HW6实验报告

姓名：刘博洋 学号：2153538 日期：2023年3月4日

1. 涉及数据结构和相关背景

排序是计算机科学中最基础的问题，一个排序算法描述确定有序次序的方法，在实际应用中，许多杂乱无章的数据需要通过排序来进行进一步处理，很多高级算法也把排序算法作为其关键子程序。现有的排序算法数量非常庞大，使用的技术也非常丰富，实际上，很多重要的算法设计都体现在近年来研究者设计的排序算法中。同时，在实现排序算法的过程中，可能会依赖一些现实因素，因此在算法的层面处理排序的速度便显得十分重要。常见且重要的算法主要有：简单排序（选择排序、插入排序）、归并排序、快速排序、基数排序等等。

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 数据结构设计

本题如果简单的用选择排序，虽然逻辑上毫无难度，但显然存在时间复杂度过高的问题。因此本题考虑用归并排序来实现，对每一个子问题，都把序列分为左右两部分，左右两部分先分别归并排序，排序完成好，设置两个游标i、j分别指向左右两部分的头部，移动游标直到发现左半部分元素比右半部分大，此时j后面的所有元素都是逆序对，一次比较结束，再移动i直到i移动到左半部分尾部，用一个计数器coun来记录。

2.1.4 功能说明（函数、类）

```
#include<iostream>
using namespace std;
int ans[100010] = {};
int coun=0;
//计算逆序对数(一般的选择法超时）
int cal(int n, int a[])
{
    int i = 0,j = 1;
```

```
    int coun = 0;
    while (i < j)
    {
        if (a[i] > a[j])
        {
            coun++;
        }
        if (j < n - 1)
            j++;
        else
        {
            i++;
            if (i == n - 1)
                break;
            else
                j = i + 1;
        }
    }
    return coun;
}

void mergesort(int a[],int left,int right)
{
    if (left >= right)
        return ;
    int mid = (left + right)/2;
    int i = left, j = mid + 1;
    mergesort(a, left, mid);//左半部分归并排序
    mergesort(a, mid+1, right);//右半部分归并排序
    int k = 0;
    while (i <= mid && j <= right)
    {
        if (a[i] <= a[j])
        {
            ans[k] = a[i];
            i++;
            k++;
        }
        else//出现逆序数 i后面所有的都是逆序对
        {
            coun += mid - i + 1;
            ans[k] = a[j];
            j++;
            k++;
        }

    }
    while (i <= mid)
        ans[k++] = a[i++];
    while (j <= right)
        ans[k++] = a[j++];
    for (int i = left, j = 0;i <= right;i++, j++)
        a[i] = ans[j];
}
int main()
```

```
{
    while (1)
    {
        int n;
        cin >> n;
        if (n == 0)//输入0结束
            break;
        int* a = (int*)malloc(sizeof(int)*(n+1));
        for (int i = 0;i < n;i++)
        {
            cin >> a[i];
        }
        mergesort(a, 0, n-1);
        cout << coun << endl;
        coun = 0;

    }
    return 0;
}
```

2.1.5 调试分析（遇到的问题和解决方法）

本题遇到的问题主要是选择排序发生超时，所以考虑用归并排序，分治策略，把问题分解为很多个子问题，递归求解。

2.1.6 总结和体会

本题主要体会分治策略降低算法的时间复杂度，是理解归并排序和分治算法的一个很好的例题。

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。

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

2.2.3数据结构设计

本题比较简单，利用选择排序的思想，只不过每次比较用两个数字拼凑起来的字符串，设a，b是两个数，利用to_string函数将它们转化为字符串，如果字符串（a+b）>字符串 （b+a），那么就把a排在b的前面，简单选择排序即可实现。

2.2.4功能说明（函数、类）

```
class Solution {
public:
    bool comp(int a, int b)
```

```
    {
        string sa = to_string(a);
        string sb = to_string(b);
        return sa + sb > sb + sa;
    }
    std::string largestNumber(std::vector<int>& nums) {
        // 这里填写你的代码
        int n = nums.size();
        //比较任意两个数构成的字符串，如果a+b>b+a，那么就把a排在b前面，选择排序即可
        for (int i=0;i<n;i++)
        {
            for (int j = i+1;j < n;j++)
            {
                if (comp(nums[i], nums[j]) == 0)
                {
                    int temp = nums[i];
                    nums[i] = nums[j];
                    nums[j] = temp;
                }
             }
        }
        string ans="";
        for (int i = 0;i < n;i++)
        {
            ans += to_string(nums[i]);
        }
        return ans;
    }
};

```

2.2.5调试与分析

本题刚开始没有想到用字符串字典序来比较，而是想拆分数字采用贪心算法来实现，但是一番尝试之后发现过于复杂，重新整理思路，发现直接调用to_string函数转化为字符串用字典序比较可以解决问题

2.2.6总结与收获

本题主要的收获是要转变思维，不能只把数当成数字来看，当成字符串来比较显然更加方便，不能存在思维定势。

2.3排序

本题就是基本的排序问题，不过数据规模和特点未知，本题我先采用了快速排序算法，不过没有得到满分，只有40分，后面六个测试数据显示超时，然后我又尝试了希尔排序，发现得到了满分。由此推测测试数据是具有一定有序性的，导致快速排序选取枢轴元素时两端极度不均衡，容易造成超时。而希尔排序对于本来有一定有序性的数据是比较迅速地。

排序算法的比较在后面给出。

代码：

```
class Solution {
public:
    void Shell_Sort(vector<int>&src, int num) {

```

```
        int gap = num / 2;  //增量
        while (gap) {
            for (int i = gap; i < num; i++) {
                int tmp = src[i];
                int j = i;
                while (j >= gap && tmp < src[j - gap]) {
                    src[j] = src[j - gap];

                    j -= gap;
                }
                src[j] = tmp;
            }
            gap = gap == 2 ? 1 : (int)(gap / 2.2);  //增量变化
        }

    }

    std::vector<int> mySort(std::vector<int>& nums) {
        // 这里填写你的代码
        Shell_Sort(nums, nums.size());
        return nums;
    }
};
```

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

输入描述

```
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
```

输出描述

```
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

根据题目描述，本题主要需要设计算法解决两个问题。

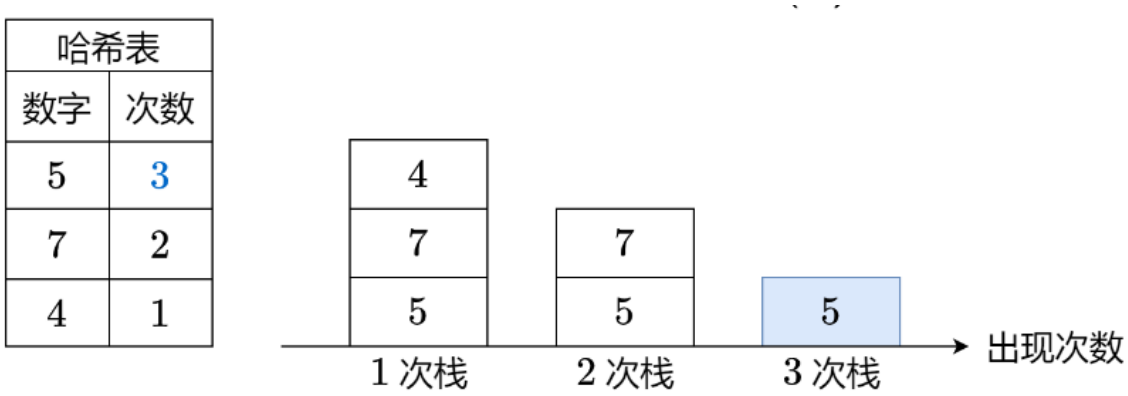
1.如何保存某个数字的出现次数？——采用Map<Integer, Integer> cnt维护

2.如何保存相同出现次数情况下，多个数字集合的“入栈”和“出栈”顺序？——采用 vector<stack> stacks 维护

第一个map保存每个数出现的频率，在入栈时入栈且频率++,弹出元素时

第二个实际上是存放许多栈的容器，每一个栈代表着不同出现不同次数的元素的位置和顺序，我们每次出栈时便调用出现次数最多的栈的顶部元素出栈，直到栈为空再弹出次多次数的栈的顶部元素，以此类推。

找到了LeetCode上题解的一个示意图比较容易理解。



2.4.4功能说明（函数、类）

```
class FreqStack {
    unordered_map<int, int> cnt;
    vector<stack<int>> stacks;
public:
    void push(int val) {
        if (cnt[val] == stacks.size()) // 这个元素的频率已经是目前最多的，现在又出现了一次
            stacks.push_back({}); // 那么必须创建一个新栈
        stacks[cnt[val]].push(val);
        ++cnt[val]; // 更新频率
    }

    int pop() {
        int val = stacks.back().top(); // 弹出最右侧栈的栈顶
        stacks.back().pop();
        if (stacks.back().empty()) // 栈为空
            stacks.pop_back(); // 删除
        --cnt[val]; // 更新频率
        return val;
    }
};
```

2.4.5调试与分析

刚开始想到了用一个键值对数来储存每一个数字出现的频率，但是没有想到要用一个容器来装代表不同频率的栈，于是采用的是每一次要调用一次findmax函数去找到那个最大频率的数，这样的话时间复杂度就变成O（n）了

```cpp
class FreqStack {
private:
    map<int, int> cnt; //键值对 数值--出现次数
    vector<int> s;
    int maxfq;
public:
    FreqStack() {
        // 这里填写你的代码
        maxfq = 0;
    }

    void push(int val) {
        // 这里填写你的代码
        s.push_back(val);
        cnt[val]++;
        maxfq = max(maxfq, cnt[val]);

    }
    void findmax()
    {
        auto it = s.begin();
        while (it < s.end())
        {
            if (cnt[*it] > maxfq)
            {
                maxfq = cnt[*it];
            }

                it++;
        }
    }
    int pop() {
        // 这里填写你的代码
        auto it=s.end()-1;
        while (it>=s.begin())
        {
            if (cnt[*it] == maxfq)
            {
                int a = *it;
                maxfq--;
                cnt[a]--;
                findmax();
                s.erase(it);//这种能过但是超时了
                return a;
            }
            else if(it>s.begin())
                it--;
         }

    }
};
```

2.4.6总结与收获

通过此题发现对于vector的使用还是不够熟练，没有想到还能够存贮栈这种类型的内容。而且本题算法的难度主要在于动态维护最大频率的元素，符合栈的特点，用多个代表不同频率的栈来存储十分恰当，而键值对实际上也是一种哈希表的思想，对于实际的应用我还需要不断提高。

3.排序算法分析

本报告主要测试一下排序算法：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。

输入随机数的个数后用rand()函数生成相应个数的随机数并将其存入容器src中，然后将容器src传入各排序算法的函数，并将容器src赋值给临时的容器temp，这样同一组数据就可以给多个排序算法使用了，保证比较的准确性。调用各排序函数，显示其排序花费的时间、比较次数和交换次数，然后比较分析各排序算法的性能。

衡量排序算法性能的三个指标：

时间复杂度：对排序数据的总的操作次数。

空间复杂度：是指算法在计算机内执行时所需存储空间的度量，它也是数据规模n的函数。

稳定：如果a原本在b前面，而a=b，排序之后a仍然在b的前面。

不稳定：如果a原本在b的前面，而a=b，排序之后 a 可能会出现在 b 的后面。

3.1快速排序QuickSort

先选定枢轴元素pivot作为基准，把数列中比它大的数全部放到右边，比它小的数都放到它左边，然后递归调用，分解成最后只有两个元素排序的子问题，实现整体有序

```
class Solution {
public:
    int partition(vector<int>& nums, int low, int high)//把一组数由枢轴元素分成两断
    {
        int i = low, j = high;
        int center = nums[i];
        while (i < j)
        {
            while (i < j && center <= nums[j])
            {
                j--;
            }
            if (i < j)//游标ij没有相遇
            {
                swap(nums[i], nums[j]);
                i++;
            }
            while (i < j && center >= nums[i])
            {
                i++;
            }
            if (i < j)
            {
                swap(nums[i], nums[j]);
                j--;
```

```
            }
        }
        return i;
    }
    void quicksort(vector<int>& nums, int low, int high)
    {
        int center;
        if (low < high)
        {
            center = partition(nums, low, high);
            quicksort(nums, low, center - 1);
            quicksort(nums, center+1, high);
        }
    }
    std::vector<int> mySort(std::vector<int>& nums) {
        // 这里填写你的代码
        int n = nums.size();
        quicksort(nums, 0, n - 1);
        return nums;
    }
};
```

(1)快速排序的平均时间为O($nlog_2n$)。

(2) 快速排序是递归的，需要有一个栈存放每层递归调用时的指针和参数。

最大递归调用层次数与递归树的深度一致，理想情况为 [$log_2(n+1)$]。 因此,要求存储开销为 $O(log_2n)$。

(3)在最坏的情况，即待排序对象序列已经按其关键码从小到大排好序的情况下，其递归树成为单支树，每次划分只得到一个比上一次少一个对象的子序列。这样，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，总的关键码比较次数将达到

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

其排序速度退化到简单排序的水平，比直接插入排序还慢。占用附加存储(即栈)将达到O(n)。

(4)快速排序是一种不稳定的排序方法

3.2归并排序MergeSort

同样是分治思想的体现，归并排序把数组不断对半分割，直到分割到子序列只有一个元素，此时实现了子序列有序，然后开始归并操作，把两个有序的子序列不断归并合成，最后合成总体有序的序列。

```
void mergesort(int a[],int left,int right)
{
    if (left >= right)
        return ;
    int mid = (left + right)/2;
    int i = left, j = mid + 1;
    mergesort(a, left, mid);//左半部分归并排序
    mergesort(a, mid+1, right);//右半部分归并排序
    int k = 0;
    while (i <= mid && j <= right)
```

```
    {
        if (a[i] <= a[j])
        {
            ans[k] = a[i];
            i++;
            k++;
        }
        else
        {
            ans[k] = a[j];
            j++;
            k++;
        }

    }
    while (i <= mid)
        ans[k++] = a[i++];
    while (j <= right)
        ans[k++] = a[j++];
    for (int i = left, j = 0;i <= right;i++, j++)
        a[i] = ans[j];
}
```

(1) 递归的归并排序方法的递归深度为O($log_2 n$)， 对象关键码的比较次数为O($nlog_2 n$)，因此算法总的时间复杂度为O($nlog_2 n$)。

(2) 归并排序占用附加存储较多，需要另外一个与原待排序对象数组同样大小的辅助数组。这是这个算法的缺点。

(3) 归并排序是一个稳定的排序方法。

3.3堆排序HeapSort

(1)先将原来的数组初始化为一个最大堆；

(2)把堆首(最大值)和堆尾互换；

(3)把堆的大小缩小1，并调用adjust函数向下调整，目的是把新的数组顶端数据调整到相应位置；

(4)重复步骤(2)（3），直到堆的大小为1。

```
void HeapAdjust(int* arr, int start, int end)
{
    int tmp = arr[start];
    for (int i = 2 * start + 1; i <= end; i = i * 2 + 1)
    {
        if (i < end&& arr[i] < arr[i + 1])//有右孩子并且左孩子小于右孩子
        {
            i++;
        }//i一定是左右孩子的最大值
        if (arr[i] > tmp)
        {
            arr[start] = arr[i];
            start = i;
        }
        else
```

```
        {
            break;
        }
    }
    arr[start] = tmp;
}
void HeapSort(int* arr, int len)
{
    //建立最大堆，从后往前依次调整
    for(int i=(len-1-1)/2;i>=0;i--)
    {
        HeapAdjust(arr, i, len - 1);
    }
    //每次将根和待排序的最后一次交换，然后在调整
    int tmp;
    for (int i = 0; i < len - 1; i++)
    {
        tmp = arr[0];
        arr[0] = arr[len - 1-i];
        arr[len - 1 - i] = tmp;
        HeapAdjust(arr, 0, len - 1-i- 1);
    }
}
```

(1)调用了n-1次FilterDown()算法，该循环的计算时间为O($nlog_2n$)。因此,堆排序的时间复杂性为O()。

(2) 该算法的附加存储主要是在用来执行对象交换时所用的一个临时对象。因此，该算法的空间复杂性为O(1)。

(3)堆排序是一个不稳定的排序方法。

3.4选择排序

首先在未排序序列中找到最小元素，存放到排序序列的起始位置再从剩余未排序元素中继续寻找最小元素，然后放到已排序序列的末尾,重复第二步，直到所有元素均排序完毕。

```
void Selection_Sort(vector<int>src, int num) {
  vector<int>temp = src;
  start = clock();
  for (int i = 0; i < num; i++) {
  int min_index = i; //存放最小值的位置
  for (int j = i + 1; j < num; j++) {
  if (temp[j] < temp[min_index]
  min_index = j;  //若有比最小值的位置，就赋给min_index
  Count_cmp++;
     }
     if (min_index != i) {
       swap(temp[i], temp[min_index]);
    }
  }
}
```

(1)**最好情况**和**最坏情况**下的时间复杂度都是$O(n^2)$，无论最好还是最坏，每次选代都要从未排序区间中找到最小值，每次找最小值的时间复杂度为O(n)，所以无论哪种情况，选择排序的时间复杂度都是$O(n^2)$。

(2)空间复杂度为O(1)。

(3)选择排序是不稳定的排序方法

3.5直接插入排序(Insert_Sort()函数)

将第一待排序序列第一个元素看做一个有序序列，把第二个元素到最后一个元素当成是未排序序列。从头到尾依次扫描未排序序列，将扫描到的每个元素插入有序序列的适当位置。(如果待插入的元素与有序序列中的某个元素相等，则将待插入元素插入到相等元素的后面)

```
//直接插入排序
void Insert_Sort(vector<int>src, int num) {
  vector<int>temp = src;
  clock_t start, finish;
  double totaltime = 0; //记录总的时间
  int itemp = 0, i = 0, j = 0;
  start = clock();
  for ( i = 1; i < num; i++) {
     if (temp[i] < temp[i - 1]) {
       itemp = temp[i];
       for ( j = i - 1; j >= 0 && temp[j] > itemp; j--) {
         temp[j + 1] = temp[j];
         Count_cmp++;
         Count_exchange++;
       }
       temp[j + 1] = itemp;
       Count_exchange++;
     }
  }
}
```

(3)若待排序对象序列中出现各种可能排列的概率相同，则可取上述最好情况和最坏情况的平均情况。在平均情况下的关键码比较次数和对象移动次数约为$n^2/4$。因此，直接插入排序的时间复杂度为$O(n^2)$。

(4)空间复杂度为O(1);
(5)直接插入排序是一种**稳定**的排序方法。

3.6冒泡排序

比较相邻的元素，如果第一个比第二个大，就交换他们两个；对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这步做完后，最后的元素会是最大的数。针对所有的元素重复以上的步骤，除了最后一个持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

```
//冒泡排序

void Bubble_Sort(vector<int>src, int num) {
  vector<int>temp = src;
  clock_t start, finish;
  int flag = 1;    //flag用来表示一趟中是否有发生交换，若没有则为1
```

```
  double totaltime = 0; //记录总的时间
  start = clock(); //开始计时
  for (int i = 0; i < num; i++) {
     flag = 1;
     for (int j = 0; j < num - i - 1; j++) {
       if (temp[j] > temp[j + 1]) {
         //如果前一个比后一个小就交换
        swap(temp[j], temp[j + 1]);
         flag = 0;
         Count_exchange += 3;
       }
       Count_cmp++;
     }
     if (flag == 1)  //flag为1说明这趟没有发生交换，序列已经按照顺序排序了
       break;
  }
}
```

时间复杂度为$O(n^2)$；空间复杂度为 O(1)；是一种稳定的排序方法

3.7希尔排序

希尔排序主要是通过不同的增量序列，每次折半对子序列进行直接插入排序，直到gap=1把整个表一起排序实现有序

```
class Solution {
public:
    void Shell_Sort(vector<int>&src, int num) {
        int gap = num / 2;  //增量
        while (gap) {
            for (int i = gap; i < num; i++) {
                int tmp = src[i];
                int j = i;
                while (j >= gap && tmp < src[j - gap]) {
                    src[j] = src[j - gap];
                    j -= gap;
                }
                src[j] = tmp;
            }
            gap = gap == 2 ? 1 : (int)(gap / 2.2);  //增量变化
        }
    }
    std::vector<int> mySort(std::vector<int>& nums) {
        Shell_Sort(nums, nums.size());
        return nums;
    }
};
```

希尔排序是一个复杂的过程，Knuth利用大量的实验统计资料得出，当 n很大时，关键码平均比较次数和对象平均移动次数大约在$n^{1.25}$ 到 1.6的范围内。这是在利用直接插入排序作为子序列排序方法的情况下得到的。

希尔排序的空间复杂度为O(1),是一个不稳定的排序。

3.9排序算法性能总结

	排序类别	时间复杂度	空间复杂度	稳定性
1	插入排序	O(n2)	1	√
2	希尔排序	O(n2)	1	×
3	冒泡排序	O(n2)	1	√
4	选择排序	O(n2)	1	×
5	快速排序	O(Nlogn)	O(logn)	×
6	堆排序	O(Nlogn)	1	×
7	归并排序	O(Nlogn)	O(n)	√

4.项目测试

100个随机数　　1000个随机数　　10000个随机数

```
冒泡排序所用时间：0
冒泡排序比较次数：4905
冒泡排序交换次数：7743

选择排序所用时间：0
选择排序比较次数：4950
选择排序交换次数：279

直接插入排序所用时间：0
直接插入排序比较次数：2581
直接插入排序交换次数：2675

希尔排序所用时间：0
希尔排序比较次数：377
希尔排序交换次数：790

快速排序所用时间：0.001
快速排序比较次数：631
快速排序交换次数：951

堆排序所用时间：0
堆排序比较次数：519
堆排序交换次数：932

归并排序所用时间：0
归并排序比较次数：672
归并排序交换次数：672

基数排序所用时间：0
基数排序比较次数：0
基数排序交换次数：0
```

```
冒泡排序所用时间：0.027
冒泡排序比较次数：499004
冒泡排序交换次数：761541

选择排序所用时间：0.01
选择排序比较次数：499500
选择排序交换次数：2976

直接插入排序所用时间：0.009
直接插入排序比较次数：253847
直接插入排序交换次数：254843

希尔排序所用时间：0.001
希尔排序比较次数：6542
希尔排序交换次数：13632

快速排序所用时间：0
快速排序比较次数：11470
快速排序交换次数：14811

堆排序所用时间：0.001
堆排序比较次数：8438
堆排序交换次数：12574

归并排序所用时间：0.001
归并排序比较次数：9976
归并排序交换次数：9976

基数排序所用时间：0.001
基数排序比较次数：0
基数排序交换次数：0
```

```
冒泡排序所用时间：2.647
冒泡排序比较次数：49980804
冒泡排序交换次数：75071526

选择排序所用时间：1.078
选择排序比较次数：49995000
选择排序交换次数：29973

直接插入排序所用时间：0.809
直接插入排序比较次数：25023842
直接插入排序交换次数：25033833

希尔排序所用时间：0.007
希尔排序比较次数：100739
希尔排序交换次数：201583

快速排序所用时间：0.007
快速排序比较次数：148787
快速排序交换次数：217392

堆排序所用时间：0.008
堆排序比较次数：117678
堆排序交换次数：159228

归并排序所用时间：0.009
归并排序比较次数：133616
归并排序交换次数：133616

基数排序所用时间：0.003
基数排序比较次数：0
基数排序交换次数：0
```