数据结构课程设计

北京郵電大學

姓　　名 tobeApe6eHok

学　　院 人工智能学院

专　　业 智能科学与技术

班　　级 1111111111

学　　号 1111111111

班内序号 1

指导教师 周延泉

2022 年　6 月

目录

数据结构课程设计实验六

一、实验题目

试编写一个双向冒泡排序算法，即在排序过程中交替改变扫描方向。

二、题目分析与算法设计

1、设计思路：

升序冒泡排序的具体思路是依次将相邻的两个数进行比较，如果下标大的那个数较小，则交换这两个数，这样就可以保证在进行依次循环后，最大的数在数组的最后一个位置上。同理，降序冒泡排序也可以得到，只需要取反交换两个相邻数的条件即可。

传统的升序冒泡排序有两个可以优化的小技巧。首先，因为每一次循环都会保证数组循环范围内的最后一个元素是循环过程中遍历到的最大值，所以下一次循环比较时就不必再进行比较，所以在每次循环后，都可以将循环范围的有边界向左缩小一个数字，这样就会节省一些计较。其次，如果在一次循环比较中，并没有逆序的相邻数字，那么意味着，整个序列已经排好序，此时就可以终止排序算法。

而双向冒泡排序算法是在冒泡排序算法的基础上改进而来的，其基本思想和传统的冒泡排序相同，只不过是双向冒泡排序将算法中将循环比较定义为了正向循环和逆向循环两次循环：首先从前往后将最大的数移动到最后，此时指针指向最后，然后反过来从后往前把最小的一个数移动到数组最前面，只需要再将指针递减即可。

同时，虽然双向冒泡排序做了一些改进，但是基本的思路没有变，所以针对传统冒泡排序的两个小技巧也同样适用。

2、算法描述：

首先，针对第一个小技巧可以定义 LeftRemain 和 RightRemain 两个指针指向还需要进行后续排序的数组范围，在一次正向排序后就可以将 RightRemain 减 1，在一次逆向排序后就可以将 LeftRemain 加 1，代表范围的缩减；其次，在设置哨兵时，因为每次循环分为正向和逆向两个子循环，所以需要设置两个哨兵，无论是正向循环还是逆向循环，只要在此次循环中没有交换元素，就代表数组已经排好序。

综上分析，可以得到双向冒泡排序的伪代码如下：

```
设置待排序的左边界LeftRemain =0
设置待排序的右边界RightRemain =0
设置哨兵Flag
只要 LeftRemain < RightRemain 就代表数组还有待排序的范围:
    从前向后遍历待排序范围内的所有相邻数字:
        如果两个数满足交换的条件 ,则交换
        并且告知哨兵

    如果从哨兵那里得知:
        此次循环中并没有交换相邻元素 ,则终止排序算法
        否则,缩小待排序范围的右边界

    从后向前遍历待排序范围内的所有相邻数字:
        如果两个数字满足交换的条件 ,则交换
        并且告知哨兵

    如果从哨兵那里得知:
        此次循环中并没有交换相邻元素 ,则终止排序算法
        否则,缩小待排序范围的左边界
```

3、函数调用：

在代码实现中，为了实现用户指定升序排序还是降序排序，我对判断交换的条件进行了函数封装，其中函数名是 CompareKey()；用户可以在调用 TowWayBubbleSort()函数进行双向冒泡排序时，指定 Reverse 的值,如果 Reverse==true 代表进行降序排序,CompareKey()函数被 TowWayBubbleSort()函数调用，可以根据 Reverse 的值判断是否交换相邻的元素值。

其次，在双向冒泡排序过程中，我使用了 Printf()函数用来打印排序的过程，以便于分析排序的过程、验证算法的正确性。

三、数据结构描述

1、数据结构的选择：

为了使代码简洁、提高代码的复用性，我将待排序的对象设置成了一个模板结构体，其中可以存储用户待进行排序的数组，其中代码

中默认使用结构体中的 Value 字段进行排序，当然，用户也可以根据需要自行定义其他需要存储的数据字段。

2、数据结构的定义：

```
template<class T>
struct Element
{
    T Value;
                        //按照Value的值大小进行排序

    //结构体中还可以定义其他数据域

    Element():Value(0){}            //无参构造函数
    Element(T value):Value(value){}//结构体的构造函数
};
```

四、程序清单

以下是我调试好的代码（我的运行环境:WIN10+VScode+Cpp20）：

```
#include<iostream>
using namespace std;

//定义需要排序的结构体,默认按照 Value 字段进行排序,用户可以在结构体中自定义其他数据字段
template<class T>
struct Element
{
    T Value;                        //按照 Value 的值大小进行排序

    //结构体中还可以定义其他数据域

    Element():Value(0){}            //无参构造函数
    Element(T value):Value(value){}//结构体的构造函数
};
//打印每次正向、逆向排序的结果,便于检测排序的正确性
//函数调用中如果 Direct<0,表示刚刚进行的是逆向排序
//如果 Direct>0,表示刚刚进行的是正向排序
template<class T>
void Printf(int Direct,int ArraySize,Element<T> Array[])
{
    if(Direct<0)
    {
        cout<<"第"<<-Direct<<"次反向排序结果:"<<"\t";
    }
```

```
    else
    {
        cout<<"第"<<Direct<<"次正向排序结果:"<<"\t";
    }
    for(int p=0;p<ArraySize;p++)
    {
        cout<<Array[p].Value<<" ";
    }
    cout<<endl;
}
//比较函数
//Reverse==true 则表示逆向排序,则 Front.Value<Latter.Value 时交换前后顺序
template<class T>
bool CompareKey(Element<T> &Front,Element<T> &Latter,bool Reverse=false)
{
    return
Reverse?(Front.Value<Latter.Value):(Front.Value>Latter.Value);
}
//双向冒泡排序
//默认是升序
//如果需要逆序排序,则在调用时将 Reverse 设成 true
template<class T>
void TowWayBubbleSort(int ArraySize,Element<T> Array[],bool Reverse=false)
{
    //LeftRemain 和 RightRemain 用来记录待排序数组范围的左右边界
    //Loop 用来记录已经循环的次数
    //Flag 用来充当哨兵
    //P 是循环过程中的指针
    int LeftRemain=0,RightRemain=ArraySize-1,Loop=1,Flag,P;
    while(LeftRemain<RightRemain)
    {
        //设置哨兵,记录是否已经排好序
        Flag=false;
        P=LeftRemain;
        //正向排序
        while(P<RightRemain)
        {
            if(CompareKey(Array[P],Array[P+1],Reverse))//如果满足交换的条
件,则交换,并且告知哨兵
            {
                swap(Array[P],Array[P+1]);
                Flag=true;//说明本次循环中有交换位置,说明还没有排好序
            }
            ++P;
        }
        //显示本次正向排序的结果
        Printf(Loop,ArraySize,Array);
        //Flag==false 表示本次循环比较中没有进行交换,说明数组已经排好序
        if(!Flag)   break;
        --RightRemain,--P,Flag=false;//缩小待排序数组的范围,并且重置哨兵
        //逆向排序
        while(P>LeftRemain)
        {
```

```
            if(CompareKey(Array[P-1],Array[P],Reverse))//如果满足交换的条
件,则交换,并且告知哨兵
            {
                swap(Array[P-1],Array[P]);
                Flag=true;//说明本次循环中有交换位置,说明还没有排好序
            }
            --P;
        }
        //显示本次逆向排序结果
        Printf(-Loop,ArraySize,Array);
        //Flag==false 表示本次循环比较中没有进行交换,说明数组已经排好序
        if(!Flag)   break;
        ++LeftRemain,++Loop,Flag=false;//缩小待排序的数组范围,并且重置哨兵,
记录下一次的循环次数
    }
}
//test1 函数,其中有一个正向排序的测试样例
void test1()
{
    Element<int> nums[10];
    for(int i=9;i>=0;i--)
    {
        nums[9-i]=Element<int>(i);
    }
    TowWayBubbleSort(10,nums);
}
//test2 函数,其中有一个逆向排序的测试样例
void test2()
{
    Element<int> nums[10];
    for(int i=0;i<10;i++)
    {
        nums[i]=Element<int>(i);
    }
    TowWayBubbleSort(10,nums,true);
}
void test3()
{
    Element<int> nums[10];
    int values[]={45,19,77,81,13,28,18,19,77,11};
    for(int i=0;i<10;i++)
    {
        nums[i]=Element<int>(values[i]);
    }
    TowWayBubbleSort(10,nums);
}
int main()
{
    cout<<"\n 第一个测试样例:"<<endl;
    test1();
    cout<<"\n 第二个测试样例:"<<endl;
    test2();
    cout<<"\n 第三个测试样例:"<<endl;
    test3();
}
```

五、程序复杂度分析

1、时间复杂度分析：

双向冒泡排序虽然将数组的两头都排好序了，只需要处理中间一段数组即可，但是双向冒泡排序在每一个大循环中引入了 2 个子循环，所以其实是相当于循环了两次。最坏的情况下，仍是将降（升）序数组进行升（降）序排序，假设数组元素个数是 n，则需要循环 2*（n/2）=n 次，比较次数和冒泡排序相同，都是 n（n-1）/2，所以时间复杂度是 O（n^2）;最好的情况下，是降（升）序数组进行降（升）序排序，此时只需要进行一次循环，发现 Flag==false，终止算法即可，时间复杂度是 O（n）；综上，平均情况来讲，时间复杂度是 O（n^2）。

3、空间复杂度分析：

在算法中只需要几个变量的存储空间，所以空间复杂度是 O（1）。

六、程序运行结果

1、测试样例：
我设置了三个测试样例：
（1）[9,8,7,6,5,4,3,2,1,0] Reverse=false
（2）[0,1,2,3,4,5,6,7,8,9] Reverse=true
（3）[45,19,77,81,13,28,18,19,77,11] Reverse=false
2、测试结果：
结果如下图，可以发现排序的过程和分析的结果相同，可以证明算法的正确性。

```
第一个测试样例:
第1次正向排序结果:      8 7 6 5 4 3 2 1 0 9
第1次反向排序结果:      0 8 7 6 5 4 3 2 1 9
第2次正向排序结果:      0 7 6 5 4 3 2 1 8 9
第2次反向排序结果:      0 1 7 6 5 4 3 2 8 9
第3次正向排序结果:      0 1 6 5 4 3 2 7 8 9
第3次反向排序结果:      0 1 2 6 5 4 3 7 8 9
第4次正向排序结果:      0 1 2 5 4 3 6 7 8 9
第4次反向排序结果:      0 1 2 3 5 4 6 7 8 9
第5次正向排序结果:      0 1 2 3 4 5 6 7 8 9
第5次反向排序结果:      0 1 2 3 4 5 6 7 8 9

第二个测试样例:
第1次正向排序结果:      1 2 3 4 5 6 7 8 9 0
第1次反向排序结果:      9 1 2 3 4 5 6 7 8 0
第2次正向排序结果:      9 2 3 4 5 6 7 8 1 0
第2次反向排序结果:      9 8 2 3 4 5 6 7 1 0
第3次正向排序结果:      9 8 3 4 5 6 7 2 1 0
第3次反向排序结果:      9 8 7 3 4 5 6 2 1 0
第4次正向排序结果:      9 8 7 4 5 6 3 2 1 0
第4次反向排序结果:      9 8 7 6 4 5 3 2 1 0
第5次正向排序结果:      9 8 7 6 5 4 3 2 1 0
第5次反向排序结果:      9 8 7 6 5 4 3 2 1 0

第三个测试样例:
第1次正向排序结果:      19 45 77 13 28 18 19 77 11 81
第1次反向排序结果:      11 19 45 77 13 28 18 19 77 81
第2次正向排序结果:      11 19 45 13 28 18 19 77 77 81
第2次反向排序结果:      11 13 19 45 18 28 19 77 77 81
第3次正向排序结果:      11 13 19 18 28 19 45 77 77 81
第3次反向排序结果:      11 13 18 19 19 28 45 77 77 81
第4次正向排序结果:      11 13 18 19 19 28 45 77 77 81
```