，在已排序序列中从后向前扫描；如果已排序序列中的元素大于新元素，将该元素移到下一位置；重复上一步直到找到已排序序列中的元素小于或者等于新元素的位置，将新元素插入到该位置后。重复上述步骤直至整个数组有序。直接插入排序的平均时间复杂度为 $O(n^2)$，最好情况下（即数组单调不减时）时间复杂度为

O(n)，是一种稳定的排序算法。

冒泡排序的工作原理是：通过对待排序序列从前向后（从下标较小的元素开始），依次比较相邻元素的值，若发现逆序则交换，使值较大的元素逐渐从前移向后部，就像水底下的气泡一样逐渐向上冒。冒泡排序的时间复杂度为 $O(n^2)$，是一种稳定的排序算法。

选择排序的的工作原理是：第一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，然后再从剩余的未排序元素中寻找到最小（大）元素，然后放到已排序的序列的末尾。以此类推，直到全部待排序的数据元素排完。选择排序的时间复杂度为 $O(n^2)$，是一种不稳定的排序算法。

2. 接下来分析时间复杂度为 O(nlogn)的快速排序算法，包含快速排序、堆排序、归并排序，分析如下：

快速排序是一种高效的排序算法，它的基本思想是通过一趟排序将待排序的记录分割成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序的目的。快速排序不是稳定的算法。在使用快速排序时，若不进行优化则会出现 TLE 的情况：

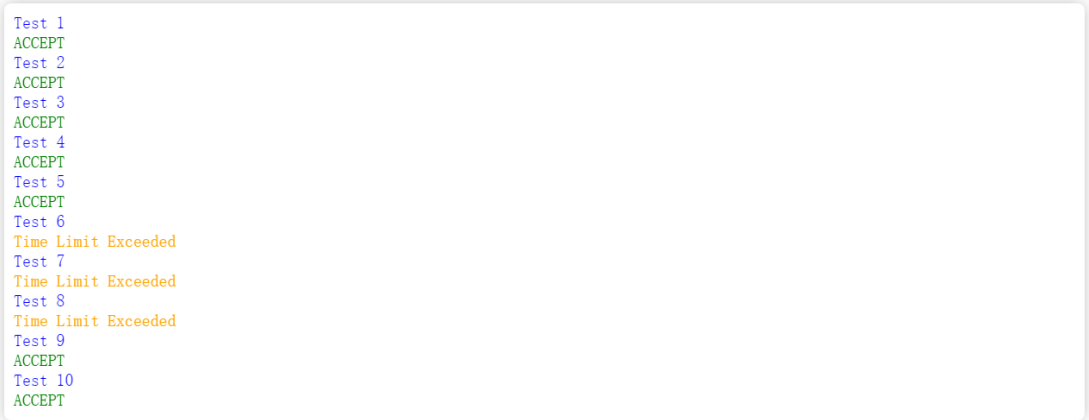

图 3　未经优化的快速排序在某些数据下时间复杂度退化

这是因为虽然快速排序通过分治的思想达到 O(nlogn)的时间复杂度，但是在比如逆序的序列时会退化成冒泡排序，时间复杂度为 $O(n^2)$。可以通过“三数取中法”来进行优化，即在数列最左侧、最右侧与中间的取值取中间值作为枢轴：

```
void findpivotkey(vector<int>& L, int low, int high)
{
    int min, max, mid = (low + high) / 2;
    if (L[low] < L[high]) {
        min = low;
        max = high;
    }
    else {
        min = high;
        max = low;
    }

    if (L[mid] < L[min])
        min = mid;

    if (L[mid] > L[max])
        max = mid;

    mid = low + high + mid - min - max;
    swap(L[low], L[mid]);
    return;
}
```

归并排序通过将两个已经排好序的序列合并成一个更大的有序序列进行排序。如在作业第一题中所分析，归并排序采用分治的策略，将原序列不断地划分成越来越小的两个子序列，直到每个子序列只有一个元素为止。然后将这些有序的子序列两两合并，不断合并得到有序序列，最终得到原序列的有序版本。归并排序的时间复杂度为 O(nlogn)，是一种稳定的排序算法。

堆排序将待排序序列构造成一个大根堆（整个序列的最大值就是堆顶的根节点）。将根节点与末尾元素进行交换，此时最大值被放至数组末尾。然后将剩余

n-1 个元素重新构造成一个堆，这样会得到 n 个元素的次小值，如此反复执行，便能得到一个有序序列。堆排序的时间复杂度为 O(nlogn)，是不稳定的算法。

希尔排序是对直接插入排序的优化，它先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，通过交换不相邻的元素，使得数组中任意间隔为 h 的元素都是有序的。待整个序列中的记录"基本有序"时，再对全体记录进行依次直接插入排序。根据不同增量序列的选择，希尔排序的时间复杂度不尽相同，本次选择的增量序列将 length 不断除以 2 所得。希尔排序不是稳定的算法。

```
/**
 * @brief 希尔排序的某一趟
 * @param nums 排序的数组
 * @param dk 本轮的增量
*/
void ShellInsert(vector<int>& nums, int dk)
{
    int rec, j;
    for (int i = dk; i < nums.size(); i++) {
        if (nums[i] < nums[i - dk]) {
            rec = nums[i];
            for (j = i - dk; j >= 0 && rec < nums[j]; j -= dk)
                nums[j + dk] = nums[j];
            nums[j + dk] = rec;
        }
    }
}

/**
 * @brief 希尔排序
 * @param nums 排序的数组
*/
void ShellSort(vector<int>& nums)
{
    for (int i = nums.size(); i >= 1; i /= 2)
        ShellInsert(nums, i);
}
```

2.4 最大频率栈

2.4.1 问题描述

设计一个类似堆栈的数据结构，将元素推入堆栈，并从堆栈中弹出出现频率最高的元素。实现 FreqStack 类:

FreqStack() 构造一个空的堆栈。

void push(int val) 将一个整数 val 压入栈顶。

int pop() 删除并返回堆栈中出现频率最高的元素。如果出现频率最高的元素不只一个，则移除并返回最接近栈顶的元素。

2.4.2 基本要求

输入：第一行包含一个整数 n。

接下来 n 行每行包含一个字符串（push 或 pop）表示一个操作，若操作为 push，则该行额外包含一个整数 val，表示压入堆栈的元素；

对于 100%的测试数据，1<=n<=20000，0<=val<=10^9，且当堆栈为空时不会输入 pop 操作输出：一个整数，表示最少交换次数。

输出：输出包含若干行，每有一个 pop 操作对应一行，为弹出堆栈的元素。

2.4.3 数据结构设计

```
class FreqStack {
private:
    map<int, int> num_freq;           //存放每个数出现频率
    map<int, stack<int>> freq_num;    //存放某频率为下标的所有整数
    int max_freq;                     //当前最大的出现频率
}
```

2.4.4 功能说明（函数、类）

```
/**
 * @brief 向最大频率栈插入新元素
 * @param val 元素值
*/

void push(int val)


/**
 * @brief 最大频数栈弹出元素
 * @return 弹出的元素值
*/

int pop()
```

2.4.5 调试分析

本题中我通过两个 map 来存储信息，其中一个 map 存储对应频率下的所有元素。如数字 3、5、7 出现频率均为 5，且按照罗列顺序依次 push 入栈，则以频率 5 为下标的栈中元素从栈顶至栈底分别为 7、5、3。若当前频率是第一次出现，则需要在对应下标的 map 中建立新的栈。在建立栈时我忘记将第一个该频率的数也 push 入栈，这会导致记录的错误。同时，若栈中仅有一个元素，还会导致访问到空的下标产生报错。除此之外，本题通过得比较顺利。

2.4.6 总结和体会

本题中我通过维护 FreqStack 类中的三个成员变量：num_freq 和 freq_num 和 max_frequency 变量来实现最大频率栈。num_freq 与 freq_num 都是 map 类型的变量，其中 num_freq 用来存储每个数字出现的频率，key 为数字，value 为出现频率；freq_num 用来存储某一频率对应的所有整数（按照输入顺序排列），key 为频率，value 为存储所有整数的栈。FreqStack 类还包含一个 private 类型的成员变量 max_freq，用来维护当前栈中出现的最大频率。

FreqStack 类的 push 函数用来向最大频数栈中插入新元素。当插入一个新元

素时，首先判断该元素是否已经存在于栈中，如果存在，则在 num_freq 中找到该元素，并将其出现频率加 1；如果不存在，则将该元素的出现频率设为 1。同时，也需要维护更新 max_freq，如果新插入的元素的频率比当前 max_freq 大，则将 max_freq 更新为新插入元素的频率。最后将新插入的元素插入 freq_num 中对应的栈中。如果 freq_num 中不存在该频率对应的栈，则新建一个栈，并将新插入的元素插入该栈中。

FreqStack 类的 pop 函数用来从最大频数栈中弹出元素。首先在 freq_num 中找到当前最大频率对应的栈，并从该栈中弹出栈顶元素。然后在 num_freq 中找到该元素，并将其出现频率减 1。接着检查 freq_num 中是否还存在当前最大频率对应的栈，如果该栈已经为空，则将 max_freq 的值减 1 以便下次输出。

2.5 序列和

2.5.1 问题描述

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：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：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

```
//ans通过不断更新累加保持个数为n个
//即输入i行之后，前i行各出一个元素的前n小和数组
int ans[MAXN];
//new_arr代表新输入的一行
int new_arr[MAXN];
```

2.5.4 功能说明（函数、类）

```
/**
 * @brief 读入新的一行new_arr后 维护新的前n小和
 * @param ans 已经读入行的前n小和
 * @param new_arr 新读入的一行
 * @param n 每行元素个数
*/
void maintain_sum(int* ans, int* new_arr, int n)
```

2.5.5 调试分析

每读入新的一行，都要将已有的元素直接进行两两比较。该算法的时间复杂度为 $O(T * m * n^2 * logn)$ （共 m 行，每次进行两两比较，log(n)由维护大根堆产生），其中 T 表示数据组数，m 表示数组个数，n 表示每个数组的数字个数，会导致 TLE。经过思考之后，考虑用如下方式进行优化：为了加快比较的速度，使用快排来将每个数组排序。排序后可以使用剪枝，即当遍历到的数字已经超过优先队列的前 n 小数字的最大值时，直接停止遍历。快排算法只在最开始的时候调用一次，用来对每个数组进行排序，因此快排的时间复杂度不会影响整个算法的时间复杂度。经此优化即可通过，具体代码如下（数组排序在 main 函数进行）：

```
/**
 * @brief 读入新的一行new_arr后 维护新的前n小和
 * @param ans 已经读入行的前n小和
 * @param new_arr 新读入的一行
 * @param n 每行元素个数
*/
void maintain_sum(int* ans, int* new_arr, int n)
{
    priority_queue<int> q;

    // O(n^2)的时间复杂度 依次比较原先队列
    for (int i = 0; i < n; ++i){
        for (int j = 0; j < n; ++j){
            //新的一个sum 后续判断是否要更新
            int sum = ans[i] + new_arr[j];

            //q要先填充满n个的大小
            if (q.size() < n)
                q.push(sum);
            else if (sum < q.top()){
                q.pop();
                q.push(sum);
            }
            //剪枝 因为两个序列均有序 若当前已经超过，则之后均不符合题意
            else
                break;
        }
    }

    //q是大根堆 ans数组的排序保持从小到大 因此反过来
    for (int i = n - 1; i >= 0; i--){
        ans[i] = q.top();
        q.pop();
    }
}
```

2.5.6 总结和体会

本题的大体实现方式如下：先考虑 m 为 2，即仅有两行的情况。此情况下，罗列出第一行与第二行元素两两之和，取出前 n 即可。可以通过大根堆来进行优化，即大根堆中存放大小为前 n 的两数之和，若新遍历到的两数和小于大根堆堆

顶的值，则将其与堆顶交换并重新维护大根堆。因此时间复杂度为 $O(n^2 * \log n)$，其中 logn 为维护大根堆的时间花费。扩展到 m 行的情况，每一次求和之后将大根堆中的元素依次 pop 作为前一行数组（从后往前插入，能够保证数组从小到大排列，对于之后的算法优化有用），则每一次均做相同的操作。该算法总的时间复杂度为 $O(T * m * n^2 * \log n)$，如果不经过优化则该时间复杂度无法通过本题。我考虑使用排序与剪枝的方法进行优化：首先使用快排将每行新读入的数组排序。排序后可以使用剪枝，即当遍历到的数字已经超过优先队列的前 n 小数字的最大值时，直接停止遍历。快排算法只在最开始的时候调用一次，用来对每个数组进行排序，因此快排的时间复杂度不会影响整个算法的时间复杂度。用快速排序加上剪枝之后，能有效对程序进行优化。

3. 实验总结

这次作业中，我学习了各种不同的内部排序算法，体会了它们各自的优劣与使用情况，分析了不同算法的时间复杂度与稳定性。具体来看，我分别实践了归并排序（求逆序数对）、快速排序（最大数）等等算法，同时在第三题中将所有学习过的内部排序算法进行实践，比较它们的特点差异与适用情况。

第一题涉及归并排序求逆序数对。归并排序将一个数组不断二分，直到每一个子数组只剩下一个元素，然后再不断合并使得最后的数组有序。在归并排序合并过程中，便可以直接统计逆序数对的个数。第二题涉及快速排序，每次将数组分成两部分，通过一趟排序将数组分为两部分，左边部分的元素都小于右边部分的元素。然后递归地对左右两部分继续进行快速排序，直到每一个子数组只剩下一个元素为止。在第三题中，我练习了时间复杂度为 $O(n^2)$的简单排序，包括冒

泡排序、选择排序、直接插入排序；也实践了时间复杂度为 O(nlogn)的快速排序、堆排序、归并排序等等。对于快速排序退化为 $O(n^2)$的情况，我采取“三数取中法”进行优化。对于希尔排序，我了解了其在直接插入排序基础上的优化，也探究了不同增量序列的时间复杂度优劣。同时，我也学习了不同算法的稳定性。第四、第五两题是对排序的更为灵活的应用，我均能在思考与学习中很好地完成。

总体来说，本次实验使我加深了对各种排序算法的理解，比较了不同排序算法的特点和优劣，并熟悉了它们的时间复杂度和稳定性。在实验过程中，我较为熟练地运用了内部排序算法的相关知识完成了实验作业。在以后的编程过程中，希望我能够继续积累经验，将这些数据结构知识应用到实际问题中。