1.题目名称

排序算法比较

2.代码行数

430行

3.算法思想

时间测量

先记录当前时间并执行排序算法，排序算法执行结束后，记录结束时间，相减便是排序时间。

4.主要/核心函数分析

generateData

```
void generateData(int length)
{
    ofstream file;
    //升序
    file.open("task-1.txt", ios::out);
    for (int i = 0; i < length; i++)
        file << i << endl;
    file.close();
    //降序
    file.open("task-2.txt", ios::out);
    for (int i = 0; i < length; i++)
        file << length - i << endl;
    file.close();
    //乱序
    for (int k = 3; k <= 10; k++)
    {
        srand(time(NULL));
        string fileName = "task-" + to_string(k) + ".txt";
        file.open(fileName, ios::out);
        for (int i = 0; i < length - 1; i++)
        {
            int val = (rand() % 5000000);
            file << val << endl;
        }
        file << (rand() % 5000000);
        file.close();
    }
}
```

生成一组升序的test以及一组降序的test，随后便是8组乱序的test。

insertSort

```
void insertSort(int length)
{
    for (int i = 1; i < length; i++)
```

```
    {
        int temp = nums[i];
        int j = i - 1;
        while (j >= 0 && nums[j] > temp)
        {
            nums[j + 1] = nums[j];
            j--;
        }
        nums[j + 1] = temp;
    }
}
```

插入排序的基本思想是，将数组分为已排序和未排序两部分。初始时，已排序部分包含一个元素，之后从未排序部分取出元素，并在已排序部分找到合适的插入位置插入，并保持已排序部分一直有序。重复这个过程，直到未排序部分元素为空。

时间复杂度分析：
插入排序的时间复杂度取决于输入数据的特性。在最坏的情况下，输入数据是完全逆序的，此时每次插入操作都需要移动所有已排序的元素，因此时间复杂度为 O(n^2)。在最好的情况下，输入数据是已排序的，此时时间复杂度为 O(n)。平均时间复杂度也是 O(n^2)。

bubbleSort

```
void bubbleSort(int length)
{
    for (int i = 0; i < length - 1; i++)
    {
        int cnt = 0;
        for (int j = 0; j < length - i - 1; j++)
        {
            if (nums[j] > nums[j + 1])
            {
                int temp = nums[j];
                nums[j] = nums[j + 1];
                nums[j + 1] = temp;
                cnt++;
            }
        }
        if (cnt == 0) //没有交换，说明已经有序
            break;
    }
}
```

冒泡排序的基本思想是，通过不断地比较相邻的两个元素并交换它们（如果它们的顺序错误），使得较大的元素逐渐“浮”到数组的末尾。这个过程会重复多次，直到没有元素需要交换，也就是说数组已经排序完成。

时间复杂度分析：

- 最好情况：如果输入数组已经是排序好的，那么不需要任何交换操作，时间复杂度为 O(n)。
- 最坏情况：如果输入数组是完全逆序的，那么需要进行 n(n-1)/2 次交换操作，时间复杂度为 O(n^2)。
- 平均情况：由于在每一轮中最多交换 n-1 次，并且需要进行 n-1 轮，所以平均时间复杂度为 O(n^2)。

selectSort

```
void selectSort(int length)
{
    for (int i = 0; i < length - 1; i++)
    {
        int min = i;
        for (int j = i + 1; j < length; j++)
        {
            if (nums[j] < nums[min])
            {
                min = j;
            }
        }
        int temp = nums[i];
        nums[i] = nums[min];
        nums[min] = temp;
    }
}
```

选择排序的基本思想是，在未排序部分中每次找到最小（或最大）的元素，将其放到已排序部分的末尾。

时间复杂度分析：

- 最好情况：如果输入数组已经是排序好的，那么时间复杂度为 O(n)。
- 最坏情况：如果输入数组是完全逆序的，那么需要进行 n(n-1)/2 次比较操作，时间复杂度为 O(n^2)。
- 平均情况：由于在每一轮中最多比较 n-1 次，并且需要进行 n-1 轮，所以平均时间复杂度为 O(n^2)。

shellSort

```
void shellSort(int length)
{
    int gap = length / 2;
    while (gap > 0)
    {
        for (int i = gap; i < length; i++)
        {
            int temp = nums[i];
            int j = i - gap;
            while (j >= 0 && nums[j] > temp)
            {
                nums[j + gap] = nums[j];
                j -= gap;
            }
            nums[j + gap] = temp;
        }
        gap /= 2;
    }
}
```

希尔排序是插入排序的一种更高效的改进版本，也称为缩小增量排序。它通过比较相隔一定间隔的元素，然后逐渐减小这个间隔，最后当间隔为0时，就变成了普通的插入排序。

时间复杂度分析：

- 最好情况：如果输入数组已经是排序好的，那么时间复杂度为 O(n)。

- 最坏情况：如果输入数组是完全逆序的，那么时间复杂度为 O(n^2)。
- 平均情况：希尔排序的平均时间复杂度为 O(n^(3/2))。

heapSort

```
void heapAdjust(int i, int length)
{
    int temp = nums[i];
    for (int k = 2 * i + 1; k < length; k = 2 * k + 1)
    {
        if (k + 1 < length && nums[k] < nums[k + 1])
            k++;
        if (nums[k] > temp)
        {
            nums[i] = nums[k];
            i = k;
        }
        else
            break;
    }
    nums[i] = temp;
}
void heapSort(int length)
{
    // 构建大顶堆
    for (int i = length / 2 - 1; i >= 0; i--)
        heapAdjust(i, length);
    // 交换堆顶元素和最后一个元素
    for (int i = length - 1; i > 0; i--)
    {
        int temp = nums[i];
        nums[i] = nums[0];
        nums[0] = temp;
        heapAdjust(0, i);
    }
}
```

堆排序的基本思想是将一个无序数组构建成一个大顶堆（或小顶堆），然后将堆顶元素（最大值或最小值）与堆尾元素互换，之后将剩余的元素重新调整为大顶堆（或小顶堆），以此类推，直到整个数组有序。

时间复杂度分析：

- 最好情况：当输入数组已排序时，时间复杂度为 O(nlogn)。
- 最坏情况：当输入数组完全逆序时，时间复杂度为 O(nlogn)。
- 平均情况：时间复杂度为 O(nlogn)。

radixSort

```
void radixSort(int length)
{
    // 获取数组中的最大值，以确定需要进行多少次排序
    int max = nums[0];
    for (int i = 1; i < length; i++)
    {
        if (nums[i] > max)
            max = nums[i];
```

```
    }

    // 获取最大值的位数
    int max_digits = 0;
    while (max != 0)
    {
        max /= 10;
        max_digits++;
    }
    int *tmp = new int[length];                 //临时数组
    int *count = new int[10];                   //统计数组，统计某一位数字相同的个数
    int *start = new int[10];                   //起始索引数组，某一位数字相同数字的第一个的位置
    int base=1;

    while(max_digits--){
        memset(count, 0, 10 * sizeof(int));//每一次都全初始化为0
        //不可以写sizeof(count),这是指针的大小(若为64位，则为8),和普通数组的数组名不一样
        for(int i = 0; i < length; i++){
            int index = nums[i] / base % 10;  //每一位数字
            count[index]++;
        }

        memset(start, 0, 10 * sizeof(int));//每一次都全初始化为0
        for(int i = 1; i < 10; i++)
            start[i] = count[i - 1] + start[i - 1];

        memset(tmp, 0, length * sizeof(int));   //每一次都全初始化为0
        for(int i = 0; i < length; i++){
            int index = nums[i] / base % 10;
            tmp[start[index]++] = nums[i];    //某一位相同的数字放到临时数组中合适的位置
        }

        memcpy(nums, tmp, length * sizeof(int));   //复制tmp中的元素到a
        base *= 10;                             //比较下一位
    }

    delete[] tmp;                               //释放空间
    delete[] count;
    delete[] start;
}
```

基数排序是一种非比较整数排序算法，其原理是将整数按位数切割成不同的数字，然后按每个位数分别比较。

时间复杂度分析：

- 最好情况：当输入数组已排序时，时间复杂度为 O(n)。
- 最坏情况：当输入数组完全逆序时，时间复杂度为 O(n)。
- 平均情况：时间复杂度为 O(n)。

quickSort

```
void quickSort(int length, int left, int right)
{
```

```
    if (left >= right)
        return;
    int pivot = nums[left]; // 选择第一个元素作为基准值
    int i = left, j = right;
    while (i < j)
    {
        while (i < j && nums[j] >= pivot)
            j--;
        nums[i] = nums[j];

        while (i < j && nums[i] <= pivot)
            i++;
        nums[j] = nums[i];
    }
    nums[i] = pivot;                  // 将基准值放到正确的位置
    quickSort(length, left, i - 1);  // 递归对基准值左侧的子数组进行快速排序
    quickSort(length, i + 1, right); // 递归对基准值右侧的子数组进行快速排序
}
```

快速排序是一种高效的排序算法，其基本思想是分治法。

时间复杂度分析：

- 最好情况：当输入数组已经排序时，时间复杂度为O(n)。
- 最坏情况：当输入数组完全逆序时，时间复杂度为O(n^2)。
- 平均情况：时间复杂度为O(n log n)。

QuickSort

```
int Partition(int low, int high) //划分算法
{
    //假设每次都以第一个元素作为枢轴值，进行一趟划分：
    int pivot = nums[low];
    while (low < high)
    {
        while (low < high && nums[high] >= pivot)
            --high;
        nums[low] = nums[high]; //停下来做交换
        while (low < high && nums[low] <= pivot)
            ++low;
        nums[high] = nums[low]; //停下来做交换
    }
    nums[low] = pivot; // pivot的最终落点
    return low;
}
void QuickSort(int length, int left, int right) //非递归快排
{
    //手动利用栈来存储每次分块快排的起始点
    //栈非空时循环获取中轴入栈
    stack<int> s;
    if (left < right)
    {
        int boundary = Partition(left, right);
        if (boundary - 1 > left) //确保左分区存在
        {
            //将左分区端点入栈
            s.push(left);
```

```
            s.push(boundary - 1);
        }
        if (boundary + 1 < right) //确保右分区存在
        {
            s.push(boundary + 1);
            s.push(right);
        }
        while (!s.empty())
        {
            //得到某分区的左右边界
            int r = s.top();
            s.pop();
            int l = s.top();
            s.pop();
            boundary = Partition(l, r);
            if (boundary - 1 > l) //确保左分区存在
            {
                //将左分区端点入栈
                s.push(l);
                s.push(boundary - 1);
            }
            if (boundary + 1 < r) //确保右分区存在
            {
                s.push(boundary + 1);
                s.push(r);
            }
        }
    }
}
```

这段代码实现了非递归版本的快速排序（QuickSort）算法。快速排序是一种高效的排序算法，其基本思想是分治法。

时间复杂度分析：

- 最好情况：当输入数组已经排序时，时间复杂度为O(n)。
- 最坏情况：当输入数组完全逆序时，时间复杂度为O(n^2)。
- 平均情况：时间复杂度为O(n log n)。

MergeSort

```
void Merge(int length, int left, int mid, int right)
{
    int i = left;
    int j = mid + 1;
    int k = 0;
    int *temp = (int *)malloc((right - left + 1) * sizeof(int));
    if (!temp)
    {
        cout << "空间不足！" << endl;
        system("pause");
        exit(0);
    }
    while (i <= mid && j <= right)
    {
        if (nums[i] <= nums[j])
            temp[k++] = nums[i++];
```

```
        else
            temp[k++] = nums[j++];
    }
    while (i <= mid)
        temp[k++] = nums[i++];
    while (j <= right)
        temp[k++] = nums[j++];
    for (i = 0; i < k; i++)
        nums[left + i] = temp[i];
}
void MergeSort(int length, int left, int right)
{
    if (left < right)
    {
        int mid = (left + right) / 2;
        MergeSort(length, left, mid);
        MergeSort(length, mid + 1, right);
        Merge(length, left, mid, right);
    }
}
```

归并排序是一种分治算法，它将一个大问题分解为若干个小问题来解决，然后将这些小问题的解合并起来，形成原问题的解。

时间复杂度为 O(n log n)。

5.测试数据(规模,测试次数)

规模:5e4

测试次数:每个排序算法测试10次

测试用例:随机生成

6.运行结果

见**8.结果截屏图片**

7.时间复杂度分析

见**4.主要/核心函数分析**下的函数分析

8.结果截屏图片

```
输入随机生成样本数据量:
50000
样本数据量大小: 50000
```

排序方式	task-1	task-2	task-3	task-4	task-5	task-6	task-7	task-8	task-9	task-10
插入排序	0	782	281	594	516	578	594	359	328	422
冒泡排序	0	1626	2390	2219	2032	2171	2375	2015	2266	1906
选择排序	516	547	562	453	454	609	516	609	531	485
希尔排序	0	0	0	0	0	0	0	15	0	0
堆排序	0	0	0	0	0	16	0	15	0	0
基数排序	0	0	0	0	0	0	16	0	0	0
快速排序	266	406	0	0	0	0	16	0	0	0
归并排序	0	0	0	0	0	0	0	16	0	15

9.心得体会

对8大排序算法的实现以及不同算法之间的优劣有了更深入的认识与理解。