作业 HW6 实验报告

姓名：段威呈 学号：2252109 日期：2024 年 1 月 6 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：

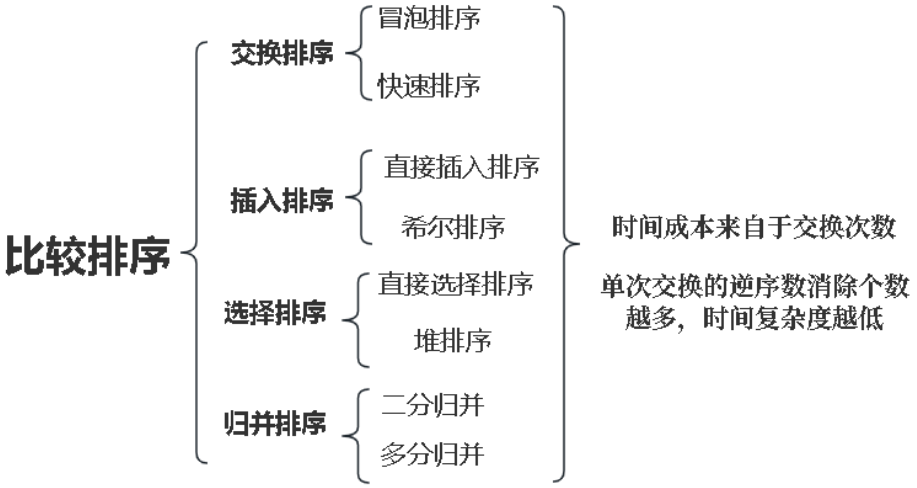

图 1.1 比较排序分类

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。

图 1.2 常见非比较排序

2. 实验内容

2.1 求逆序对数

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 算法设计

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。

下面对常见的冒泡、插入、选择、归并以及快速排序算法的过程进行分析，来判断他们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。

①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：

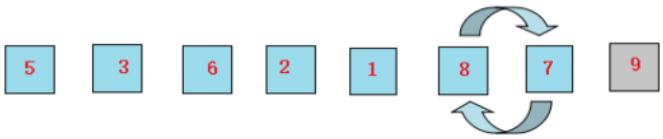

图 2.1.1 冒泡排序

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。

②插入排序是对每个数从后向前依次比较，若后插数比原排列中的前数小，则与前交换。在满足$a[j-1] \le a[j] \le a[j+1]$ 的情况下停止，即将数字成功插入到j位置：

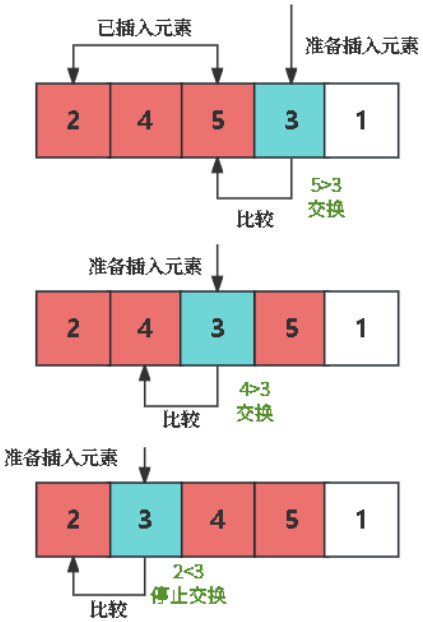

图 2.1.2 插入排序

在插入排序中，也进行了逆序对交换的过程，且所有交换过程都仅交换了一个逆序对都。因此插入排序也可以实现统计逆序对数。

③选择排序实质是从序列中不断挑选出最大值并排放到最后端（堆排序核心思想也是如此）。因此在一次每次交换时往往会跳过很多项。但是由于跳过的这些项也是乱序的，因此并不能确定到底消除了多少对逆序数。故选择排序，包括堆排序，都不可以用于统计逆序对数。

④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。

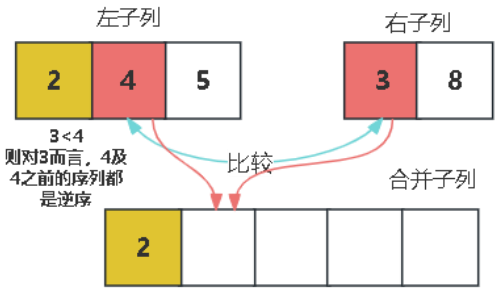

图 2.1.3 顺序子列合并

综上可知，冒泡排序、选择排序以及归并排序都可以用来统计逆序对数。注意到三种排序算法的时间复杂度分别是 O(n)、O(n)以及 O(nlogn)，因此应当优先选择归并排序的方法。

2.1.4 算法的具体实现

归并算法实现分为两部分：第一是二分，第二是合并。

二分的模拟过程如下：

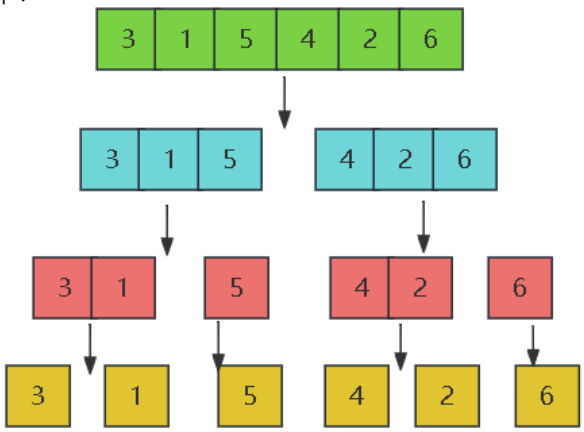

图 2.1.4 归并排序——分

由以上过程可知，对于一个长度为 n 的序列，“分”的次数成本可以近似为$logn$.

分的代码如下：

```
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;
}
```

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：

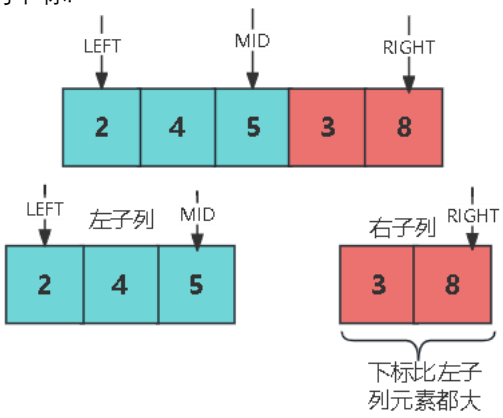

图 2.1.5 归并排序子列合并

在合并的时候，要注意如果是右子列的元素比左子列小，那么说明出现了逆序对，则从左子列中这个元素以左的元素都与右子列的该元素构成逆序对。

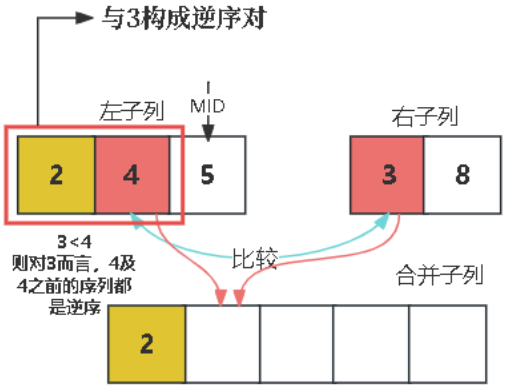

图 2.1.6 子列合并

合并过程如下：

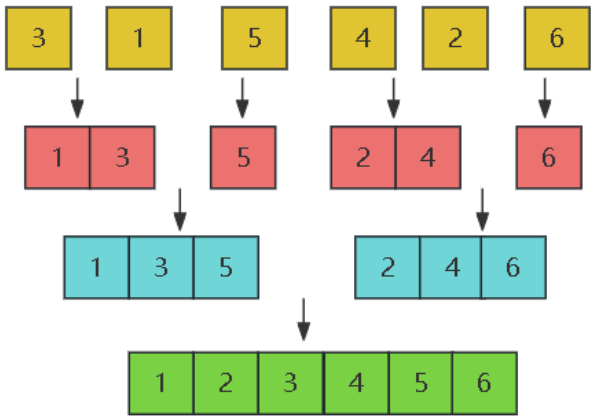

图 2.1.7 归并排序——并

合并的实现代码是:

```
void Merge(int Seq[MAX_LEN], int left, int right, int &num,int temp[MAX_LEN]) {
    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];
    }
}
```

注意，每次交换，逆序对数都要加 ***mid+1-i***.

2.1.5 调试分析（遇到的问题和解决方法）

本题目中特别注意到每次交换，逆序对数都是 mid-i+1。

2.1.6 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(1)。

2.2 三数之和

2.2.1 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。

2.2.2 基本要求

输入

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

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。

2.2.3 算法设计

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

注意到题目中有要求输出 3 数的顺序要从小到大，这也说明其实需要先对原数列进行排序。这里先对序列按照从小到大的顺序进行排序。使用快速排序的方法。

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。

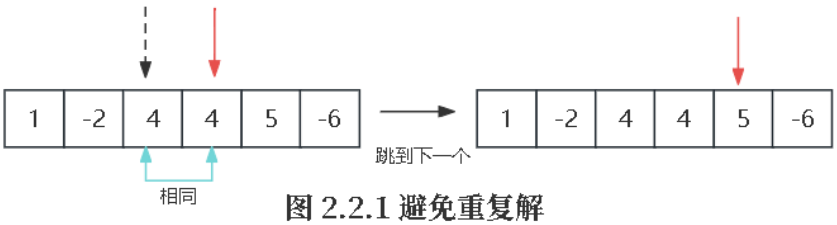

图 2.2.1 避免重复解

2.2.4 算法的具体实现

(1)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。

QuickSort()代码如下：

```
void swap(int i, int j, std::vector<int>& nums) {
    int tmp;
    tmp = nums[i];
    nums[i] = nums[j];
    nums[j] = tmp;
}
void QuickSort(std::vector<int>& nums, int left, int right) {
    if (left > right)
        return;
    int pivot = left;
    int i = left;
    int j = right;
    while (i < j) {
        while (i < j && nums[j] > nums[pivot]) j--;
        while (i < j && nums[i] < nums[pivot]) i++;
        if (i < j) {
            swap(i, j, nums);
        }
    }
    swap(pivot, i, nums);
    QuickSort(nums, left, i - 1);
    QuickSort(nums, i + 1, right);
}
std::vector<int> mySort(std::vector<int>& nums) {
    QuickSort(nums, 0, nums.size() - 1);
    return nums;
}
```

(2)找符合条件的三数

```
void Find3Sum(int Seq[MAX_LEN],int left,int right) {
    int pivot = left;
    int i = pivot;
    int j = right;
    while (pivot < right) {
        //避免重复
        if (pivot > 0) {
            while (Seq[pivot] == Seq[pivot - 1]) {
                pivot++;
            }
        }
        j = right;
        i = pivot + 1;
        while (i < j) {
            if (Seq[pivot] + Seq[i] + Seq[j] == 0) {
                Save3Sum(Seq[pivot], Seq[i], Seq[j]);
                j--;
                //避免重复
                if (j < right) {
                    while (Seq[j] == Seq[j + 1]) {
                        j--;
                    }
                }
            }
            else if (Seq[pivot] + Seq[i] + Seq[j] > 0) {
                j--;
                //避免重复
                if (j < right) {
                    while (Seq[j] == Seq[j + 1]) {
                        j--;
                    }
                }
            }
```

```
            else {
                i++;
            }
        }
        pivot++;
    }
```

2.2.5 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。

2.3 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。

2.3.2 基本要求

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、空间复杂度，以及是否是稳定排序和部分排序。

2.3.3 算法实现过程

(1)冒泡排序

一种经典的 BF 算法，思想是依次两两比较交换：

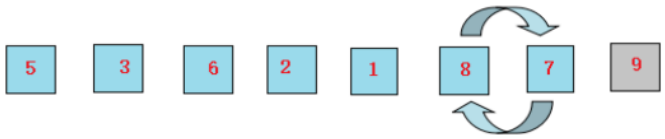

图 2.1.1 冒泡排序

实现代码：

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

时间复杂度是 O(n)

(2)插入排序

插入排序是从后向前插入，依次比较。效率略比冒泡排序高。实现代码如下：

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

(3)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。

```cpp
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;
            }
        }
    }
}
```

(4)归并排序

归并排序过程如下：

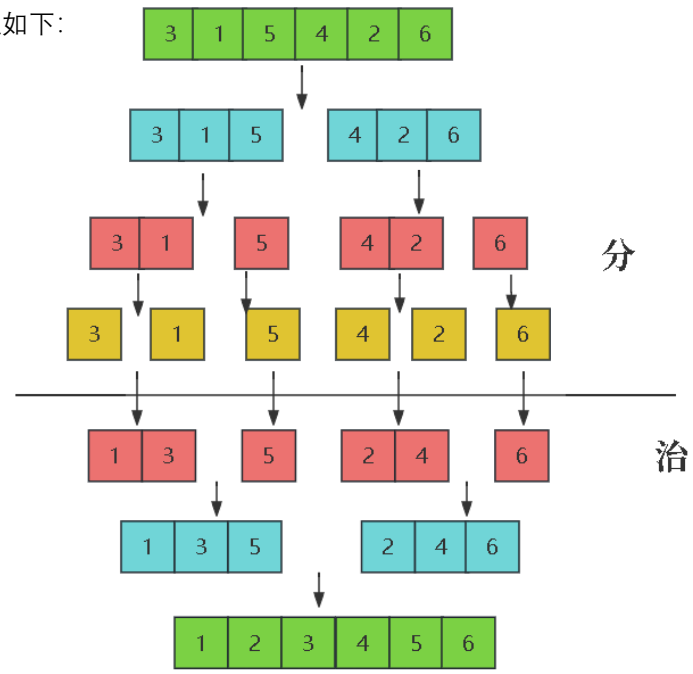

图 2.3.1 归并排序

实现代码如下：

```
void MergeSort(std::vector<int>& nums, int left, int right, std::vector<int>& temp) {
    if (left < right) {
        int mid = (left + right) / 2;
        MergeSort(nums, left, mid, temp);
        MergeSort(nums, mid + 1, right, temp);
        Merge(nums, left, right, temp);
    }
    return;
}
void Merge(std::vector<int>& nums, int left, int right, std::vector<int>& temp) {
    int mid = (left + right) / 2;
    int i = left;
    int j = mid + 1;
    int tmp = 0;
    while (i <= mid && j <= right) {
        if (nums[i] <= nums[j]) {
            temp.push_back(nums[i++]);
            ++tmp;
        }
        else {
            temp.push_back(nums[j++]);
            ++tmp;
        }
    }

    while (i <= mid) {
        temp.push_back(nums[i++]);
        ++tmp;
    }

    while (j <= right) {
        temp.push_back(nums[j++]);
        ++tmp;
    }

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

(5)快速排序

快速排序的实质是尽可能的把小的值放到左边、大的值放到右边，从而使得一次交换可以消除更多的逆序对数。为了判断什么是小、什么是大，因此每次做分割的时候都要定义一个基准值$pivot$，这个$pivot$一般都选取第一个值。分割完后，$pivot$位于中间，左边都是比$pivot$小的值，右边都是比$pivot$大的值。之后再分别对左边和右边进行同样的操作，直到分无可分。在快速排序的实现过程中用到了分治的思想。实现代码已在问题 2 中展示，这里不再重复展示。

(6)选择排序

选择排序的实质是找出序列中的最大值，把最大值调整到最末端；之后再找剩下的子列中的最大值，再排到次末端……依次进行排序，直到排列完所有元素。实现代码如下：

```
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;
    }
}
```

(7)堆排序

堆排序的核心理念与选择排序相同。但是堆排序通过构建一个大顶堆优先级队列，可以及其高效的获取到序列中的最大值。而大顶堆实际上是一种完全二叉树，不断维护大顶堆使得二叉树的根节点就是最大值。构建大顶堆的代码如下：

```
//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;
        }
    }
}
```

构造大顶堆后，要依次把最大值放在序列最后，再把剩下的元素调整为大顶堆结构。重复进行操作，直到所有元素都排列完成。调整过程的实现代码如下：

```
//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;
    }

}
```

2.3.4 算法的性能比较

因这里自行构造了 10K， 10K 正序，10K 逆序。

	10K	10K 正序	10K 逆序
快速排序	4582	20454	21584
归并排序	5527	6974	7258
堆排序	6985	4857	9851

2.3.5 调试分析（遇到的问题和解决方法）

在实验过程中，发现只有堆排序通过了所有检查点。但是快速排序没有通过检查点 6,7,8

```
Test 1
ACCEPT
Test 2
ACCEPT
Test 3
ACCEPT
Test 4
ACCEPT
Test 5
ACCEPT
Test 6
Time Limit Exceeded
Test 7
Time Limit Exceeded
Test 8
Time Limit Exceeded
Test 9
ACCEPT
Test 10
ACCEPT
```

这说明 6,7,8 是大规模数据的正序/逆序。于是这里想到优化方法，如果可以破坏顺序，则可以实现优化。因此先对原序列进行随机扰动：

生成随机扰动序列：

```
double epsilon = 0.3;
double atnum = epsilon * nums.size();
int atp;
std::vector<int>p;
srand((unsigned int)time(NULL));
for (int i = 0; i < atnum; i++) {
    atp = rand() % (nums.size());
    p.push_back(atp);
}
RandomAttack(p, nums);
```

进行随机扰动：

```
void RandomAttack(std::vector<int> p, std::vector<int>& nums) {
    int i = 0;
    int j = p.size() - 1;
    while (i < j) {
        swap(p[i], p[j], nums);
        i++;
        j--;
    }
}
```

同时还发现现象，作为同为 O(n)复杂度的排序算法，冒泡排序算法比插入排序算法通过的检查点更少。冒泡排序算法只能通过 5 个检查点，而插入排序算法通过检查点 7 个。经过仔细的时间复杂度分析发现，冒泡排序的时间成本代价精确值是(n,2)，而插入排序算法的时间成本代价近似接近 n^2/4，因此插入排序算法可以通过更多检查点。

2.3.6 总结和体会

本题实现了常用的 7 个排序算法，每种算法在不同情况下表现出不同的性能，要在根据实际情况选择最优的算法。

2. 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。