必须知道的八大种排序算法【java实现】

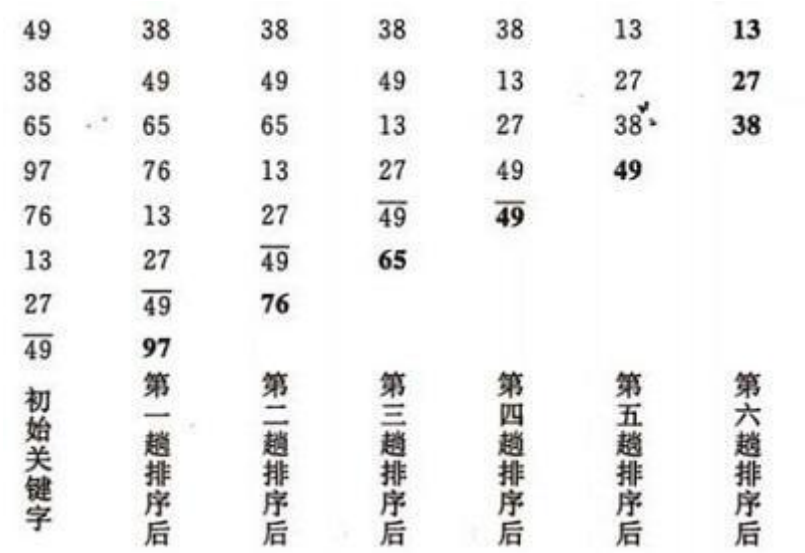
shadow000902

22016.02.03 18:32:25字数 1,397阅读 57,326

一、冒泡排序

冒泡排序是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

冒泡排序的**示例**：



冒泡排序的**算法实现**如下:【排序后，数组从小到大排列】

```
/* * 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。在这一点，最后的元素应该会是最大的数。 * 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。 * @param numbers 需要排序的整型数组 */ public static void bubbleSort(int[] numbers) { int temp = 0; int size = numbers.length; for(int i = 0 ; i < size-1; i ++) { for(int j = 0 ;j < size-1-i ; j++) { if(numbers[j] > numbers[j+1]) //交换两数位置 { temp = numbers[j]; numbers[j] = numbers[j+1]; numbers[j+1] = temp; } } } }
```

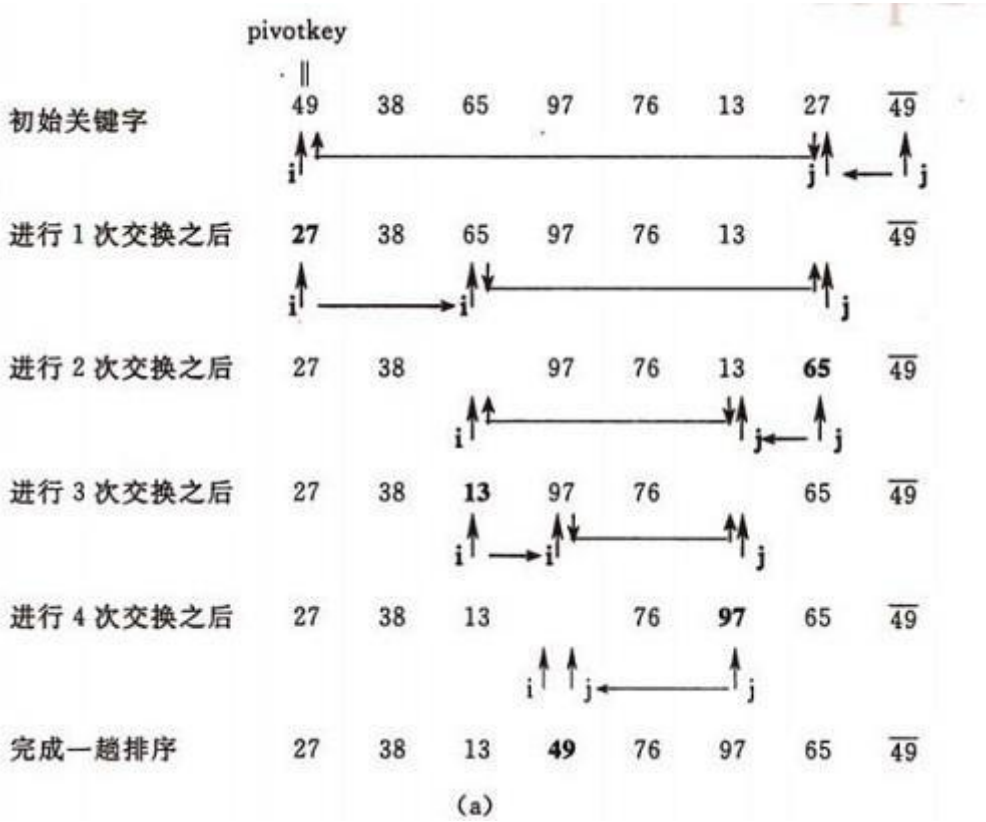
二、快速排序

快速排序的基本思想：

通过一趟排序将待排序记录分割成独立的两部分，其中一部分记录的关键字均比另一部分关键字小，则分别对这两部分继续进行排序，直到整个序列有序。

快速排序的**示例**：

（a）一趟排序的过程：



（b）排序的全过程：

把整个序列看做一个数组，把第零个位置看做中轴，和最后一个比，如果比它小交换，比它大不做任何处理；交换了以后再和小的那端比，比它小不交换，比他大交换。这样循环往复，一趟排序完成，左边就是比中轴小的，右边就是比中轴大的，然后再用分治法，分别对这两个独立的数组进行排序。

代码实现如下：

1.查找中轴（最低位作为中轴）所在位置：

```
/** * 查找出中轴（默认是最低位low）的在numbers数组排序后所在位置 * * @param numbers 带查找数组 * @param low 开始位置 * @param high 结束位置 * @return 中轴所在位置 */ public static int getMiddle(int[] numbers, int low,int high) { int temp = numbers[low]; //数组的第一个作为中轴 while(low < high) { while(low < high && numbers[high] >= temp) { high--; } numbers[low] = numbers[high];//比中轴小的记录移到低端 while(low < high && numbers[low] < temp) { low++; } numbers[high] = numbers[low] ; //比中轴大的记录移到高端 } numbers[low] = temp ; //中轴记录到尾 return low ; // 返回中轴的位置 }
```

2、 递归形式的分治排序算法：

```
/** * * @param numbers 带排序数组 * @param low 开始位置 * @param high 结束位置 */ public static void quickSort(int[] numbers,int low,int high) { if(low < high) {     int middle = getMiddle(numbers,low,high); //将numbers数组进行一分为二     quickSort(numbers, low, middle-1); //对低字段表进行递归排序     quickSort(numbers, middle+1, high); //对高字段表进行递归排序 } }
```

3、快速排序提供方法调用：

```
/** * 快速排序 * @param numbers 带排序数组 */ public static void quick(int[] numbers) { if(numbers.length > 0) //查看数组是否为空 { quickSort(numbers, 0, numbers.length-1); } }
```

分析：

快速排序是通常被认为在**同数量级（O(nlog2n)）**的排序方法中平均性能最好的。但若初始序列按关键码有序或基本有序时，快排序反而蜕化为冒泡排序。为改进之，通常以“三者取中法”来选取基准记录，即将排序区间的两个端点与中点三个记录关键码居中的调整为支点记录。快速排序是一个不稳定的排序方法。

三、方法测试

打印函数：

```
public static void printArr(int[] numbers) { for(int i = 0 ; i < numbers.length ; i ++ ) { System.out.print(numbers[i] + ","); } System.out.println(""); }
```

测试：

```
public static void main(String[] args) { int[] numbers = {10,20,15,0,6,7,2,1,-5,55}; System.out.print("排序前："); printArr(numbers); bubbleSort(numbers); System.out.print("冒泡排序后："); printArr(numbers); quick(numbers); System.out.print("快速排序后："); printArr(numbers); }
```

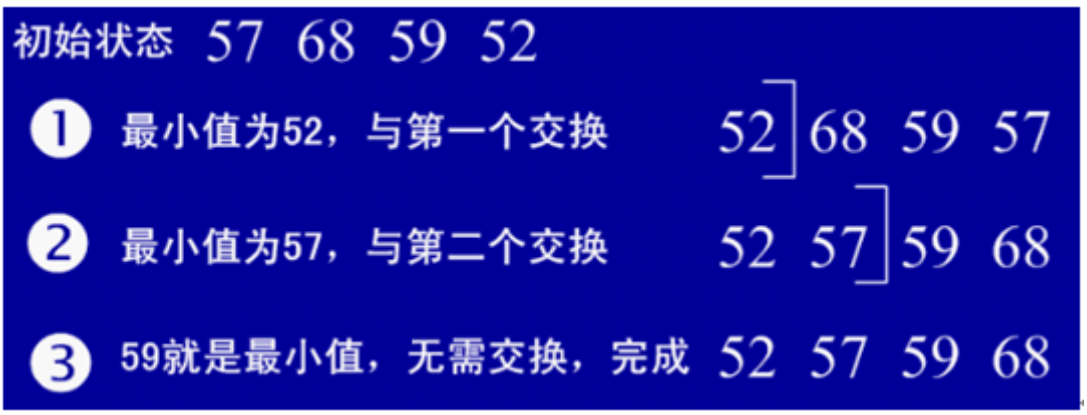
结果：

```
排序前：10,20,15,0,6,7,2,1,-5,55, 冒泡排序后：-5,0,1,2,6,7,10,15,20,55, 快速排序后：-5,0,1,2,6,7,10,15,20,55,
```

四、选择排序

1、基本思想：在要排序的一组数中，选出最小的一个数与第一个位置的数交换；然后在剩下的数当中再找最小的与第二个位置的数交换，如此循环到倒数第二个数和最后一个数比较为止。

2、实例：



3、算法实现：

```
/** * 选择排序算法 * 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，然后放到排序序列末尾。 * 以此类推，直到所有元素均排序完毕。 * @param numbers */ public static void selectSort(int[] numbers) { int size = numbers.length; //数组长度 int temp = 0 ; //中间变量 for(int i = 0 ; i < size ; i++) { int k = i; //待确定的位置 //选择出应该在第i个位置的数 for(int j = size -1 ; j > i ; j--) { if(numbers[j] < numbers[k]) { k = j; } } //交换两个数 temp = numbers[i]; numbers[i] = numbers[k]; numbers[k] = temp; } }
```

五、插入排序

1、基本思想：每步将一个待排序的记录，按其顺序码大小插入到前面已经排序的字序列的合适位置（从后向前找到合适位置后），直到全部插入排序完为止。

2、实例：

3、算法实现：

```
/** * 插入排序 * * 从第一个元素开始，该元素可以认为已经被排序 * 取出下一个元素，在已经排序的元素序列中从后向前扫描 * 如果该元素（已排序）大于新元素，将该元素移到下一位置 * 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置 * 将新元素插入到该位置中 * 重复步骤2 * @param numbers 待排序数组 */ public static void insertSort(int[] numbers) { int size = numbers.length; int temp = 0 ; int j = 0; for(int i = 0 ; i < size ; i++) { temp = numbers[i]; //假如temp比前面的值小，则将前面的值后移 for(j = i ; j > 0 && temp < numbers[j-1] ; j --) { numbers[j] = numbers[j-1]; } numbers[j] = temp; } }
```

4、效率：

时间复杂度：O（n^2）.

六、希尔算法

1、基本思想：

先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行依次直接插入排序。

2、操作方法：

<code>

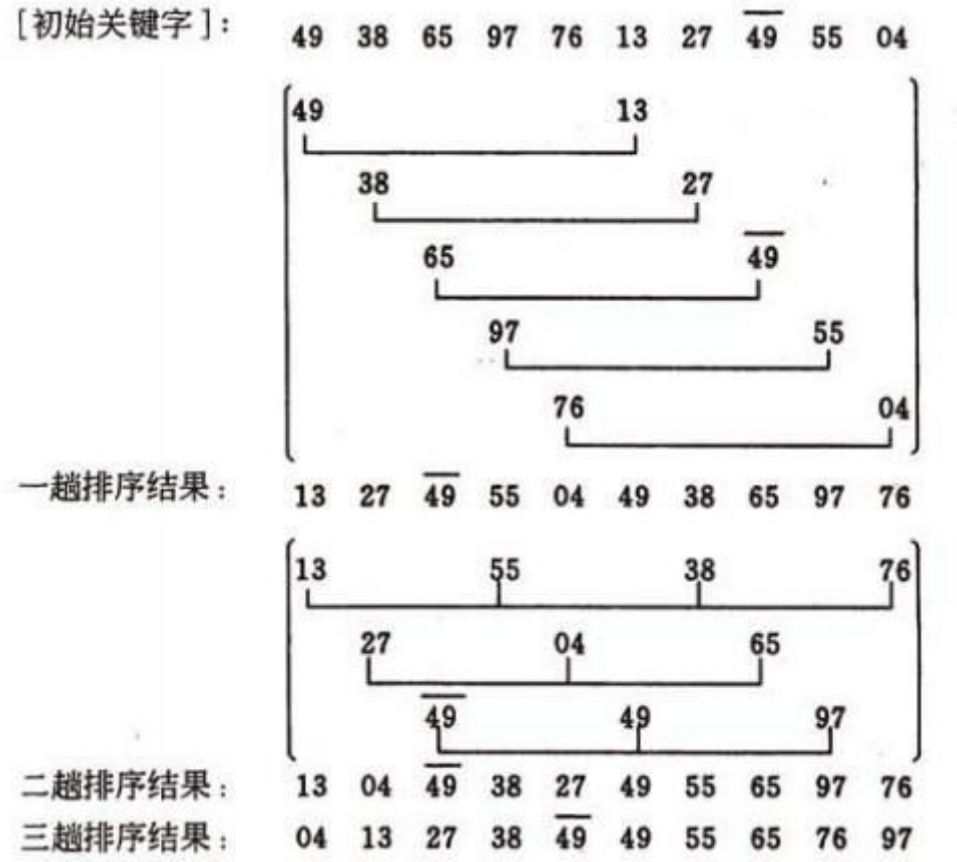
1、选择一个增量序列t1，t2，…，tk，其中ti>tj，tk=1；

2、按增量序列个数k，对序列进行k 趟排序；

3、每趟排序，根据对应的增量ti，将待排序列分割成若干长度为m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

</code>

希尔排序的示例：



3、**算法实现：**

/**希尔排序的原理:根据需求，如果你想要结果从大到小排列，它会首先将数组进行分组，然后将较大值移到前面，较小值 * 移到后面，最后将整个数组进行插入排序，这样比起一开始就用插入排序减少了数据交换和移动的次数，可以说希尔排序是加强 * 版的插入排序 * 拿数组5, 2, 8, 9, 1, 3, 4来说，数组长度为7，当increment为3时，数组分为两个序列 * 5，2，8和9，1，3，4，第一次排序，9和5比较，1和2比较，3和8比较，4和比其下标值小increment的数组值相比较 * 此例子是按照从大到小排列，所以大的会排在前面，第一次排序后数组为9, 2, 8, 5, 1, 3, 4 * 第一次后increment的值变为3/2=1,此时对数组进行插入排序， *实现数组从大到小排 */ public static void shellSort(int[] data) { int j = 0; int temp = 0; //每次将步长缩短为原来的一半 for (int increment = data.length / 2; increment > 0; increment /= 2) { for (int i = increment; i < data.length; i++) { temp = data[i]; for (j = i; j >= increment; j -= increment) { if(temp > data[j - increment])//如想从小到大排只需修改这里 { data[j] = data[j - increment]; } else { break; } } data[j] = temp; } } }

4、效率：

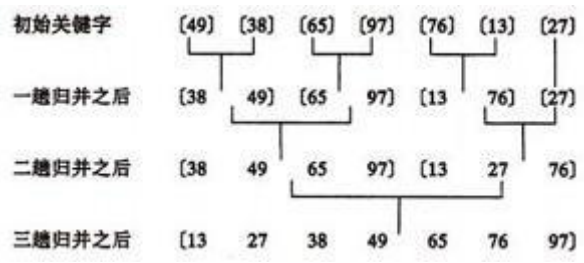
时间复杂度：O（n^2）.

七、 归并排序算法

基本思想：

归并（Merge）排序法是将两个（或两个以上）有序表合并成一个新的有序表，即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。

归并排序示例：



合并方法：

设r[i...n]由两个有序子表r[i...m]和r[m+1...n]组成，两个子表长度分别为n-i +1、n-m。

1、j=m+1；k=i；i=i；//置两个子表的起始下标及辅助数组的起始下标 2、若i>m 或j>n，转⑷ //其中一个子表已合并完，比较选取结束 3、//选取r[i]和r[j]较小的存入辅助数组rf 如果r[i]<r[j]，rf[k]=r[i]； i++； k++； 转⑵ 否则，rf[k]=r[j]； j++； k++； 转⑵ 4、//将尚未处理完的子表中元素存入rf 如果i<=m，将r[i…m]存入rf[k…n] //前一子表非空 如果j<=n ，将r[j…n] 存入rf[k…n] //后一子表非空 5、合并结束。

算法实现：

```
/** * 归并排序 * 简介:将两个（或两个以上）有序表合并成一个新的有序表 即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列 * 时间复杂度为O(nlogn) * 稳定排序方式 * @param nums 待排序数组 * @return 输出有序数组 */ public static int[] sort(int[] nums, int low, int high) { int mid = (low + high) / 2; if (low < high) { // 左边 sort(nums, low, mid); // 右边 sort(nums, mid + 1, high); // 左右归并 merge(nums, low, mid, high); } return nums; } /** * 将数组中low到high位置的数进行排序 * @param nums 待排序数组 * @param low 待排的开始位置 * @param mid 待排中间位置 * @param high 待排结束位置 */ public static void merge(int[] nums, int low, int mid, int high) { int[] temp = new int[high - low + 1]; int i = low;// 左指针 int j = mid + 1;// 右指针 int k = 0; // 把较小的数先移到新数组中 while (i <= mid && j <= high) { if (nums[i] < nums[j]) { temp[k++] = nums[i++]; } else { temp[k++] = nums[j++]; } } // 把左边剩余的数移入数组 while (i <= mid) { temp[k++] = nums[i++]; } // 把右边边剩余的数移入数组 while (j <= high) { temp[k++] = nums[j++]; } // 把新数组中的数覆盖nums数组 for (int k2 = 0; k2 < temp.length; k2++) { nums[k2 + low] = temp[k2]; } }
```

八、堆排序算法

1、基本思想：

堆排序是一种树形选择排序，是对直接选择排序的有效改进。

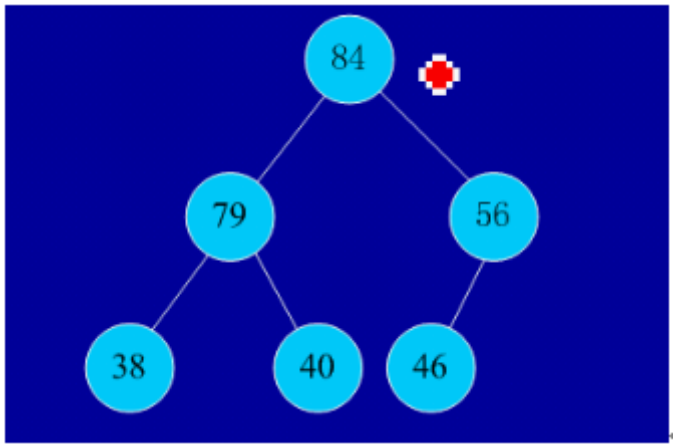
堆的定义下：具有n个元素的序列 （h1,h2,...,hn),当且仅当满足（hi>=h2i,hi>=2i+1）或（hi<=h2i,hi<=2i+1） (i=1,2,...,n/2)时称之为堆。在这里只讨论满足前者条件的堆。由堆的定义可以看出，堆顶元素（即第一个元素）必为最大项（大顶堆）。完全二 叉树可以很直观地表示堆的结构。堆顶为根，其它为左子树、右子树。

思想:初始时把要排序的数的序列看作是一棵顺序存储的二叉树，调整它们的存储序，使之成为一个 堆，这时堆的根节点的数最大。然后将根节点与堆的最后一个节点交换。然后对前面(n-1)个数重新调整使之成为堆。依此类推，直到只有两个节点的堆，并对 它们作交换，最后得到有n个节点的有序序列。从算法描述来看，堆排序需要两个过程，一是建立堆，二是堆顶与堆的最后一个元素交换位置。所以堆排序有两个函数组成。一是建堆的渗透函数，二是反复调用渗透函数实现排序的函数。

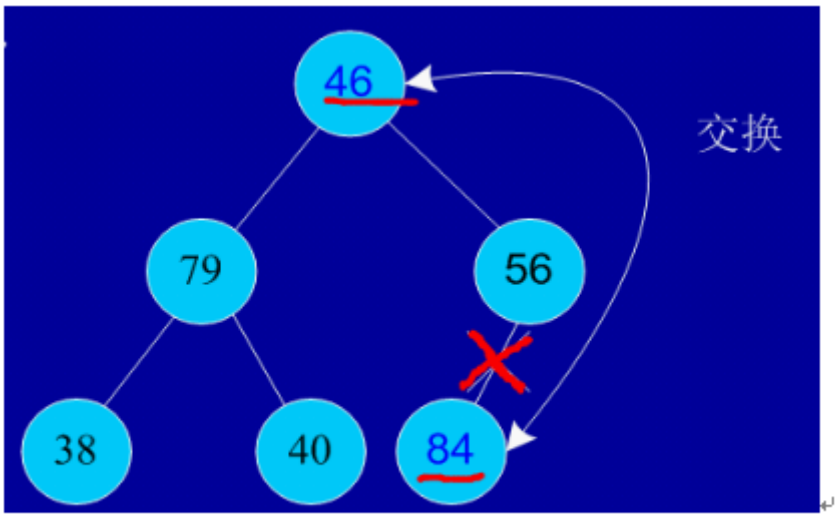
2、实例：

初始序列：46,79,56,38,40,84

建堆：



交换，从堆中踢出最大数：



剩余结点再建堆，再交换踢出最大数

依次类推：最后堆中剩余的最后两个结点交换，踢出一个，排序完成。

3.算法实现：

```
public class HeapSort { public static void main(String[] args) { int[] a= {49,38,65,97,76,13,27,49,78,34,12,64}; int arrayLength=a.length; //循环建堆 for(int i=0;i<arrayLength-1;i++){ //建堆 buildMaxHeap(a,arrayLength-1-i); //交换堆顶和最后一个元素 swap(a,0,arrayLength-1-i); System.out.println(Arrays.toString(a)); } } //对data数组从0到lastIndex建大顶堆 public static void buildMaxHeap(int[] data, int lastIndex){ //从lastIndex处节点（最后一个节点）的父节点开始 for(int i=(lastIndex-1)/2;i>=0;i--){ //k保存正在判断的节点 int k=i; //如果当前k节点的子节点存在 while(k*2+1<=lastIndex){ //k节点的左子节点的索引 int biggerIndex=2*k+1; //如果biggerIndex小于lastIndex，即biggerIndex+1代表的k节点的右子节点存在 if(biggerIndex<lastIndex){ //若果右子节点的值较大 if(data[biggerIndex]<data[biggerIndex+1]){ //biggerIndex总是记录较大子节点的索引 biggerIndex++; } } //如果k节点的值小于其较大的子节点的值 if(data[k]<data[biggerIndex]){ //交换他们 swap(data,k,biggerIndex); //将biggerIndex赋予k，开始while循环的下一次循环，重新保证k节点的值大于其左右子节点的值 k=biggerIndex; }else{ break; } } } } //交换 private static void swap(int[] data, int i, int j) { int tmp=data[i]; data[i]=data[j]; data[j]=tmp; } }
```

九、各种算法的时间复杂度

各种常用排序算法						
类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	不稳定
归并排序		$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定
注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数						