算法设计与分析——排序算法浅析

2018211302 班 2018210074 熊宇

目录

各典型排序算法介绍及分析 **2**
插入排序 2
思想及方法 2
实现 2
合并排序 3
思想及方法 3
实现 4
快速排序 6
思想及方法 6
实现 7
冒泡排序 8
思想及方法 8
实现 9
选择排序 9
思想及方法 9
实现 10
希尔排序 10
思想及方法 10
实现 12
堆排序 13
思想及方法 13
实现 14
总结 15
测试数据生成程序 **16**
实验程序 **17**
实验结果与分析 **27**
心得体会 **39**

一、各典型排序算法介绍及分析

1. 插入排序

（1） 思想及方法

直接插入排序（straight insertion sort），有时也简称为插入排序（insertion sort），是减治法的一种典型应用。其基本思想如下：

① 对于一个数组 A[0,n]的排序问题，假设认为数组在 A[0,n-1]排序的问题已经解决了。

② 考虑 A[n]的值，从右向左扫描有序数组 A[0,n-1]，直到第一个小于等于 A[n]的元素，将 A[n]插在这个元素的后面。

很显然，基于增量法的思想在解决这个问题上拥有更高的效率。

直接插入排序对于最坏情况（严格递减的数组），需要比较和移位的次数为 n(n-1)/2；对于最好的情况（严格递增的数组），需要比较的次数是 n-1，需要移位的次数是 0。当然，对于最好和最坏的研究其实没有太大的意义，因为实际情况下，一般不会出现如此极端的情况。然而，直接插入排序对于基本有序的数组，会体现出良好的性能，这一特性，也给了它进一步优化的可能性。（希尔排序）。直接插入排序的时间复杂度是 O(n^2)，空间复杂度是 O(1)，同时也是稳定排序。

（2） 代码实现

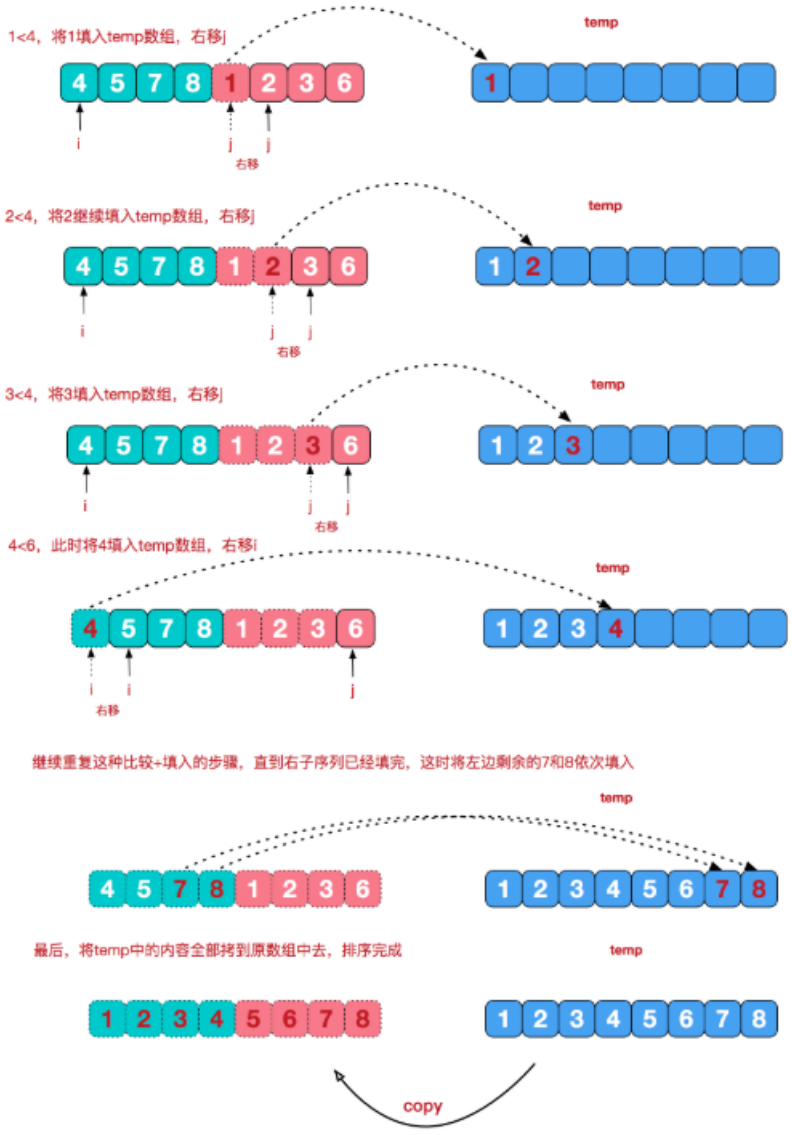
```
//插入排序
void InsertSort(int *h, size_t len)
{
    int pc = 0;
    if (h == NULL)
    {
        pc++;
        return;
    }
    if (len <= 1)
    {
        pc++;
        return;
    }

    int i, j;
    //i 是次数，也即排好的个数;j 是继续排
    for (i = 1; i < len; ++i)
    {
        pc++;
        for (j = i; j > 0; --j)
        {
            pc++;
            if (h[j] < h[j - 1])
            {
                Swap(h[j], h[j - 1]);
```

```
                pc++;
                pc++; //判断和交换
            }
            else
                break;
        }
    }

    cout << "Insert Sort Time:" << pc << endl;
    return;
}
```

2. 合并排序

（1） 思想及方法

归并排序（MERGE-SORT）是利用归并的思想实现的排序方法，该算法采用经典的分治（divide-and-conquer）策略（分治法将问题分(divide)成一些小的问题然后递归求解，而治(conquer)的阶段则将分的阶段得到的各答案"修补"在一起，即分而治之)。



（2） 代码实现

```
//合并排序
void MergeArray(int *arr, size_t left, size_t mid, size_t right, int
*temp, int &pc)
{
    if (arr == NULL)
    {
        pc++;
        return;
    }

    size_t i = left, j = mid + 1, k = 0;
    pc += 3;
    while (i <= mid && j <= right)
    {
        if (arr[i] <= arr[j])
```

```
        {
            temp[k++] = arr[i++];
            pc++;
            pc++;
            continue;
        }

        temp[k++] = arr[j++];
        pc++;
        pc++; //while 判断次数
    }

    while (i <= mid)
    {
        temp[k++] = arr[i++];
        pc++;
        pc++;
    }

    while (j <= right)
    {
        temp[k++] = arr[j++];
        pc++;
        pc++;
    }

    memcpy(&arr[left], temp, k * sizeof(int));

    return;
}

void MMergeSort(int *arr, size_t left, size_t right, int *temp, int
&pc)
{
    if (left < right)
    {
        size_t mid = (left + right) / 2;
        pc++;
        MMergeSort(arr, left, mid, temp, pc);
        MMergeSort(arr, mid + 1, right, temp, pc);
        MergeArray(arr, left, mid, right, temp, pc);
        pc++;
    }
}

void MergeSort(int *h, size_t len)
{
    int pc = 0;
    if (h == NULL)
```

```
    {
        pc++;
        return;
    }
    if (len <= 1)
    {
        pc++;
        return;
    }
    int *temp = (int *)calloc(len, sizeof(int));
    MMergeSort(h, 0, len - 1, temp, pc);

    memcpy(h, temp, sizeof(int) * len);

    free(temp);

    cout << "Merge Sort Time:" << pc << endl;
    return;
}
```

3. 快速排序

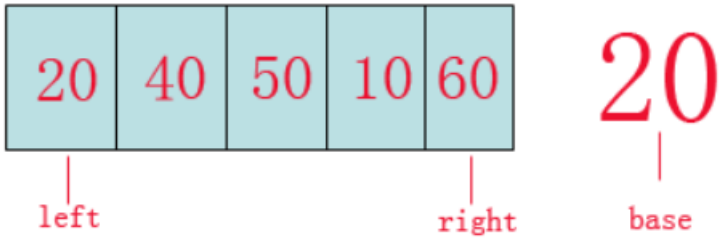
（1）思想及方法

假设我们现在对“6 1 2 7 9 3 4 5 10 8”这个 10 个数进行排序。首先在这个序列中随便找一个数作为基准数（不要被这个名词吓到了，就是一个用来参照的数，待会你就知道它用来做啥的了）。为了方便，就让第一个数 6 作为基准数吧。接下来，需要将这个序列中所有比基准数大的数放在 6 的右边，比基准数小的数放在 6 的左边，类似下面这种排列：

3 1 2 5 4 6 9 7 10 8

在初始状态下，数字 6 在序列的第 1 位。我们的目标是将 6 挪到序列中间的某个位置，假设这个位置是 k。现在就需要寻找这个 k，并且以第 k 位为分界点，左边的数都小于等于 6，右边的数都大于等于 6，递归对左右两个区间进行同样排序即可。想一想，你有办法可以做到这点吗？这就是快速排序所解决的问题。

快速排序是 C.R.A.Hoare 于 1962 年提出的一种划分交换排序。它采用了一种分治的策略，通常称其为分治法(Divide-and-ConquerMethod)。它的平均时间复杂度为 O(nlogn)，最坏时间复杂度为 O(n^2).



首先上图：

从图中我们可以看到：left 指针，right 指针，base 参照数。

其实思想是蛮简单的，就是通过第一遍的遍历（让 left 和 right 指针重合）来找到数组的切割点。

第一步：首先我们从数组的 left 位置取出该数（20）作为基准（base）参照物。(如果是选取随机的，则找到随机的哨兵之后，将它与第一个元素交换，开始普通的快排)

第二步：从数组的 right 位置向前找，一直找到比（base）小的数，如果找到，将此数赋给 left 位置（也就是将 10 赋给 20），此时数组为：10，40，50，10，60， left 和 right 指针分别为前后的 10。

第三步：从数组的 left 位置向后找，一直找到比（base）大的数，如果找到，将此数赋给 right 的位置（也就是 40 赋给 10），此时数组为：10，40，50，40，60， left 和 right 指针分别为前后的 40。

第四步：重复“第二,第三“步骤，直到 left 和 right 指针重合，最后将（base）放到 40 的位置， 此时数组值为： 10，20，50，40，60，至此完成一次排序。

第五步：此时 20 已经潜入到数组的内部，20 的左侧一组数都比 20 小，20 的右侧作为一组数都比 20 大， 以 20 为切入点对左右两边数按照"第一，第二，第三，第四"步骤进行，最终快排大功告成。

（2）代码实现

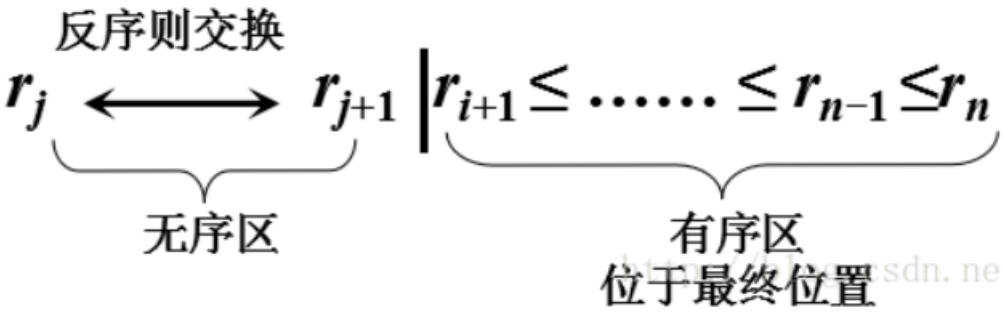
```
//快速排序，随机选取哨兵放前面
void QuickSort(int *h, int left, int right)
{
    int pc = 0;
    if (h == NULL)
    {
        pc++;
        return;
    }
    if (left >= right)
    {
        pc++;
        return;
    }

    //防止有序队列导致快速排序效率降低
    srand((unsigned)time(NULL));
    int len = right - left;
    int kindex = rand() % (len + 1) + left;
    Swap(h[left], h[kindex]);
    pc += 3;

    int key = h[left], i = left, j = right;
    pc += 3;
    while (i < j)
    {
        while (h[j] >= key && i < j)
        {
            --j;
            pc++;
```

```
        }
        if (i < j)
        {
            h[i] = h[j];
            pc++;
            pc++;
        }
        while (h[i] < key && i < j)
        {
            ++i;
            pc++;
            pc++;
        }
        if (i < j)
        {
            h[j] = h[i];
            pc++;
            pc++;
        }
        pc++;
    }

    h[i] = key;
    pc++;

    //QuickSort(&h[left],0,i-1);
    //QuickSort(&h[j+1],0,right-j-1);

    QuickSort(h, left, i - 1);
    QuickSort(h, j + 1, right);
    cout << "Quick Sort Time:" << pc << endl;
}
```

4. 冒泡排序

（1） 思想及方法

冒泡排序在扫描过程中两两比较相邻记录，如果反序则交换，最终，最大记录就被“沉到”了序列的最后一个位置，第二遍扫描将第二大记录“沉到”了倒数第二个位置，重复上述操作，直到 n-1 遍扫描后，整个序列就排好序了。



（2） 代码实现

```
//冒泡排序
void BubbleSort(int *h, size_t len)
{
    int pc = 0;
    if (h == NULL)
    {
        pc++;
        return;
    }
    if (len <= 1)
    {
        pc++;
        return;
    }
    //i 是次数，j 是具体下标
    for (int i = 0; i < len - 1; ++i)
    {
        for (int j = 0; j < len - 1 - i; ++j)
        {
            if (h[j] > h[j + 1])
            {
                Swap(h[j], h[j + 1]);
                pc++;
                pc++;
            }
            pc++;
        }
        pc++;
    }

    cout << "Bubble Sort Time:" << pc << endl;
    return;
}
```

5. 选择排序

（1） 思想及方法

选择排序也是一种简单直观的排序算法。它的工作原理很容易理解：初始时在序列中找到最小（大）元素，放到序列的起始位置作为已排序序列；然后，再从剩余未排序元素中继续寻找最小（大）元素，放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

注意选择排序与冒泡排序的区别：冒泡排序通过依次交换相邻两个顺序不合法的元素位置，从而将当前最小（大）元素放到合适的位置；而选择排序每遍历一次都记住了当前最小（大）元素的位置，最后仅需一次交换操作即可将其放到合适的位置。

（2） 代码实现

```
//选择排序
void SelectionSort(int *h, size_t len)
{
    int pc = 0;
    if (h == NULL)
    {
        pc++;
        return;
    }
    if (len <= 1)
    {
        pc++;
        return;
    }

    int minindex, i, j;
    //i 是次数，也即排好的个数;j 是继续排
    for (i = 0; i < len - 1; ++i)
    {
        minindex = i;
        pc++;
        for (j = i + 1; j < len; ++j)
        {
            if (h[j] < h[minindex])
            {
                minindex = j;
                pc++;
                pc++;
            }
            pc++;
        }
        Swap(h[i], h[minindex]);
        pc++;
        pc++;
    }

    cout << "Select Sort Time:" << pc << endl;

    return;
}
```

6. 希尔排序

（1） 思想及方法

希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量

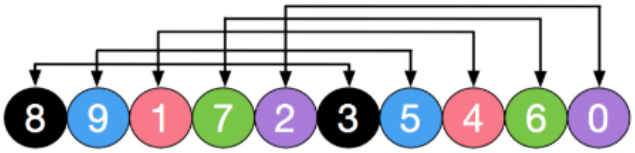
减至 1 时，整个文件恰被分成一组，算法便终止。

简单插入排序很循规蹈矩，不管数组分布是怎么样的，依然一步一步的对元素进行比较，移动，插入，比如[5,4,3,2,1,0]这种倒序序列，数组末端的 0 要回到首位置很是费劲，比较和移动元素均需 n-1 次。而希尔排序在数组中采用跳跃式分组的策略，通过某个增量将数组元素划分为若干组，然后分组进行插入排序，随后逐步缩小增量，继续按组进行插入排序操作，直至增量为 1。希尔排序通过这种策略使得整个数组在初始阶段达到从宏观上看基本有序，小的基本在前，大的基本在后。然后缩小增量，到增量为 1 时，其实多数情况下只需微调即可，不会涉及过多的数据移动。

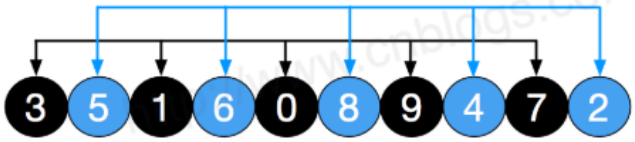
我们来看下希尔排序的基本步骤，在此我们选择增量 gap=length/2，缩小增量继续以 gap = gap/2 的方式，这种增量选择我们可以用一个序列来表示，{n/2,(n/2)/2...1}，称为增量序列。希尔排序的增量序列的选择与证明是个数学难题，我们选择的这个增量序列是比较常用的，也是希尔建议的增量，称为希尔增量，但其实这个增量序列不是最优的。

原始数组 以下数据元素颜色相同为一组

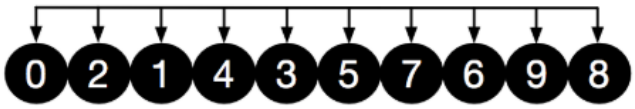
初始增量 gap=length/2=5，意味着整个数组被分为5组，[8,3] [9,5] [1,4] [7,6] [2,0]



对这5组分别进行直接插入排序，结果如下，可以看到，像3，5，6这些小元素都被调到前面了，
然后缩小增量 gap=5/2=2,数组被分为2组 [3,1,0,9,7] [5,6,8,4,2]



对以上2组再分别进行直接插入排序，结果如下，可以看到，此时整个数组的有序程度更进一步啦。
再缩小增量gap=2/2=1，此时，整个数组为1组[0,2,1,4,3,5,7,6,9,8]，如下



经过上面的“宏观调控”，整个数组的有序化程度成果喜人。
此时，仅仅需要对以上数列简单微调，无需大量移动操作即可完成整个数组的排序。

（2） 代码实现

```
//希尔排序
void ShellSort(int *h, size_t len)
{
    int pc = 0;
    if (h == NULL)
    {
        pc++;
        return;
    }
    if (len <= 1)
    {
        pc++;
        return;
    }
```

```
    for (int div = len / 2; div >= 1; div /= 2)
    {
        pc++;
        for (int k = 0; k < div; ++k)
        {
            pc++;
            for (int i = div + k; i < len; i += div)
            {
                pc++;
                for (int j = i; j > k; j -= div)
                {
                    if (h[j] < h[j - div])
                        {
                            Swap(h[j], h[j - div]);
                            pc++;
                        }
                    else
                        break;
                    pc++;
                    pc++;
                }
            }
        }
    }

    cout << "Shell Sort Time:" << pc << endl;
    return;
}
```

7. 堆排序（也属选择排序）

（1） 思想及方法

堆排序实际上是利用堆的性质来进行排序的，要知道堆排序的原理我们首先一定要知道什么是堆。

堆的定义：堆实际上是一棵完全二叉树。

堆满足两个性质：①堆的每一个父节点都大于（或小于）其子节点；
②堆的每个左子树和右子树也是一个堆。

堆的分类：①最大堆（大顶堆）：堆的每个父节点都大于其孩子节点；
②最小堆（小顶堆）：堆的每个父节点都小于其孩子节点；

堆的存储：一般都用数组来表示堆，i 结点的父结点下标就为(i – 1) / 2。它的左右子结点下标分别为 2 * i + 1 和 2 * i + 2。

堆排序：由上面的介绍我们可以看出堆的第一个元素要么是最大值(大顶堆)，要么是最小值（小顶堆），这样在排序的时候（假设共 n 个节点），直接将第一个元素和最后一个元素进行交换，然后从第一个元素开始进行向下调整至第 n-1 个元素。所以，如果需要升序，就建一个大堆，需要降序，就建一个小堆。

三步走：①建堆（升序建大堆，降序建小堆）；

②交换数据；
③向下调整。

（2） 代码实现

```
//堆排序
/*
大顶堆 sort 之后，数组为从小到大排序
*/
//====调整=====
void AdjustHeap(int *h, int node, int len,int &pc) //----node 为需要调整的结点编号，从 0 开始编号;len 为堆长度
{
    int index = node;
    int child = 2 * index + 1; //
    pc+=2;
    while (child < len)
    {
        pc++;
        //右子树
        if (child + 1 < len && h[child] < h[child + 1])
        {
            pc++;
            child++;
            pc++;
        }
        if (h[index] >= h[child])
            {
                pc++;
                break;
            }
        Swap(h[index], h[child]);
        pc++;
        index = child;
        child = 2 * index + 1;
        pc++;
        pc++;
    }
}

//====建堆=====
void MakeHeap(int *h, int len,int &pc)
{
    for (int i = len / 2; i >= 0; --i)
    {
        pc++;
        pc++;
        AdjustHeap(h, i, len,pc);
    }
```

```
}

//====排序=====
void HeapSort(int *h, int len)
{
    int pc=0;
    MakeHeap(h, len,pc);
    for (int i = len - 1; i >= 0; --i)
    {
        Swap(h[i], h[0]);
        AdjustHeap(h, 0, i,pc);
        pc++;
        pc++;
    }
    cout << "Heap Sort Time:" << pc << endl;
}
```

8. 总结

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	稳定
合并排序	$O(nlog_2 n)$	$O(nlog_2 n)$	$O(nlog_2 n)$	稳定
快速排序	$O(nlog_2 n)$	$O(n^2)$	$O(nlog_2 n)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	不稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	不稳定
堆排序	$O(nlog_2 n)$	$O(nlog_2 n)$	$O(nlog_2 n)$	不稳定

【测试数据生成程序】采用了随机数方式的进行生成。

【白盒测试】自己设计测试数据来遍历每个分支，对于排序算法来说，选择不同分支并不会出现严格意义上的错误或者效率差极大。我们分析最坏情况、最好情况、平均情况即可。

【黑盒测试】输出未排序数据，再输出排序数据，两相比对即可。

【分析算法效率】考虑到设备的限制，盲目比较运行时间是很愚昧的行为，这里我们在程序内部设置一个 pc 整型变量，每次执行腾挪操作就让它自增，然后通过比较 pc 来比较腾挪次数，进而进行效率的比较。值得意味的是，由于合并排序不断涉及到自我调用，每次调用都将 pc 清零，因此，我们每处理完一组数据就打印一行空白用以区分。

二、测试数据生成程序

```
#define MaxLineNum 100
#define MaxDataNum 1000

using namespace std;

int numOfNum, numOfLine; //测试数据组数，每组数据量
int TestData[MaxLineNum][MaxDataNum] = {0};

void ProduceTestData()
{
    ofstream OutFile("F:\\AlghorthmDesign\\sort\\TestData.txt"); //利用构造函数创建 txt 文本，并且打开该文本

    cout << "Please input the number of line and the number of data every line." << endl;
    cin >> numOfLine >> numOfNum;
    int arr[numOfNum], T;
    for (int i = 1; i <= numOfNum; i++)
        arr[i] = 0;
    srand(time(NULL));
    int line = 0;
    int tempLine = numOfLine;
    while (tempLine--)
    {

        for (int i = 1; i <= numOfNum; i++)
        {

            arr[i] = rand() % 100;
        }
        for (int i = 1; i <= numOfNum; i++)
        {
            OutFile << arr[i];
            TestData[line][i - 1] = arr[i];
            OutFile << " ";
        }
        OutFile << endl;
        line++;
    }
    OutFile.close();
    cout << "The test data has been produced successfully." << endl;
}
```

三、实验程序

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <algorithm>
#include <cmath>
#include <cstring>
#include <vector>
#include <map>

#define MaxLineNum 100
#define MaxDataNum 1000

using namespace std;

int numOfNum, numOfLine; //测试数据组数，每组数据量
int TestData[MaxLineNum][MaxDataNum] = {0};

void ProduceTestData()
{
    ofstream OutFile("F:\\AlghorthmDesign\\sort\\TestData.txt"); //利用构造函数创建 txt 文本，并且打开该文本

    cout << "Please input the number of line and the number of data every line." << endl;
    cin >> numOfLine >> numOfNum;
    int arr[numOfNum], T;
    for (int i = 1; i <= numOfNum; i++)
        arr[i] = 0;
    srand(time(NULL));
    int line = 0;
    int tempLine = numOfLine;
    while (tempLine--)
    {

        for (int i = 1; i <= numOfNum; i++)
        {

            arr[i] = rand() % 100;
        }
        for (int i = 1; i <= numOfNum; i++)
        {
            OutFile << arr[i];
            TestData[line][i - 1] = arr[i];
            OutFile << " ";
        }
        OutFile << endl;
        line++;
```

```
    }
    OutFile.close();
    cout << "The test data has been produced successfully." << endl;
}

//交换函数
void Swap(int &a, int &b)
{
    int t = a;
    a = b;
    b = t;

    return;
}

//插入排序
void InsertSort(int *h, size_t len)
{
    int pc = 0;
    if (h == NULL)
    {
        pc++;
        return;
    }
    if (len <= 1)
    {
        pc++;
        return;
    }

    int i, j;
    //i 是次数，也即排好的个数;j 是继续排
    for (i = 1; i < len; ++i)
    {
        pc++;
        for (j = i; j > 0; --j)
        {
            pc++;
            if (h[j] < h[j - 1])
            {
                Swap(h[j], h[j - 1]);
                pc++;
                pc++; //判断和交换
            }
            else
                break;
        }
    }
```

```
    cout << "Insert Sort Time:" << pc << endl;
    return;
}

//合并排序
void MergeArray(int *arr, size_t left, size_t mid, size_t right, int
*temp, int &pc)
{
    if (arr == NULL)
    {
        pc++;
        return;
    }

    size_t i = left, j = mid + 1, k = 0;
    pc += 3;
    while (i <= mid && j <= right)
    {
        if (arr[i] <= arr[j])
        {
            temp[k++] = arr[i++];
            pc++;
            pc++;
            continue;
        }

        temp[k++] = arr[j++];
        pc++;
        pc++; //while 判断次数
    }

    while (i <= mid)
    {
        temp[k++] = arr[i++];
        pc++;
        pc++;
    }

    while (j <= right)
    {
        temp[k++] = arr[j++];
        pc++;
        pc++;
    }

    memcpy(&arr[left], temp, k * sizeof(int));

    return;
}
```

```
void MMergeSort(int *arr, size_t left, size_t right, int *temp, int
&pc)
{
    if (left < right)
    {
        size_t mid = (left + right) / 2;
        pc++;
        MMergeSort(arr, left, mid, temp, pc);
        MMergeSort(arr, mid + 1, right, temp, pc);
        MergeArray(arr, left, mid, right, temp, pc);
        pc++;
    }
}

void MergeSort(int *h, size_t len)
{
    int pc = 0;
    if (h == NULL)
    {
        pc++;
        return;
    }
    if (len <= 1)
    {
        pc++;
        return;
    }
    int *temp = (int *)calloc(len, sizeof(int));
    MMergeSort(h, 0, len - 1, temp, pc);

    memcpy(h, temp, sizeof(int) * len);

    free(temp);

    cout << "Merge Sort Time:" << pc << endl;
    return;
}

//快速排序，随机选取哨兵放前面
void QuickSort(int *h, int left, int right)
{
    int pc = 0;
    if (h == NULL)
    {
        pc++;
        return;
    }
    if (left >= right)
```

```
{
    pc++;
    return;
}

//防止有序队列导致快速排序效率降低
srand((unsigned)time(NULL));
int len = right - left;
int kindex = rand() % (len + 1) + left;
Swap(h[left], h[kindex]);
pc += 3;

int key = h[left], i = left, j = right;
pc += 3;
while (i < j)
{
    while (h[j] >= key && i < j)
    {
        --j;
        pc++;
    }
    if (i < j)
    {
        h[i] = h[j];
        pc++;
        pc++;
    }
    while (h[i] < key && i < j)
    {
        ++i;
        pc++;
        pc++;
    }
    if (i < j)
    {
        h[j] = h[i];
        pc++;
        pc++;
    }
    pc++;
}

h[i] = key;
pc++;

//QuickSort(&h[left],0,i-1);
//QuickSort(&h[j+1],0,right-j-1);

QuickSort(h, left, i - 1);
```

```
    QuickSort(h, j + 1, right);
    cout << "Quick Sort Time:" << pc << endl;
}

//冒泡排序
void BubbleSort(int *h, size_t len)
{
    int pc = 0;
    if (h == NULL)
    {
        pc++;
        return;
    }
    if (len <= 1)
    {
        pc++;
        return;
    }
    //i 是次数，j 是具体下标
    for (int i = 0; i < len - 1; ++i)
    {
        for (int j = 0; j < len - 1 - i; ++j)
        {
            if (h[j] > h[j + 1])
            {
                Swap(h[j], h[j + 1]);
                pc++;
                pc++;
            }
            pc++;
        }
        pc++;
    }

    cout << "Bubble Sort Time:" << pc << endl;
    return;
}

//选择排序
void SelectionSort(int *h, size_t len)
{
    int pc = 0;
    if (h == NULL)
    {
        pc++;
        return;
    }
    if (len <= 1)
    {
```

```
        pc++;
        return;
    }

    int minindex, i, j;
    //i 是次数，也即排好的个数;j 是继续排
    for (i = 0; i < len - 1; ++i)
    {
        minindex = i;
        pc++;
        for (j = i + 1; j < len; ++j)
        {
            if (h[j] < h[minindex])
            {
                minindex = j;
                pc++;
                pc++;
            }
            pc++;
        }
        Swap(h[i], h[minindex]);
        pc++;
        pc++;
    }

    cout << "Select Sort Time:" << pc << endl;

    return;
}

//希尔排序
void ShellSort(int *h, size_t len)
{
    int pc = 0;
    if (h == NULL)
    {
        pc++;
        return;
    }
    if (len <= 1)
    {
        pc++;
        return;
    }

    for (int div = len / 2; div >= 1; div /= 2)
    {
        pc++;
        for (int k = 0; k < div; ++k)
```

```
        {
            pc++;
            for (int i = div + k; i < len; i += div)
            {
                pc++;
                for (int j = i; j > k; j -= div)
                {
                    if (h[j] < h[j - div])
                        {
                            Swap(h[j], h[j - div]);
                            pc++;
                        }
                    else
                        break;
                    pc++;
                    pc++;
                }
            }
        }
    }

    cout << "Shell Sort Time:" << pc << endl;
    return;
}

//堆排序
/*
大顶堆sort之后，数组为从小到大排序
*/
//====调整=====
void AdjustHeap(int *h, int node, int len,int &pc) //----node为需要调整的结点编号，从0开始编号;len为堆长度
{
    int index = node;
    int child = 2 * index + 1; //
    pc+=2;
    while (child < len)
    {
        pc++;
        //右子树
        if (child + 1 < len && h[child] < h[child + 1])
        {
            pc++;
            child++;
            pc++;
        }
        if (h[index] >= h[child])
            {
                pc++;
```

```
                break;
            }
        Swap(h[index], h[child]);
        pc++;
        index = child;
        child = 2 * index + 1;
        pc++;
        pc++;
    }
}

//====建堆=====
void MakeHeap(int *h, int len,int &pc)
{
    for (int i = len / 2; i >= 0; --i)
    {
        pc++;
        pc++;
        AdjustHeap(h, i, len,pc);
    }
}

//====排序=====
void HeapSort(int *h, int len)
{
    int pc=0;
    MakeHeap(h, len,pc);
    for (int i = len - 1; i >= 0; --i)
    {
        Swap(h[i], h[0]);
        AdjustHeap(h, 0, i,pc);
        pc++;
        pc++;
    }
    cout << "Heap Sort Time:" << pc << endl;
}

int main()
{
    ProduceTestData();
    cout << numOfLine << " " << numOfNum << endl;
    cout << endl;

    //【黑盒测试】输出未排序数据，再输出排序数据，两相比对即可。
    cout << "Unsort" << endl;
    for (int i = 0; i < numOfLine; i++)
    {
        for (int j = 0; j < numOfNum; j++)
            cout << TestData[i][j] << " ";
```

```
        cout << endl;
    }

    cout << endl;

    for (int i = 0; i < numOfLine; i++)
    {
        //插入排序
        //InsertSort(TestData[i],numOfNum);

        //合并排序
        //MergeSort(TestData[i],numOfNum);

        //快速排序
        //QuickSort(TestData[i],0,numOfNum); cout<<"A QuickSort is
over"<<endl<<endl; //存在递归调用，通过这种方式区分计数

        //冒泡排序
        //BubbleSort(TestData[i],numOfNum);

        //选择排序
        //SelectionSort(TestData[i],numOfNum);

        //希尔排序排序
        //ShellSort(TestData[i],numOfNum);

        //堆排序
        //HeapSort(TestData[i],numOfNum);
    }
    cout<<endl;
    cout << "Sort" << endl;
    for (int i = 0; i < numOfLine; i++)
    {
        for (int j = 0; j < numOfNum; ++j)
        {
            cout << TestData[i][j] << " ";
        }
        cout << endl;
    }

    system("pause");
    return 0;
}
```

四、实验结果与分析

这里注释掉测试数据生成函数，手动构造数据，然后分别执行各排序算法，比较他们的腾挪次数，也就是程序里的 pc，从而比较他们的性能。

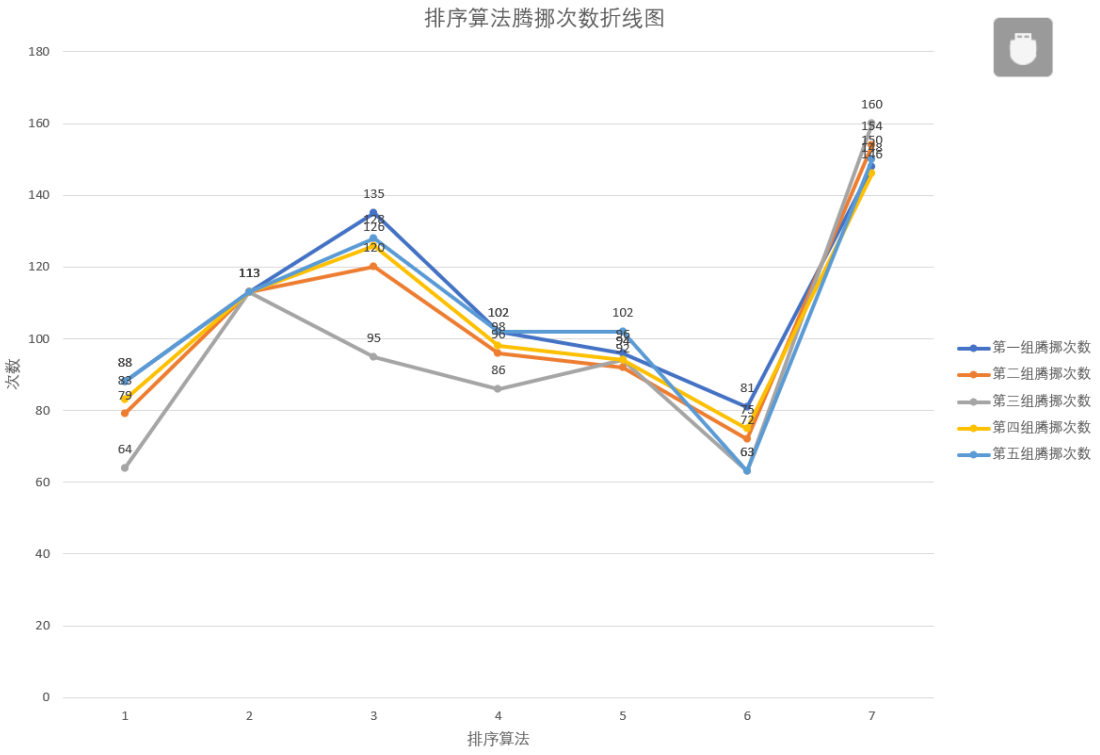
测试数据：

46 5 20 49 23 61 3 13 29 14
54 67 32 78 0 63 91 10 23 78
40 26 25 78 85 86 50 65 95 34
6 79 13 57 9 89 78 71 75 4
47 46 61 75 71 90 49 20 24 51

输出数据：

3 5 13 14 20 23 29 46 49 61
0 10 23 32 54 63 67 78 78 91
25 26 34 40 50 65 78 85 86 95
4 6 9 13 57 71 75 78 79 89
20 24 46 47 49 51 61 71 75 90

排序方法	第一组数据 腾挪次数	第二组数据 腾挪次数	第三组数据 腾挪次数	第四组数据 腾挪次数	第五组数据 腾挪次数
插入排序	88	79	64	83	88
合并排序	113	113	113	113	113
快速排序	135	120	95	126	128
冒泡排序	102	96	86	98	102
选择排序	96	92	94	94	102
希尔排序	81	72	63	75	63
堆排序	148	154	160	146	150


排序算法腾挪次数折线图
180
160
140
120
100
80
60
40
20
0
次数
1
2
3
4
5
6
7
排序算法
第一组腾挪次数
第二组腾挪次数
第三组腾挪次数
第四组腾挪次数
第五组腾挪次数

五、心得体会

经过这次对典型七种排序算法进行理论学习、代码实现，使我对这几种排序算法有了比之前数据结构更加深刻、更加系统化的认知，在使用大量测试数据进行测试的过程中，逐渐认识到对于每一种排序算法，都有它极其适合的使用场景。

【测试数据生成程序】采用了随机数方式的进行生成。

【白盒测试】自己设计测试数据来遍历每个分支，对于排序算法来说，选择不同分支并不会出现严格意义上的错误或者效率差极大。我们分析最坏情况、最好情况、平均情况即可。

【黑盒测试】输出未排序数据，再输出排序数据，两相比对即可。

【分析算法效率】考虑到设备的限制，盲目比较运行时间是很愚昧的行为，这里我们在程序内部设置一个 pc 整型变量，每次执行腾挪操作就让它自增，然后通过比较 pc 来比较腾挪次数，进而进行效率的比较。值得意味的是，由于合并排序不断涉及到自我调用，每次调用都将 pc 清零，因此，我们每处理完一组数据就打印一行空白用以区分。