```cpp
/*****************************实现七种内部排序算法*****************************/
//直接插入排序
//基本思想是将第一个数据元素看成是一个有序子序列，再依次从第二个记录起逐个插入到
//这个有序的子序列中
//直接插入排序算法的时间效率是在O(n)到O(n^2）之间，数据序列的初始排列越接近有序，
//直接插入排序的时间效率越高。
//直接插入排序算法中的temp占用一个存储单元，空间复杂度为O(1)
//稳定
void InsertSort()
{
    cout << "InsertSort: " << endl;
    //目标数组
    int array_i[20] = { 3, 29, 10, 29, 34, 2, 8, 4, 19, 7, 31, 2, 54, 62, 12, 23,
                        43, 60, 26, 13 };
    int size = sizeof(array_i) / sizeof(int);
    //排序
    int cur = 1;
    //从第二个开始向第一个序列中添加元素
    for (; cur < size; cur++)
    {
        if ( array_i[cur] < array_i[cur - 1] )
        {
            int tem_cur = array_i[cur];
            int j = cur - 1;
            //找到比当前元素小的元素下标，并将前面的进行后移
            for (; j >= 0 && tem_cur<array_i[j]; j--)
            {
                array_i[j+1] = array_i[j];
            }
            array_i[j+1] = tem_cur;
        }
    }
    OutputArray(array_i, size);
};
//Shell排序
//将原数组按照一定增量分割成若干个子序列，进行插入排序；不断缩小增量，同时对子序
//列进行插入排序，直到增量为1
//在最优的情况下，时间复杂度为：O（n^(1.3)）（元素已经排序好顺序）
//在最差的情况下，时间复杂度为：O（n^2）
//空间复杂度O(1)
//不稳定
void ShellSort()
{
    cout << "ShellSort: " << endl;
    //目标数组
    int array_i[20] = { 3, 29, 10, 29, 34, 2, 8, 4, 19, 7, 31, 2, 54, 62, 12, 23,
                        43, 60, 26, 13 };
    int size = sizeof(array_i) / sizeof(int);
    //增量初值为数组的1/2，然后不断减半减半
    for (int gap = size / 2; gap > 0; gap = gap / 2)
    {
        //对于当前增量下分割出的每一个数组进行插入排序
        for (int m = 0; m < gap; m++)
        {
            //排序
            int cur = 1;
            //从第二个开始向第一个序列中添加元素
            for (; cur < size; cur+=gap)
            {
                if (array_i[cur] < array_i[cur - gap])
                {
                    int tem_cur = array_i[cur];
                    int j = cur - gap;
                    //找到比当前元素小的元素下标，并将前面的进行后移
                    for (; j >= 0 && tem_cur<array_i[j]; j-=gap)
                        array_i[j + gap] = array_i[j];
                    array_i[j + gap] = tem_cur;
                }
            }
        }
    }
    OutputArray(array_i,size);
};
```

```
74
75  //冒泡排序
76  //每次比较相邻两个元素的大小，如果不符合排序要求，则交换两个数据，每一轮比较会冒
77  //泡出一个最值，用于比较的数组就少一个，直到比较数列只剩一个为止
78  //时间复杂度O（n^2）
79  //空间复杂度O(1)
80  //稳定
81  void BubbleSort()
82  {
83      cout << "BubbleSort: " << endl;
84      //目标数组
85      int array_i[20] = { 3, 29, 10, 29, 34, 2, 8, 4, 19, 7, 31, 2, 54, 62, 12, 23,
86                          43, 60, 26, 13 };
87      int size = sizeof(array_i) / sizeof(int);
88
89      //排序
90      for (int i = 0; i < size - 1; i++)
91      {
92          for (int j = 0; j < size - i-1; j++)
93          {
94              if (array_i[j]>array_i[j + 1])
95              {
96                  int tem = array_i[j];
97                  array_i[j] = array_i[j + 1];
98                  array_i[j + 1] = tem;
99              }
100         }
101     }
102     OutputArray(array_i,size);
103 }
104 //快速排序
105 //选择一个元素（首元素）作为枢轴，经过一轮排序，比它大的元素放到它的后面，比它小
106 //的元素放到它的前面，以此将数组分为两段，重新对得到的两个数组进行上述操作，直到
107 //得到的数组只剩一个。
108 //时间复杂度O(n^2)
109 //空间复杂度O(n)
110 //不稳定
111 int QuickPartition(int arr[],int low ,int high)
112 {
113     //分段排序，并返回枢轴位置
114     //保存轴枢元素
115     int arr_low = arr[low];
116     //循环两头的指针
117     while (low < high)
118     {
119
        //首先从右往左查找，找到小于枢轴的数，并将其放到low地址位，原low地址位的数据已
        经保存，不会被覆盖
120         while (low < high&&arr_low <= arr[high])
121             high--;
122         if (low < high)
123             arr[low]=arr[high];
124
        //然后从左往右查找，找到大于枢轴的数，并将其放到high地址位，high地址位的数据曾
        放到过前low地址位，所以不会被覆盖
125         while (low<high&&arr_low >= arr[low])
126             low++;
127         if (low < high)
128         arr[high] = arr[low];
129     }
130     arr[low] = arr_low;
131     return low;
132 }
133 void QuickMain(int array_int[],int low,int high)
134 {
135     //递归调用，直到数组只有一个元素
136     if (low < high)
137     {
138         int half = QuickPartition(array_int,low,high);
139         QuickMain(array_int,low,half-1);
140         QuickMain(array_int,half+1,high);
141     }
142 };
```

```
143   void QuickSort()
144   {
145       cout << "QuickSort: " << endl;
146       //目标数组
147       int array_i[20] = { 3, 29, 10, 29, 34, 2, 8, 4, 19, 7, 31, 2, 54, 62, 12, 23,
148                           43, 60, 26, 13 };
149       int size = sizeof(array_i) / sizeof(int);
150
151       QuickMain(array_i,0,size-1);
152       OutputArray(array_i,size);
153   };
154
155   //简单选择排序
156   //在（i，n-1）个元素中，寻找出最小的元素，与i元素交换，直到i==n-1
157   //时间复杂度O(n^2)
158   //空间复杂度O(1)
159   void SelectSort()
160   {
161       cout << "SelectSort: " << endl;
162       //目标数组
163       int array_i[20] = { 3, 29, 10, 29, 34, 2, 8, 4, 19, 7, 31, 2, 54, 62, 12, 23,
164                           43, 60, 26, 13 };
165       int size = sizeof(array_i) / sizeof(int);
166
167       for (int i = 0; i < size ; i++)
168       {
169           int tem = i;
170           for (int j = i + 1; j < size ; j++)
171           {
172               if (array_i[j] < array_i[tem])
173                   tem = j;
174           }
175           if (tem != i)
176           {
177               int tem_arry = array_i[tem];
178               array_i[tem] = array_i[i];
179               array_i[i] = tem_arry;
180           }
181       }
182
183       OutputArray(array_i,size);
184   }
185
186   //堆排序
187   //将数组的所有n个元素构建成一个小顶堆，输出堆顶元素，将剩下的n-1个元素重新构成小
188   //顶堆，直到只剩一个元素重建堆的时间复杂度O(n)
189   //新建堆O(nlogn)
190   //时间复杂度O(nlogn)
191   //空间复杂度O(1)
192   void BuildHeap(int arr[],int start,int size)
193   {
194       for (int pnode = start, i = 2 * start + 1; i <= size - 1;)
195       {
196           //找到两个子节点中最大的一个
197           if (i < size - 1 && arr[i] < arr[i + 1])
198               i++;
199           //如果当前节点大于最大的子节点，那么无需调整
200           if (arr[pnode] >= arr[i])
201               break;
202           //否则，交换当前节点和最大子节点
203           int tem = arr[i];
204           arr[i] = arr[pnode];
205           arr[pnode] = tem;
206           //由于与子节点发生交换，可能破坏原先的子节点的大小关系，所以需要重新调整子节点
207           pnode = i;
208           i = 2 * i + 1;
209       }
210   }
211
212
213
214
215
```

```
216 void HeapSort()
217 {
218     cout << "HeapSort: " << endl;
219     //目标数组
220     int array_i[20] = { 3, 29, 10, 29, 34, 2, 8, 4, 19, 7, 31, 2, 54, 62, 12, 23,
221                         43, 60, 26, 13 };
222     int size = sizeof(array_i) / sizeof(int);
223     //初始化顶堆
224     //从编号最大的节点开始构建，直到根节点
225     for (int i = size / 2; i >= 0; i--)
226         BuildHeap(array_i,i,size);
227     //将数组第一个元素（最大值）与末尾元素交换，重新构建除末尾元素的大顶堆
228     for (int i = size-1; i >=0; i--)
229     {
230         int tem = array_i[0];
231         array_i[0] = array_i[i];
232         array_i[i] = tem;
233         BuildHeap(array_i,0,i);
234     }
235     OutputArray(array_i,size);
236 }
237
238 //归并排序
239 //分治思想，首先把数组n分成长度为1的n个数组，接着两两合并并排好顺序，再两两合并，
240 //重复上述操作，直到合并成原数组
241 void Merge(int arr[], int left,int mid, int right)
242 {
243     //将分开后的两个数组进行合并
244     int t_left = left;
245     int t_mid = mid+1;
246     int t = 0;
247     int *temp = new int[right - left+1];
248     //按顺序将元素放到临时数组中
249     while (t_left <= mid&& t_mid <= right)
250     {
251         if (arr[t_left] <= arr[t_mid])
252             temp[t++] = arr[t_left++];
253         else
254             temp[t++] = arr[t_mid++];
255     }
256     while (t_left <= mid)
257         temp[t++] = arr[t_left++];
258     while (t_mid <= right)
259         temp[t++] = arr[t_mid++];
260     t = 0;
261     while (left<=right)
262         arr[left++] = temp[t++] ;
263     delete []temp;
264
265 }
266 void MergeMain(int arr[],int left ,int right)
267 {
268     //主函数
269     if (left < right)
270     {
271         int mid = (left + right) / 2;
272         MergeMain(arr, left, mid);
273         MergeMain(arr,mid+1,right);
274         Merge(arr,left,mid,right);
275     }
276
277 }
278 void MergeSort()
279 {
280     cout << "MergeSort: " << endl;
281     //目标数组
282     int array_i[20] = { 3, 29, 10, 29, 34, 2, 8, 4, 19, 7, 31, 2, 54, 62, 12, 23,
283                         43, 60, 26, 13 };
284     int size = sizeof(array_i) / sizeof(int);
285
286     MergeMain(array_i,0,size-1);
287     OutputArray(array_i,size);
288 }
```