常见排序算法的实现(一)-插入排序

插入排序是最简单最直观的排序算法了，它的依据是：遍历到第 N 个元素的时候前面的N-1个元素已经是排序好的了，那么就查找前面的 N-1个元素把这第 N 个元素放在合适的位置，如此下去直到遍历完序列的元素为止。

算法的复杂度也是简单的，排序第一个需要1的复杂度，排序第二个需要2的复杂度，因此整个的复杂度就是

1 + 2 + 3 + …… + N = O（N ^ 2）的复杂度。

```
 // 插入排序
void InsertSort(int array[], int length)
{
	int i, j, key;

	for (i = 1; i < length; i++)
	{
		key = array[i];
		// 把 i 之前大于 array[i]的数据向后移动
		for (j = i - 1; j >= 0 && array[j] > key; j--)
		{
			array[j + 1] = array[j];
		}
		// 在合适位置安放当前元素
		array[j + 1] = key;
	}
}
```

常见排序算法的实现(二)-shell 排序

shell 排序是对插入排序的一个改装，它每次排序把序列的元素按照某个增量分成几个子序列，对这几个子序列进行插入排序，然后不断的缩小增量扩大每个子序列的元素数量，直到增量为一的时候子序列就和原先的待排列序列一样了，此时只需要做少量的比较和移动就可以完成对序列的排序了。

```
// shell 排序
void ShellSort(int array[], int length)
{
	int temp;

	// 增量从数组长度的一半开始,每次减小一倍
	for (int increment = length / 2; increment > 0; increment /= 2)
		for (int i = increment; i < length; ++i)
		{
			temp = array[i];
			// 对一组增量为 increment 的元素进行插入排序
			for (int j = i; j >= increment; j -= increment)
			{
				// 把 i 之前大于 array[i]的数据向后移动
				if (temp < array[j - increment])
				{
					array[j] = array[j - increment];
				}
				else
				{
					break;
				}
			}
			// 在合适位置安放当前元素
			array[j] = temp;
		}
}
```

常见排序算法的实现(三)-堆排序

堆的定义：

n 个关键字序列 Kl，K2，…，Kn 称为堆，当且仅当该序列满足如下性质（简称为堆性质）：

（1） ki≤K2i 且 ki≤K2i+1 或（2）Ki≥K2i 且 ki≥K2i+1（1≤i≤）

若将此序列所存储的向量 R[1……n]看做是一棵完全二叉树的存储结构，则堆实质上是满足如下性质的完全二叉树：树中任一非叶结点的关键字均不大于（或不小于）其左右孩子（若存在）结点的关键字。

堆的这个性质使得可以迅速定位在一个序列之中的最小（大）的元素。

堆排序算法的过程如下：1）得到当前序列的最小（大）的元素 2）把这个元素和最后一个元素进行交换，这样当前的最小（大）的元素就放在了序列的最后，而原先的最后一个元素放到了序列的最前面 3）的交换可能会破坏堆序列的性质（注意此时的序列是除去已经放在最后面的元素），因此需要对序列进行调整，使之满足于上面堆的性质。重复上面的过程，直到序列调整完毕为止。

```
 // array 是待调整的堆数组,i 是待调整的数组元素的位置,length 是数组的长度
void HeapAdjust(int array[], int i, int nLength)
{
	int nChild, nTemp;

	for (nTemp = array[i]; 2 * i + 1 < nLength; i = nChild)
	{
		// 子结点的位置是 父结点位置 * 2 + 1
		nChild = 2 * i + 1;

		// 得到子结点中较大的结点
		if (nChild != nLength - 1 && array[nChild + 1] > array[nChild])
			++nChild;

		// 如果较大的子结点大于父结点那么把较大的子结点往上移动,替换它的父结点
		if (nTemp < array[nChild])
		{
			array[i] = array[nChild];
```

```
                }
                else        // 否则退出循环
                {
                        break;
                }
        }

        // 最后把需要调整的元素值放到合适的位置
        array[i] = nTemp;
}

// 堆排序算法
void HeapSort(int array[], int length)
{
        // 调整序列的前半部分元素,调整完之后第一个元素是序列的最大的元素
        for (int i = length / 2 - 1; i >= 0; --i)

0// for (int i = length / 2; i >= 1; i--)
        {
                HeapAdjust(array, i, length);
        }

        // 从最后一个元素开始对序列进行调整,不断的缩小调整的范围直到第一个元素
        for (int i = length - 1; i > 0; --i)
        {
                // 把第一个元素和当前的最后一个元素交换,
                // 保证当前的最后一个位置的元素都是在现在的这个序列之中最大的
                Swap(&array[0], &array[i]);

                // 不断缩小调整 heap 的范围,每一次调整完毕保证第一个元素是当前序列的最大值
                HeapAdjust(array, 0, i);
        }
}
```

一个测试及输出的结果，在每次 HeapAdjust 之后显示出来当前数组的情况

```
 before Heap sort:
71 18 151 138 160 63 174 169 79 78
// 开始调整前半段数组元素
71 18 151 138 160 63 174 169 79 78
71 18 151 169 160 63 174 138 79 78
71 18 174 169 160 63 151 138 79 78
71 169 174 138 160 63 151 18 79 78
174 169 151 138 160 63 71 18 79 78
// 开始进行全局的调整
169 160 151 138 78 63 71 18 79 174
160 138 151 79 78 63 71 18 169 174
151 138 71 79 78 63 18 160 169 174
138 79 71 18 78 63 151 160 169 174
```

```
79 78 71 18 63 138 151 160 169 174
78 63 71 18 79 138 151 160 169 174
71 63 18 78 79 138 151 160 169 174
63 18 71 78 79 138 151 160 169 174
18 63 71 78 79 138 151 160 169 174
```

常见排序算法的实现(四)-冒泡排序

冒泡排序算法的思想：很简单，每次遍历完序列都把最大（小）的元素放在最前面，然后再对剩下的序列从父前面的一个过程，每次遍历完之后待排序序列就少一个元素，当待排序序列减小为只有一个元素的时候排序就结束了。因此，复杂度在最坏的情况下是 O（N ^ 2）。

```
void    Swap( int      * a,    int      * b)
  {
        int    temp;

      temp    =      * a;
        * a        =      * b;
        * b        =    temp;
}

 //    冒泡排序
 void    BubbleSort( int    array[],    int    length)
  {
        //    记录一次遍历中是否有元素的交换
          bool    exchange;
        for    ( int    i    =      0 ; i    <    length;    ++ i)
          {
              exchange    =      false ;
                for    ( int    j    =    i    +      1 ; j    <    length;    ++ j)
                  {
                        if    (array[j]    <    array[i])
                          {
                              exchange    =      true ;
                              Swap( & array[j],    & array[i]);
                        }
              }
```

```
                //    如果这次遍历没有元素的交换,那么排序结束
                  if    ( false       ==     exchange)
                        break ;
        }
}
```

常见排序算法的实现(五)-快速排序

快速排序的算法思想： 选定一个枢纽元素，对待排序序列进行分割，分割之后的序列一个部分小于枢纽元素，一个部分大于枢纽元素，再对这两个分割好的子序列进行上述的过程。

```
 // 对一个给定范围的子序列选定一个枢纽元素,执行完函数之后返回分割元素所在的位置,
// 在分割元素之前的元素都小于枢纽元素,在它后面的元素都大于这个元素
int Partition(int array[], int low, int high)
{
        // 采用子序列的第一个元素为枢纽元素
        int pivot = array[low];

        while (low < high)
        {
                // 从后往前在后半部分中寻找第一个小于枢纽元素的元素
                while (low < high && array[high] >= pivot)
                {
                        --high;
                }

                // 将这个比枢纽元素小的元素交换到前半部分
                Swap(&array[low], &array[high]);

                // 从前往后在前半部分中寻找第一个大于枢纽元素的元素
                while (low < high && array[low] <= pivot)
                {
                        ++low;
                }

                // 将这个比枢纽元素大的元素交换到后半部分
                Swap(&array[low], &array[high]);
        }

        // 返回枢纽元素所在的位置
```

```
        return low;
}

// 快速排序
void QuickSort(int array[], int low, int high)
{
        if (low < high)
        {
                int n = Partition(array, low, high);
                QuickSort(array, low, n);
                QuickSort(array, n + 1, high);
        }
}
```

常见排序算法的实现(六)-归并排序

归并排序的算法思想：把待排序序列分成相同大小的两个部分，依次对这两部分进行归并排序，完毕之后再按照顺序进行合并。

```
 // 归并排序中的合并算法
void Merge(int array[], int start, int mid, int end)
{
        int temp1[10], temp2[10];
        int n1, n2;
        n1 = mid - start + 1;
        n2 = end - mid;

        // 拷贝前半部分数组
        for (int i = 0; i < n1; i++)
        {
                temp1[i] = array[start + i];
        }
        // 拷贝后半部分数组
        for (int i = 0; i < n2; i++)
        {
                temp2[i] = array[mid + i + 1];
        }
        // 把后面的元素设置的很大
        temp1[n1] = temp2[n2] = 1000;
        // 逐个扫描两部分数组然后放到相应的位置去
        for (int k = start, i = 0, j = 0; k <= end; k++)
        {
                if (temp1[i] <= temp2[j])
```

```
                {
                        array[k] = temp1[i];
                        i++;
                }
                else
                {
                        array[k] = temp2[j];
                        j++;
                }
        }
}
// 归并排序
void MergeSort(int array[], int start, int end)
{
        if (start < end)
        {
                int i;
                i = (end + start) / 2;
                // 对前半部分进行排序
                MergeSort(array, start, i);
                // 对后半部分进行排序
                MergeSort(array, i + 1, end);
                // 合并前后两部分
                Merge(array, start, i, end);
        }
}
```