常用排序算法分析与实现（Java版）

作者: junJZ_2008 http://jiangzhengjun.javaeye.com

排序算法分析与实现

目 录

1.1 常用排序算法分析与实现（一）（Java版）

发表时间: 2009-12-13

这篇排序文章从思想 理解 到实现，然后到整理，花了我几天的时间，现把它记录于此，希望对大家有一定的帮助，写的不好的请不要见笑，写错了的，请指出来我更正。最后如果对你有一定的帮助，请回贴支持一下哦^_^ ！

申明： 排序算法思想来自互联网，代码自己实现，仅供参考。

- **插入排序**

直接插入排序、希尔排序

- **选择排序**

简单选择排序、堆排序

- **交换排序**

冒泡排序、快速排序

- **归并排序**
- **基数排序**

排序基类

```
package sort;

import java.util.Arrays;
import java.util.Comparator;
import java.util.Random;

/**
 * 排序接口，所有的排序算法都要继承该抽象类，并且要求数组中的
 * 元素要具有比较能力，即数组元素已实现了Comparable接口
 *
 * @author jzj
 * @date 2009-12-5
```

```
 *
 * @param <E>
 */
public abstract class Sort<E extends Comparable<E>> {

        public final Comparator<E> DEFAULT_ORDER = new DefaultComparator();
        public final Comparator<E> REVERSE_ORDER = new ReverseComparator();

        /**
         * 排序算法，需实现，对数组中指定的元素进行排序
         * @param array 待排序数组
         * @param from 从哪里
         * @param end 排到哪里
         * @param c
         */
        public abstract void sort(E[] array, int from, int end, Comparator<E> c);

        /**
         * 对数组中指定部分进行排序
         * @param from 从哪里
         * @param len 排到哪里
         * @param array 待排序数组
         * @param c 比较器
         */
        public void sort(int from, int len, E[] array, Comparator<E> c) {
                sort(array, 0, array.length - 1, c);
        }

        /**
         * 对整个数组进行排序，可以使用自己的排序比较器，也可使用该类提供的两个比较器
         * @param array 待排序数组
         * @param c 比较器
         */
        public final void sort(E[] array, Comparator<E> c) {
                sort(0, array.length, array, c);
        }
```

```
/**
 * 对整个数组进行排序，采用默认排序比较器
 * @param array 待排序数组
 */
public final void sort(E[] array) {
        sort(0, array.length, array, this.DEFAULT_ORDER);
}

//默认比较器（一般为升序，但是否真真是升序还得看E是怎样实现Comparable接口的）
private class DefaultComparator implements Comparator<E> {
        public int compare(E o1, E o2) {
                return o1.compareTo(o2);
        }
}

//反序比较器，排序刚好与默认比较器相反
private class ReverseComparator implements Comparator<E> {
        public int compare(E o1, E o2) {
                return o2.compareTo(o1);
        }
}

/**
 * 交换数组中的两个元素的位置
 * @param array 待交换的数组
 * @param i 第一个元素
 * @param j 第二个元素
 */
protected final void swap(E[] array, int i, int j) {
        if (i != j) {//只有不是同一位置时才需交换
                E tmp = array[i];
                array[i] = array[j];
                array[j] = tmp;
        }
}

/**
```

```
     * 数组元素后移
     * @param array 待移动的数组
     * @param startIndex 从哪个开始移
     * @param endIndex 到哪个元素止
     */
    protected final void move(E[] array, int startIndex, int endIndex) {
            for (int i = endIndex; i >= startIndex; i--) {
                    array[i + 1] = array[i];
            }
    }

    /**
     * 以指定的步长将数组元素后移，步长指定每个元素间的间隔
     * @param array 待排序数组
     * @param startIndex 从哪里开始移
     * @param endIndex 到哪个元素止
     * @param step 步长
     */
    protected final void move(E[] array, int startIndex, int endIndex, int step) {
            for (int i = endIndex; i >= startIndex; i -= step) {
                    array[i + step] = array[i];
            }
    }

    //测试方法
    @SuppressWarnings("unchecked")
    public static final <E extends Comparable<E>> void testSort(Sort<E> sorter, E[] array)

            if (array == null) {
                    array = randomArray();
            }
            //为了第二次排序，需拷贝一份
            E[] tmpArr = (E[]) new Comparable[array.length];
            System.arraycopy(array, 0, tmpArr, 0, array.length);

            System.out.println("源 - " + Arrays.toString(tmpArr));
```

```
                sorter.sort(array, sorter.REVERSE_ORDER);
                System.out.println("降 - " + Arrays.toString(array));

                sorter.sort(tmpArr, sorter.DEFAULT_ORDER);
                System.out.println("升 - " + Arrays.toString(tmpArr));
        }

        //生成随机数组
        @SuppressWarnings("unchecked")
        private static <E extends Comparable<E>> E[] randomArray() {
                Random r = new Random(System.currentTimeMillis());
                Integer[] a = new Integer[r.nextInt(30)];
                for (int i = 0; i < a.length; i++) {
                        a[i] = new Integer(r.nextInt(100));
                }
                return (E[]) a;
        }
}
```

插入排序

直接插入排序

一般直接插入排序的时间复杂度为O（n^2），但是当数列基本有序时，如果按照有数列顺序排时，时间复杂度将改善到O(n)，另外，因直接插入排序算法简单，如果待排序列规模不很大时效率也较高。

在已经排好序的序列中查找待插入的元素的插入位置，并将待插入元素插入到有序列表中的过程。

将数组分成两部分，初始化时，前部分数组为只有第一个元素，用来存储已排序元素，我们这里叫 arr1 ；后部分数组的元素为除第一个元素的所有元素，为待排序或待插入元素，我们这里叫 arr2 。
排序时使用二层循环：第一层对 arr2 进行循环，每次取后部分数组（待排序数组）里的第一个元素（我们称为待排序元素或称待插入元素） e1 ，然后在第二层循环中对 arr1 （已排好序的数组）从第一个元素往后进行循环，查到第一个大于待插入元素（如果是升序排列）或第一个小于待插入元素（如果是降序排列） e2 ，然后对 arr1 从 e2 元素开始往后的所有元素向后移，最后把 e1 插入到原来 e2 所在的位置。这样反复地对 arr2 进行

循环，直到 arr2 中所有的待插入的元素都插入到 arr1 中。


```
package sort;

import java.util.Comparator;

/**
 *  直接插入排序算法
 * @author jzj
 * @date 2009-12-5
 *
 * @param <E>
 */
public class InsertSort<E extends Comparable<E>> extends Sort<E> {

        /**
         * 排序算法的实现，对数组中指定的元素进行排序
         * @param array 待排序的数组
         * @param from 从哪里开始排序
         * @param end 排到哪里
         * @param c 比较器
         */
        public void sort(E[] array, int from, int end, Comparator<E> c) {

                /*
                 * 第一层循环：对待插入（排序）的元素进行循环
                 * 从待排序数组断的第二个元素开始循环，到最后一个元素（包括）止
```

```
         */
        for (int i = from + 1; i <= end; i++) {
                /*
                 * 第二层循环：对有序数组进行循环，且从有序数组最第一个元素开始向后循环
                 * 找到第一个大于待插入的元素
                 * 有序数组初始元素只有一个，且为源数组的第一个元素，一个元素数组总是有序的
                 */
                for (int j = 0; j < i; j++) {
                        E insertedElem = array[i];//待插入到有序数组的元素
                        //从有序数组中最一个元素开始查找第一个大于待插入的元素
                        if (c.compare(array[j], insertedElem) > 0) {
                                //找到插入点后，从插入点开始向后所有元素后移一位
                                move(array, j, i - 1);
                                //将待排序元素插入到有序数组中
                                array[j] = insertedElem;
                                break;
                        }
                }
        }

        //=======以下是java.util.Arrays的插入排序算法的实现
        /*
         * 该算法看起来比较简洁一j点，有点像冒泡算法。
         * 将数组逻辑上分成前后两个集合，前面的集合是已经排序好序的元素，而后面集合为待排序的
         * 集合，每次内层循从后面集合中拿出一个元素，通过冒泡的形式，从前面集合最后一个元素开
         * 始往前比较，如果发现前面元素大于后面元素，则交换，否则循环退出
         *
         * 总感觉这种算术有点怪怪，既然是插入排序，应该是先找到插入点，而后再将待排序的元素插
         * 入到的插入点上，那么其他元素就必然向后移，感觉算法与排序名称不匹，但返过来与上面实
         * 现比，其实是一样的，只是上面先找插入点，待找到后一次性将大的元素向后移，而该算法却
         * 是走一步看一步，一步一步将待排序元素往前移
         */
        /*
        for (int i = from; i <= end; i++) {
                for (int j = i; j > from && c.compare(array[j - 1], array[j]) > 0; j--)
                        swap(array, j, j - 1);
                }
```

```
                }
                */
        }

        /**
         * 测试
         * @param args
         */
        public static void main(String[] args) {
                Integer[] intgArr = { 5, 9, 1, 4, 1, 2, 6, 3, 8, 0, 7 };
                InsertSort<Integer> insertSort = new InsertSort<Integer>();
                Sort.testSort(insertSort, intgArr);
                Sort.testSort(insertSort, null);
        }
}
```

插入排序算法对于大数组，这种算法非常慢。但是对于小数组，它比其他算法快。其他算法因为待的数组元素很少，反而使得效率降低。在Java集合框架中，排序都是借助于java.util.Arrays来完成的，其中排序算法用到了插入排序、快速排序、归并排序。插入排序用于元素个数小于7的子数组排序，通常比插入排序快的其他排序方法，由于它们强大的算法是针对大数量数组设计的，所以元素个数少时速度反而慢。

希尔排序

希尔思想介绍

希尔算法的本质是缩小增量排序，是对直接插入排序算法的改进。一般直接插入排序的时间复杂度为O（n^2），但是当数列基本有序时，如果按照有数列顺序排时，时间复杂度将改善到O(n)，另外,因直接插入排序算法简单,如果待排序列规模不很大时效率也较高，Shell 根据这两点分析结果进行了改进，将待排记录序列以一定的增量间隔h 分割成多个子序列，对每个子序列分别进行一趟直接插入排序, 然后逐步减小分组的步长h ，对于每一个步长h 下的各个子序列进行同样方法的排序,直到步长为1 时再进行一次整体排序。
因为不管记录序列多么庞大,关键字多么混乱,在先前较大的分组步长h 下每个子序列的规模都不大，用直接插入排序效率都较高。 尽管在随后的步长h 递减分组中子序列越来越大，但由于整个序列的有序性也越来越明显，则排序效率依然较高。这种改进抓住了直接插入排序的两点本质，大大提高了它的时间效率。

希尔增量研究

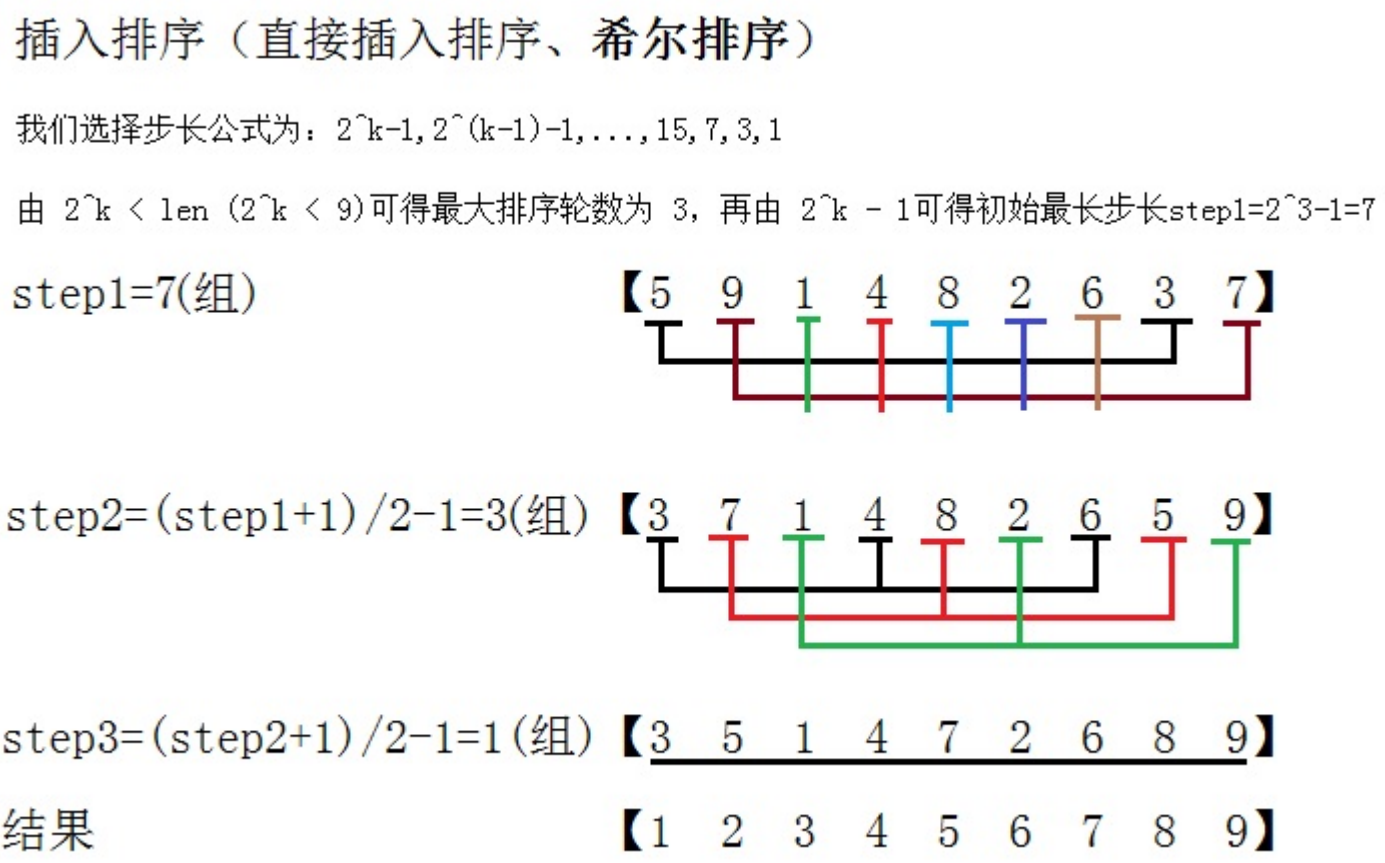
综上所述：

(1) 希尔排序的核心是以某个增量h 为步长跳跃分组进行插入排序，由于分组的步长h 逐步缩小，所以也叫“缩小增量排序”插入排序。其关键是如何选取分组的步长序列ht ,. . . , hk ,. . . , h1 , h0 才能使得希尔方法的时间效率最高；

(2) 待排序列记录的个数n 、跳跃分组步长逐步减小直到为1时所进行的扫描次数T、增量的和、记录关键字比较的次数以及记录移动的次数或各子序列中的反序数等因素都影响希尔算法的时间复杂度：其中记录关键字比较的次数是重要因素，它主要取决于分组步长序列的选择；

(3) 希尔方法是一种不稳定排序算法，因为其排序过程中各趟的步长不同，在第k 遍用hk作为步长排序之后，第k +1 遍排序时可能会遇到多个逆序存在，影响排序的稳定性。

试验结果表明,SHELL 算法的时间复杂度受增量序列的影响明显大于其他因素，选取恰当的增量序列能明显提高排序的时间效率，我们认为第k 趟排序扫描的增量步长为 2^k - 1 ,即增量序列为. . . 2^k - 1 ,. . . ,15 ,7 ,3 ,1时较为理想，但它并不是唯一的最佳增量序列，这与其关联函数目前尚无确定解的理论结果是一致的。




```
package sort;

import java.util.Comparator;
```

```
/**
 * 希尔排序算法
 * @author jzj
 * @date 2009-12-5
 *
 * @param <E>
 */
public class ShelltSort<E extends Comparable<E>> extends Sort<E> {

        /**
         * 排序算法的实现，对数组中指定的元素进行排序
         * @param array 待排序的数组
         * @param from 从哪里开始排序
         * @param end 排到哪里
         * @param c 比较器
         */
        public void sort(E[] array, int from, int end, Comparator<E> c) {
                //初始步长，实质为每轮的分组数
                int step = initialStep(end - from + 1);

                //第一层循环是对排序轮次进行循环。(step + 1) / 2 - 1 为下一轮步长值
                for (; step >= 1; step = (step + 1) / 2 - 1) {
                        //对每轮里的每个分组进行循环
                        for (int groupIndex = 0; groupIndex < step; groupIndex++) {

                                //对每组进行直接插入排序
                                insertSort(array, groupIndex, step, end, c);
                        }
                }
        }

        /**
         * 直接插入排序实现
         * @param array 待排序数组
         * @param groupIndex 对每轮的哪一组进行排序
         * @param step 步长
```

```
     * @param end 整个数组要排哪个元素止
     * @param c 比较器
     */
    private void insertSort(E[] array, int groupIndex, int step, int end, Comparator<E> c)
            int startIndex = groupIndex;//从哪里开始排序
            int endIndex = startIndex;//排到哪里
            /*
             * 排到哪里需要计算得到，从开始排序元素开始，以step步长，可求得下元素是否在数组范围内
             * 如果在数组范围内，则继续循环，直到索引超现数组范围
             */
            while ((endIndex + step) <= end) {
                    endIndex += step;
            }

            // i为每小组里的第二个元素开始
            for (int i = groupIndex + step; i <= end; i += step) {
                    for (int j = groupIndex; j < i; j += step) {
                            E insertedElem = array[i];
                            //从有序数组中最一个元素开始查找第一个大于待插入的元素
                            if (c.compare(array[j], insertedElem) >= 0) {
                                    //找到插入点后，从插入点开始向后所有元素后移一位
                                    move(array, j, i - step, step);
                                    array[j] = insertedElem;
                                    break;
                            }
                    }
            }
    }

    /**
     * 根据数组长度求初始步长
     *
     * 我们选择步长的公式为：2^k-1,2^(k-1)-1,...,15,7,3,1 ，其中2^k 减一即为该步长序列，k
     * 为排序轮次
     *
     * 初始步长：step = 2^k-1
     * 初始步长约束条件：step < len - 1 初始步长的值要小于数组长度还要减一的值（因
```

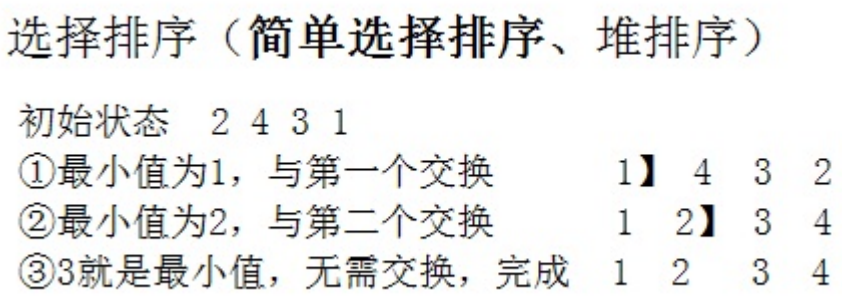
```
     * 为第一轮分组时尽量不要分为一组，除非数组本身的长度就小于等于4）
     *
     * 由上面两个关系试可以得知：2^k - 1 < len - 1 关系式，其中k为轮次，如果把 2^k 表 达式
     * 转换成 step 表达式，则 2^k-1 可使用 (step + 1)*2-1 替换（因为 step+1 相当于第k-1
     * 轮的步长，所以在 step+1 基础上乘以 2 就相当于 2^k 了），即步长与数组长度的关系不等式为
     * (step + 1)*2 - 1 < len -1
     *
     * @param len 数组长度
     * @return
     */
    private static int initialStep(int len) {
            /*
             * 初始值设置为步长公式中的最小步长，从最小步长推导出最长初始步长值，即按照以下公式来推
             * 1,3,7,15,...,2^(k-1)-1,2^k-1
             * 如果数组长度小于等于4时，步长为1，即长度小于等于4的数组不且分组，此时直接退化为直接
             * 入排序
             */
            int step = 1;

            //试探下一个步长是否满足条件，如果满足条件，则步长置为下一步长
            while ((step + 1) * 2 - 1 < len - 1) {
                    step = (step + 1) * 2 - 1;
            }

            System.out.println("初始步长 - " + step);
            return step;
    }

    /**
     * 测试
     * @param args
     */
    public static void main(String[] args) {
            Integer[] intgArr = { 5, 9, 1, 4, 8, 2, 6, 3, 7, 10 };
            ShelltSort<Integer> shellSort = new ShelltSort<Integer>();
            Sort.testSort(shellSort, intgArr);
            Sort.testSort(shellSort, null);
```

```
        }
}
```

选择排序

简单选择排序

每一趟从待排序的数据元素中选出最小（或最大）的一个元素，顺序放在已排好序的数列的最后，直到全部待排序的数据元素排完。
选择排序不像冒泡排序算法那样先并不急于调换位置，第一轮（k=1）先从array[k]开始逐个检查，看哪个数最小就记下该数所在的位置于minlIndex中，等一轮扫描完毕，如果找到比array[k-1]更小的元素，则把array[minlIndex]和a[k-1]对调，这时array[k]到最后一个元素中最小的元素就换到了array[k-1]的位置。 如此反复进行第二轮、第三轮...直到循环至最后一元素




```
package sort;


import java.util.Comparator;


/**
 * 简单选择排序算法
 * @author jzj
 * @date 2009-12-5
 *
 * @param <E>
 */
public class SelectSort<E extends Comparable<E>> extends Sort<E> {


        /**
         * 排序算法的实现，对数组中指定的元素进行排序
         * @param array 待排序的数组
```

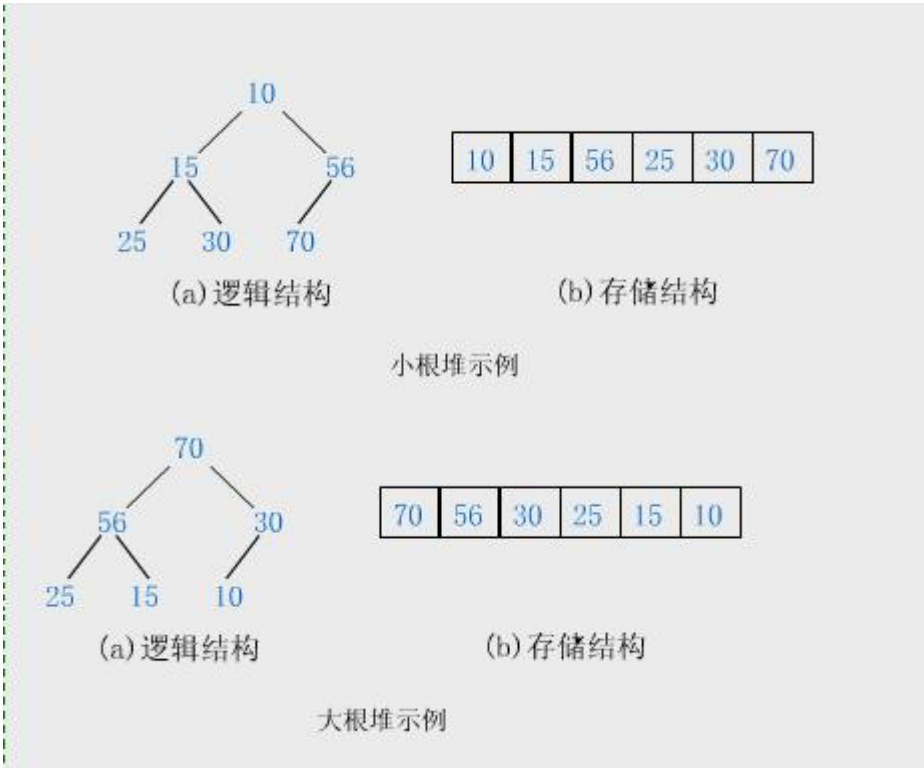
```
     * @param from 从哪里开始排序
     * @param end 排到哪里
     * @param c 比较器
     */
    public void sort(E[] array, int from, int end, Comparator<E> c) {
            int minlIndex;//最小索引
            /*
             * 循环整个数组（其实这里的上界为 array.length - 1 即可，因为当 i= array.length-1
             * 时，最后一个元素就已是最大的了，如果为array.length时，内层循环将不再循环），每轮假
             * 第一个元素为最小元素，如果从第一元素后能选出比第一个元素更小元素，则让让最小元素与第
             * 个元素交换
             */
            for (int i = from; i <= end; i++) {
                    minlIndex = i;//假设每轮第一个元素为最小元素
                    //从假设的最小元素的下一元素开始循环
                    for (int j = i + 1; j <= end; j++) {
                            //如果发现有比当前array[smallIndex]更小元素，则记下该元素的索引于sm
                            if (c.compare(array[j], array[minlIndex]) < 0) {
                                    minlIndex = j;
                            }
                    }

                    //先前只是记录最小元素索引，当最小元素索引确定后，再与每轮的第一个元素交换
                    swap(array, i, minlIndex);
            }
    }

    /**
     * 测试
     * @param args
     */
    public static void main(String[] args) {
            Integer[] intgArr = { 5, 9, 1, 4, 1, 2, 6, 3, 8, 0, 7 };
            SelectSort<Integer> insertSort = new SelectSort<Integer>();
            Sort.testSort(insertSort, intgArr);
            Sort.testSort(insertSort, null);
```

```
        }
}
```

堆排序

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。

【例】关键字序列(10，15，56，25，30，70)和(70，56，30，25，15，10)分别满足堆性质(1)和(2)，故它们均是堆，其对应的完全二叉树分别如小根堆示例和大根堆示例所示：



根结点(亦称为堆顶)的关键字是堆里所有结点关键字中最小者的堆称为小顶堆。
根结点(亦称为堆顶)的关键字是堆里所有结点关键字中最大者，称为大顶堆。

堆是一种完全二叉树，一般使用数组来实现。堆排序也是一种选择性的排序，每次选择第i大的元素。

另外排序过程中借助了堆结构，堆就是一种完全二叉树，所以这里先要熟悉要用的二叉树几个性质：

N（N>1）个节点的的完全二叉树从层次从左自右编号，最后一个分枝节点（非叶子节点）的编号为 N/2 取整。

且对于编号 i（1<=i<=N）有：父节点为 i/2 向下取整；若2i>N，则节点i没有左孩子，否则其左孩子为2i；若2i+1>N，则没有右孩子，否则其右孩子为2i+1。

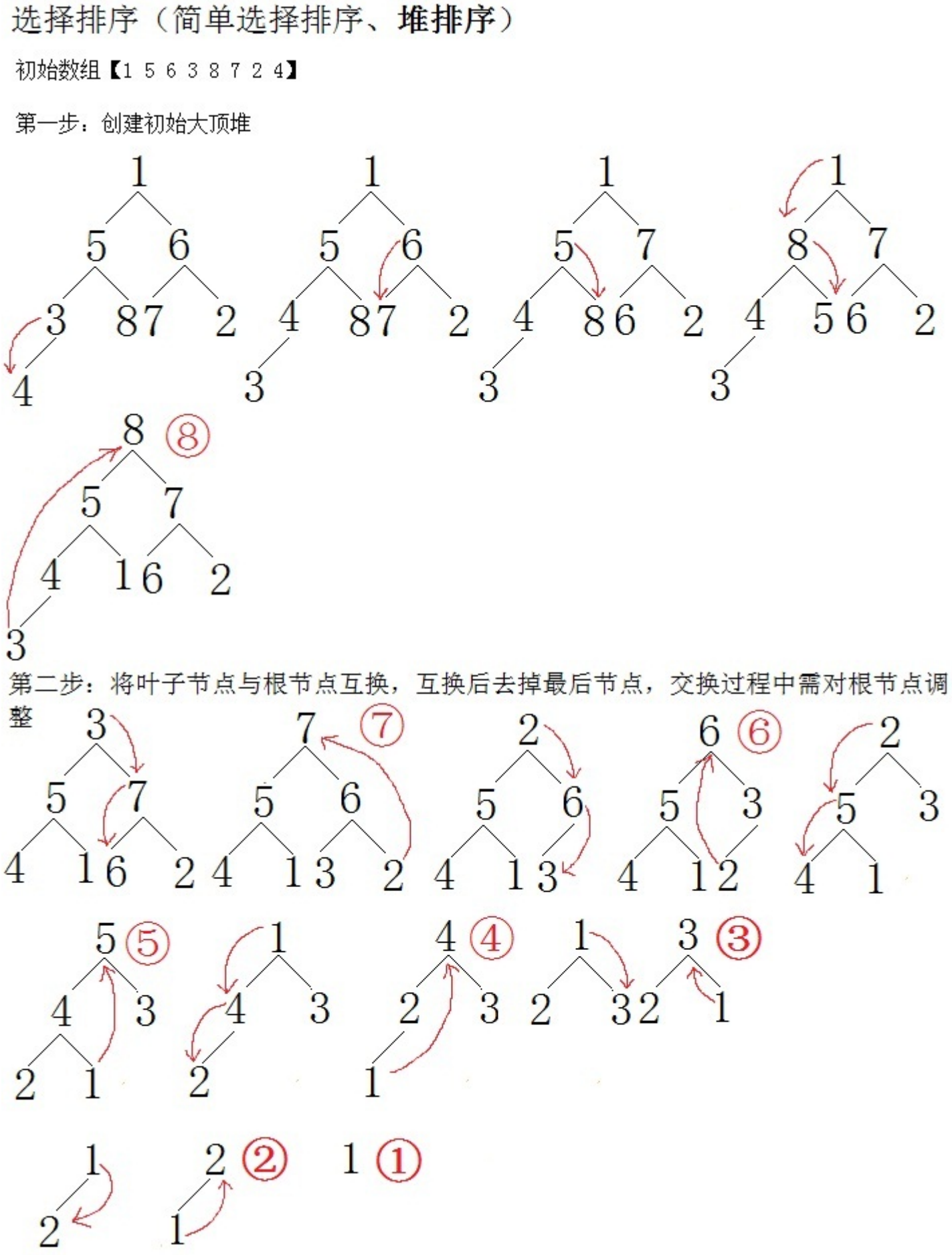
注，这里使用完全二叉树只是为了好描述算法，它只是一种逻辑结构，真真在实现时我们还是使用数组来存储这棵二叉树的，因为完全二叉树完全可以使用数组来存储。

算法描述：

堆排序其实最主要的两个过程：第一步，创建初始堆；第二步，交换根节点与最后一个非叶子节

第一步实现 ：从最后一个非叶子节点为开始向前循环每个会支节点，比较每个分支节点与他左右子节点，如果其中某个子节点比父节点大，则与父节点交换，交换后原父节点可能还小于原子节点的子节点，所以还需对原父节点进行调整，使用原父节点继续下沉，直到没有子节点或比左右子节点都大为止，调用过程可通过递归完成。当某个非叶子节点调整完毕后，再处理下一个非叶子节点，直到根节点也调整完成，这里初始堆就创建好了，这里我们创建的是大顶堆，即大的元素向树的根浮，这样排序最后得到的结果为升序，因为最大的从树中去掉，并从数组最后往前存放。

第二步实现：将树中的最后一个元素与堆顶元素进行交换，并从树中去掉最后叶子节点。交换后再按创建初始堆的算法调整根节点，如此下去直到树中只有一个节点为止。



```
package sort;

import java.util.Comparator;

public class HeapSort<E extends Comparable<E>> extends Sort<E> {

	/**
	 * 排序算法的实现，对数组中指定的元素进行排序
	 * @param array 待排序的数组
	 * @param from 从哪里开始排序
	 * @param end 排到哪里
	 * @param c 比较器
	 */
	public void sort(E[] array, int from, int end, Comparator<E> c) {
		//创建初始堆
		initialHeap(array, from, end, c);

		/*
		 * 对初始堆进行循环，且从最后一个节点开始，直接树只有两个节点止
		 * 每轮循环后丢弃最后一个叶子节点，再看作一个新的树
		 */
		for (int i = end - from + 1; i >= 2; i--) {
			//根节点与最后一个叶子节点交换位置，即数组中的第一个元素与最后一个元素互换
			swap(array, from, i - 1);
			//交换后需要重新调整堆
			adjustNote(array, 1, i - 1, c);
		}

	}

	/**
	 * 初始化堆
	 * 比如原序列为：7,2,4,3,12,1,9,6,8,5,10,11
	 * 则初始堆为：1,2,4,3,5,7,9,6,8,12,10,11
	 * @param arr 排序数组
```

```
     * @param from 从哪
     * @param end 到哪
     * @param c 比较器
     */
    private void initialHeap(E[] arr, int from, int end, Comparator<E> c) {
            int lastBranchIndex = (end - from + 1) / 2;//最后一个非叶子节点
            //对所有的非叶子节点进行循环 ，且从最一个非叶子节点开始
            for (int i = lastBranchIndex; i >= 1; i--) {
                    adjustNote(arr, i, end - from + 1, c);
            }
    }

    /**
     * 调整节点顺序，从父、左右子节点三个节点中选择一个最大节点与父节点转换
     * @param arr 待排序数组
     * @param parentNodeIndex 要调整的节点，与它的子节点一起进行调整
     * @param len 树的节点数
     * @param c 比较器
     */
    private void adjustNote(E[] arr, int parentNodeIndex, int len, Comparator<E> c) {
            int minNodeIndex = parentNodeIndex;
            //如果有左子树，i * 2为左子节点索引
            if (parentNodeIndex * 2 <= len) {
                    //如果父节点小于左子树时
                    if (c.compare(arr[parentNodeIndex - 1], arr[parentNodeIndex * 2 - 1]) <
                            minNodeIndex = parentNodeIndex * 2;//记录最大索引为左子节点索引
                    }

                    // 只有在有或子树的前提下才可能有右子树，再进一步断判是否有右子树
                    if (parentNodeIndex * 2 + 1 <= len) {
                            //如果右子树比最大节点更大
                            if (c.compare(arr[minNodeIndex - 1], arr[(parentNodeIndex * 2 +
                                    minNodeIndex = parentNodeIndex * 2 + 1;//记录最大索引为右
                            }
                    }
            }
```

```
                //如果在父节点、左、右子节点三都中，最大节点不是父节点时需交换，把最大的与父节点交换，
                if (minNodeIndex != parentNodeIndex) {
                        swap(arr, parentNodeIndex - 1, minNodeIndex - 1);
                        //交换后可能需要重建堆，原父节点可能需要继续下沉
                        if (minNodeIndex * 2 <= len) {//是否有子节点，注，只需判断是否有左子树即可
                                adjustNote(arr, minNodeIndex, len, c);
                        }
                }
        }

        /**
        * 测试
        * @param args
        */
        public static void main(String[] args) {
                Integer[] intgArr = { 7, 2, 4, 3, 12, 1, 9, 6, 8, 5, 10, 11 };
                HeapSort<Integer> sort = new HeapSort<Integer>();
                HeapSort.testSort(sort, intgArr);
                HeapSort.testSort(sort, null);
        }

}
```

1.2 常用排序算法分析与实现（二）（Java版）

发表时间: 2009-12-13

交换排序

冒泡排序

将最后一个元素与倒数第二个元素对比，如果最后一个元素比倒数第二个小，则交换两个元素的位置，再用倒数第二个元素与倒数第三个元数对比，直到比到第一个元素，这样经过第一趟排序后得到第一个最小元素。如此反复几过N（N=length-1）次后可得到排序结果。


```
package sort;


import java.util.Comparator;


/**
 * 冒泡排序算法
 * @author jzj
 * @date 2009-12-9
 *
 * @param <E>
 */
public class BubbleSort<E extends Comparable<E>> extends Sort<E> {


        /**
```

```
         * 排序算法的实现，对数组中指定的元素进行排序
         * @param array 待排序的数组
         * @param from 从哪里开始排序
         * @param end 排到哪里
         * @param c 比较器
         */
        public void sort(E[] array, int from, int end, Comparator<E> c) {
                //需array.length - 1轮比较
                for (int k = 1; k < end - from + 1; k++) {
                        //每轮循环中从最后一个元素开始向前起泡，直到i=k止，即i等于轮次止
                        for (int i = end - from; i >= k; i--) {
                                //按照一种规则（后面元素不能小于前面元素）排序
                                if (c.compare(array[i], array[i - 1]) < 0) {
                                        //如果后面元素小于了（当然是大于还是小于要看比较器实现了）前
                                        swap(array, i, i - 1);
                                }
                        }
                }
        }

        /**
        * 测试
        * @param args
        */
        public static void main(String[] args) {
                Integer[] intgArr = { 7, 2, 4, 3, 12, 1, 9, 6, 8, 5, 11, 10 };
                BubbleSort<Integer> sort = new BubbleSort<Integer>();
                BubbleSort.testSort(sort, intgArr);
                BubbleSort.testSort(sort, null);
        }
}
```

快速排序

快速排序采用了分治法的思想，把大的问题分解为同类型的小问题。

一般分如下步骤：

1）选择一个中枢元素（有很多选法，我的实现里使用第一个元素为中枢的简单方法）

2）以该中枢元素为基准点，将小于中枢的元素放在中枢后集合的前部分，比它大的在集合后部分，待集合基本排序完成后（此时前部分元素小于后部分元素），把中枢元素放在合适的位置。
3）根据中枢元素最后确定的位置，把数组分成三部分，左边的，右边的，枢纽元素自己，对左边的，右边的分别递归调用快速排序算法即可。

这里的重点与难点在于第二步，实现的方式有很多种，我这里实现了三种。
第一种实现（partition1方法）：
以第一个元素为中枢元素，在中枢元素后面集合中从前往后寻找第一个比中枢元素小的元素，并与第一个元素交换，然后从剩余的元素中寻找第二个比中枢元素小的元素，并与第二位元素交换，这样直到所有小于中枢元素找完为止，并记下最后一次放置小于中枢的元素位置minIndex（即小于中枢与大于中枢的分界），并将中枢元素与minIndex位置元素互换，然后对中枢元素两边的序列进行同样的操作。
此种实现最为简洁，处理过程中不需要把中枢元素移来移去，只是在其它元素完成基本排序后（前部分小于后部分元素）再把中枢元素放置到适当的位置

第二种实现（partition2方法）：
以第一个元素为中枢元素，刚开始时使用低指针指向中枢元素。当中枢元素在低指针位置时，此时我们判断高指针指向的元素是否小于中枢元素，如果大于中枢元素则高指针继续向头移动，如果小于则与中枢元素交换，此时中枢元素被移到了高指针位置；当中枢元素在高指针位置时，我们此时判断低指针指向的元素是否大于中枢元素，如果小于中枢元素则低指针继续向尾移动，如果大于则与中枢元素交换，此时中枢元素又回到了低指针位置；这时是拿高还是低指针所指向的元素与中枢比较时根据前面逻辑来处理，直到高低指针指向同一位置则完成一轮排序，然后再对中枢元素两边的序列进行同样的操作直到排序完成

此种实现逻辑比较好理解，中枢元素的永远在低指针或指针所指向的位置，每次找到需处理的元 素后，要与中枢交换，中枢就像皮球一样从这里踢到那里，又从那里踢到这里。但此种实现会频繁地交换中枢元素，性能可能不如第一种

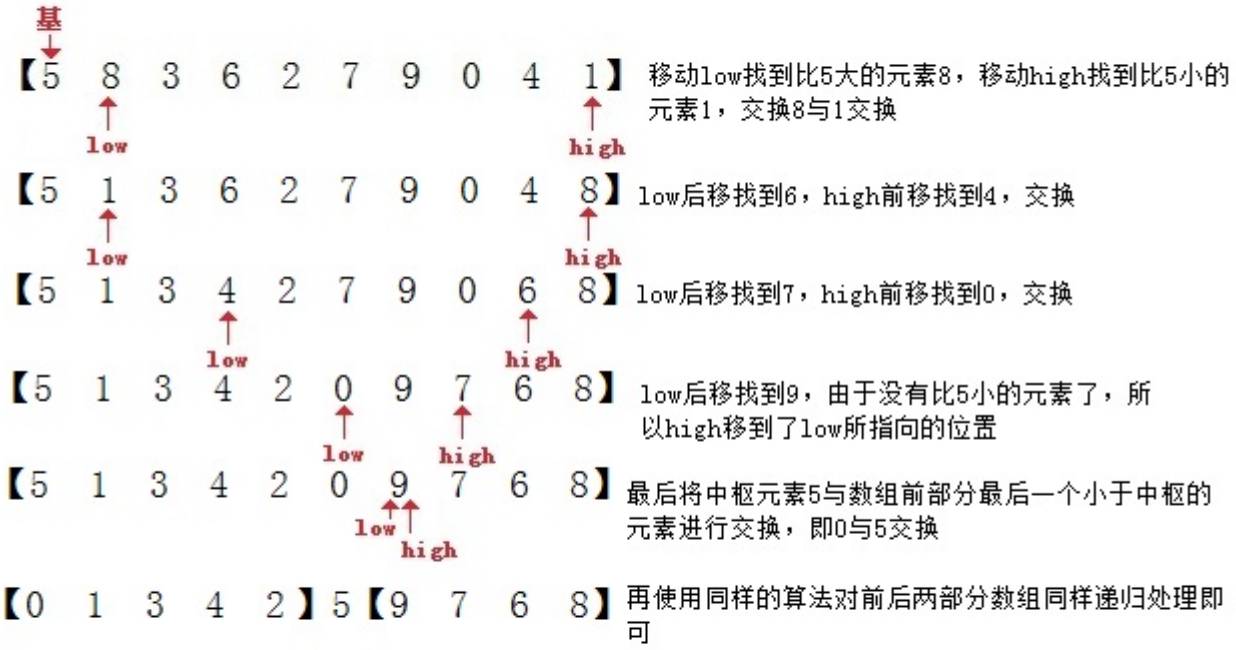
第三种实现（partition3方法）：

此种方式与前两种方式不太一样，同时移动高低指针，低指针向尾找出大于等于中枢的元素，而高向头找出小于中枢的元素，待两者都找出后交换高低指针所指向的元素，直到高低指针指向同一位置止，然后比较中枢与高低指针所指向的元素大小，如果中枢元素大，则直接与高低指针元素交换，如果中枢元素小于等于高低指针元素，则中枢元素与高低指针前一元素交换，完成一轮比较，然后再对中枢元素两边的序列进行同样的操作直到排序完成

此种方式有点难度，在移动元素时要注意的是：与中枢相等的元素也要向集合后部移动，不然的话如[3,3,0,3,3]第一轮排序结果不准确，虽然最后结果正确。当中枢后面的元素集合移动完成后，还得要把中枢元素放置在集合中的合适位置，这就需要找准集合中前部分与后部分的边界，最后只能把中枢元素与最后一个小于中枢的元素进位置互换。但此种实现方式与第一种有点像，也不需要把中枢元素调来调去的，而是待后面集合排序完成

后将中枢放入适当位置

实现三：
基准点5先不移动，但low与high都会移动，low找大于等于中枢的元素，high找小于中枢的元素，然后交换low与high，直到low=high止，最后将中枢元素5与前部分数组中的最后一个元素交换



这里的难点与重点就是在于最后一步：怎样将中枢元素放置到适当位置，此种算法不如前两种很易容就知道前后两部分数组的边界，所以在交换之前一定要与low和high指向的元素进行比较，如果大于等于中枢元素，则与low和high前面一个元素进行交换，如果小于中枢元素，则直接与low和high指向的元素交换，具体过程请参照代码

```
package sort;


import java.util.Arrays;
import java.util.Comparator;


/**
 * 快速排序算法
 * @author jzj
 * @date 2009-12-9
 *
 * @param <E>
 */
public class QuickSort<E extends Comparable<E>> extends Sort<E> {
```

```
/**
 * 排序算法的实现，对数组中指定的元素进行排序
 * @param array 待排序的数组
 * @param from 从哪里开始排序
 * @param end 排到哪里
 * @param c 比较器
 */
public void sort(E[] array, int from, int end, Comparator<E> c) {
        quickSort(array, from, end, c);
}

/**
 * 递归快速排序实现
 * @param array 待排序数组
 * @param low 低指针
 * @param high 高指针
 * @param c 比较器
 */
private void quickSort(E[] array, int low, int high, Comparator<E> c) {
        /*
         * 如果分区中的低指针小于高指针时循环；如果low=higth说明数组只有一个元素，无需再处理；
         * 如果low > higth，则说明上次枢纽元素的位置pivot就是low或者是higth，此种情况
         * 下分区不存，也不需处理
         */
        if (low < high) {
                //对分区进行排序整理
                int pivot = partition1(array, low, high, c);
                /*
                 * 以pivot为边界，把数组分成三部分[low, pivot - 1]、[pivot]、[pivot + 1,
                 * 其中[pivot]为枢纽元素，不需处理，再对[low, pivot - 1]与[pivot + 1, high
                 * 各自进行分区排序整理与进一步分区
                 */
                quickSort(array, low, pivot - 1, c);
                quickSort(array, pivot + 1, high, c);
        }
```

```
    }

    /**
     * 实现一
     *
     * @param array 待排序数组
     * @param low 低指针
     * @param high 高指针
     * @param c 比较器
     * @return int 调整后中枢位置
     */
    private int partition1(E[] array, int low, int high, Comparator<E> c) {
            E pivotElem = array[low];//以第一个元素为中枢元素
            //从前向后依次指向比中枢元素小的元素，刚开始时指向中枢，也是小于与大小中枢的元素的分界
            int border = low;

            /*
             * 在中枢元素后面的元素中查找小于中枢元素的所有元素，并依次从第二个位置从前往后存放
             * 注，这里最好使用i来移动，如果直接移动low的话，最后不知道数组的边界了，但后面需要
             * 知道数组的边界
             */
            for (int i = low + 1; i <= high; i++) {
                    //如果找到一个比中枢元素小的元素
                    if (c.compare(array[i], pivotElem) < 0) {
                            swap(array, ++border, i);//border前移，表示有小于中枢元素的元素
                    }
            }
            /*
             * 如果border没有移动时说明说明后面的元素都比中枢元素要大，border与low相等，此时是
             * 同一位置交换，是否交换都没关系；当border移到了high时说明所有元素都小于中枢元素，此
             * 时将中枢元素与最后一个元素交换即可，即low与high进行交换，大的中枢元素移到了 序列最
             * 后；如果 low <minIndex< high，表 明中枢后面的元素前部分小于中枢元素，而后部分大于
             * 中枢元素，此时中枢元素与前部分数组中最后一个小于它的元素交换位置，使得中枢元素放置在
             * 正确的位置
             */
            swap(array, border, low);
            return border;
```

```
}

/**
 * 实现二
 *
 * @param array 待排序数组
 * @param low 待排序区低指针
 * @param high 待排序区高指针
 * @param c 比较器
 * @return int 调整后中枢位置
 */
private int partition2(E[] array, int low, int high, Comparator<E> c) {
        int pivot = low;//中枢元素位置，我们以第一个元素为中枢元素
        //退出条件这里只可能是 low = high
        while (true) {
                if (pivot != high) {//如果中枢元素在低指针位置时，我们移动高指针
                        //如果高指针元素小于中枢元素时，则与中枢元素交换
                        if (c.compare(array[high], array[pivot]) < 0) {
                                swap(array, high, pivot);
                                //交换后中枢元素在高指针位置了
                                pivot = high;
                        } else {//如果未找到小于中枢元素，则高指针前移继续找
                                high--;
                        }
                } else {//否则中枢元素在高指针位置
                        //如果低指针元素大于中枢元素时，则与中枢元素交换
                        if (c.compare(array[low], array[pivot]) > 0) {
                                swap(array, low, pivot);
                                //交换后中枢元素在低指针位置了
                                pivot = low;
                        } else {//如果未找到大于中枢元素，则低指针后移继续找
                                low++;
                        }
                }
                if (low == high) {
                        break;
                }
```

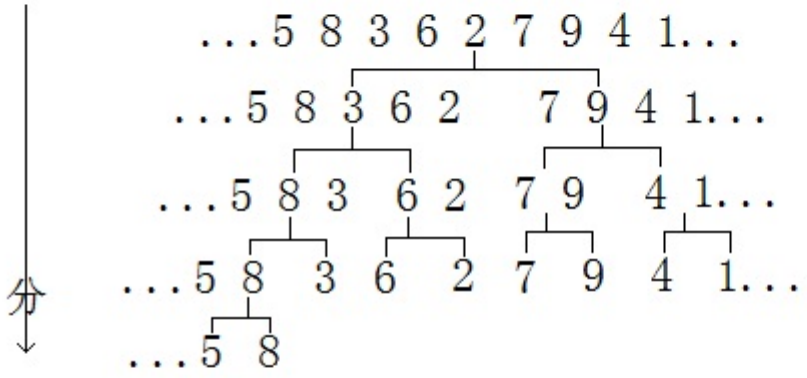
```
        }
        //返回中枢元素所在位置，以便下次分区
        return pivot;
    }

    /**
     * 实现三
     *
     * @param array 待排序数组
     * @param low 待排序区低指针
     * @param high 待排序区高指针
     * @param c 比较器
     * @return int 调整后中枢位置
     */
    private int partition3(E[] array, int low, int high, Comparator<E> c) {
        int pivot = low;//中枢元素位置，我们以第一个元素为中枢元素
        low++;
        //----调整高低指针所指向的元素顺序，把小于中枢元素的移到前部分，大于中枢元素的移到后面
        //退出条件这里只可能是 low = high

        while (true) {
            //如果高指针未超出低指针
            while (low < high) {
                //如果低指针指向的元素大于或等于中枢元素时表示找到了，退出，注：等于
                if (c.compare(array[low], array[pivot]) >= 0) {
                    break;
                } else {//如果低指针指向的元素小于中枢元素时继续找
                    low++;
                }
            }

            while (high > low) {
                //如果高指针指向的元素小于中枢元素时表示找到，退出
                if (c.compare(array[high], array[pivot]) < 0) {
                    break;
                } else {//如果高指针指向的元素大于中枢元素时继续找
                    high--;
```

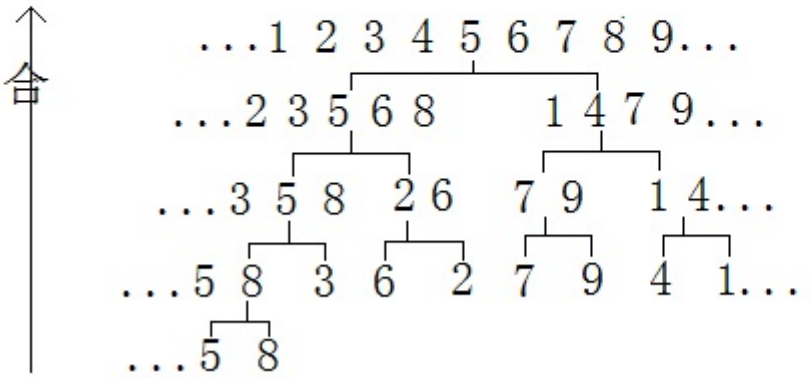
```
                        }
                }
                //退出上面循环时 low = high
                if (low == high) {
                        break;
                }

                swap(array, low, high);
        }

        //----高低指针所指向的元素排序完成后，还得要把中枢元素放到适当的位置
        if (c.compare(array[pivot], array[low]) > 0) {
                //如果退出循环时中枢元素大于了低指针或高指针元素时，中枢元素需与low元素交换
                swap(array, low, pivot);
                pivot = low;
        } else if (c.compare(array[pivot], array[low]) <= 0) {
                swap(array, low - 1, pivot);
                pivot = low - 1;
        }

        //返回中枢元素所在位置，以便下次分区
        return pivot;
    }

    /**
    * 测试
    * @param args
    */
    public static void main(String[] args) {
            Integer[] intgArr = { 3, 1, 1, 1, 1, 1, 1 };
            QuickSort<Integer> sort = new QuickSort<Integer>();
            QuickSort.testSort(sort, intgArr);
            QuickSort.testSort(sort, null);
    }
}
```

归并排序

归并排序

归并排序分两步操作：第一步将数组分解成更小的数组，直到数组只有一个元素止，每次划分点为(len-1)/2=mid，将数组分成 [from ,mid]与[mid+1,to]；第二步就是将分解的数组两两的合并，合并后的数组是有序的，直到合并成一个数组止，合并过程中会用一个临时数组，用来临时存储合并后的结果，待每次合并完后，再将临时数组拷贝到原数组对应的位置中






```
package sort;


import java.lang.reflect.Array;
import java.util.Comparator;


/**
 * 归并排序算法
 * @author jzj
 * @date 2009-12-11
 *
 * @param <E>
 */
```

```
public class MergeSort<E extends Comparable<E>> extends Sort<E> {

	/**
	 * 排序算法的实现，对数组中指定的元素进行排序
	 * @param array 待排序的数组
	 * @param from 从哪里开始排序
	 * @param end 排到哪里
	 * @param c 比较器
	 */
	public void sort(E[] arr, int from, int end, Comparator<E> c) {
		partition(arr, from, end, c);
	}

	/**
	 * 递归划分数组
	 * @param arr
	 * @param from
	 * @param end
	 * @param c void
	 */
	private void partition(E[] arr, int from, int end, Comparator<E> c) {
		//划分到数组只有一个元素时才不进行再划分
		if (from < end) {
			//从中间划分成两个数组
			int mid = (from + end) / 2;
			partition(arr, from, mid, c);
			partition(arr, mid + 1, end, c);
			//合并划分后的两个数组
			merge(arr, from, end, mid, c);
		}
	}

	/**
	 * 数组合并，合并过程中对两部分数组进行排序
	 * 前后两部分数组里是有序的
	 * @param arr
	 * @param from
```

```
     * @param end
     * @param mid
     * @param c void
     */
    private void merge(E[] arr, int from, int end, int mid, Comparator<E> c) {
            E[] tmpArr = (E[]) Array.newInstance(arr[0].getClass(), end - from + 1);
            int tmpArrIndex = 0;//指向临时数组
            int part1ArrIndex = from;//指向第一部分数组
            int part2ArrIndex = mid + 1;//指向第二部分数组

            //由于两部分数组里是有序的，所以每部分可以从第一个元素依次取到最后一个元素，再对两部分
            //取出的元素进行比较。只要某部分数组元素取完后，退出循环
            while ((part1ArrIndex <= mid) && (part2ArrIndex <= end)) {
                    //从两部分数组里各取一个进行比较，取最小一个并放入临时数组中
                    if (c.compare(arr[part1ArrIndex], arr[part2ArrIndex]) < 0) {
                            //如果第一部分数组元素小，则将第一部分数组元素放入临时数组中，并且临
                            //tmpArrIndex下移一个以做好下次存储位置准备，前部分数组指针part1Arr
                            //也要下移一个以便下次取出下一个元素与后部分数组元素比较
                            tmpArr[tmpArrIndex++] = arr[part1ArrIndex++];
                    } else {
                            //如果第二部分数组元素小，则将第二部分数组元素放入临时数组中
                            tmpArr[tmpArrIndex++] = arr[part2ArrIndex++];
                    }
            }
            //由于退出循环后，两部分数组中可能有一个数组元素还未处理完，所以需要额外的处理，当然不
            //能两部分数组都有未处理完的元素，所以下面两个循环最多只有一个会执行，并且都是大于已放
            //临时数组中的元素
            while (part1ArrIndex <= mid) {
                    tmpArr[tmpArrIndex++] = arr[part1ArrIndex++];
            }
            while (part2ArrIndex <= end) {
                    tmpArr[tmpArrIndex++] = arr[part2ArrIndex++];
            }

            //最后把临时数组拷贝到源数组相同的位置
            System.arraycopy(tmpArr, 0, arr, from, end - from + 1);
    }
```

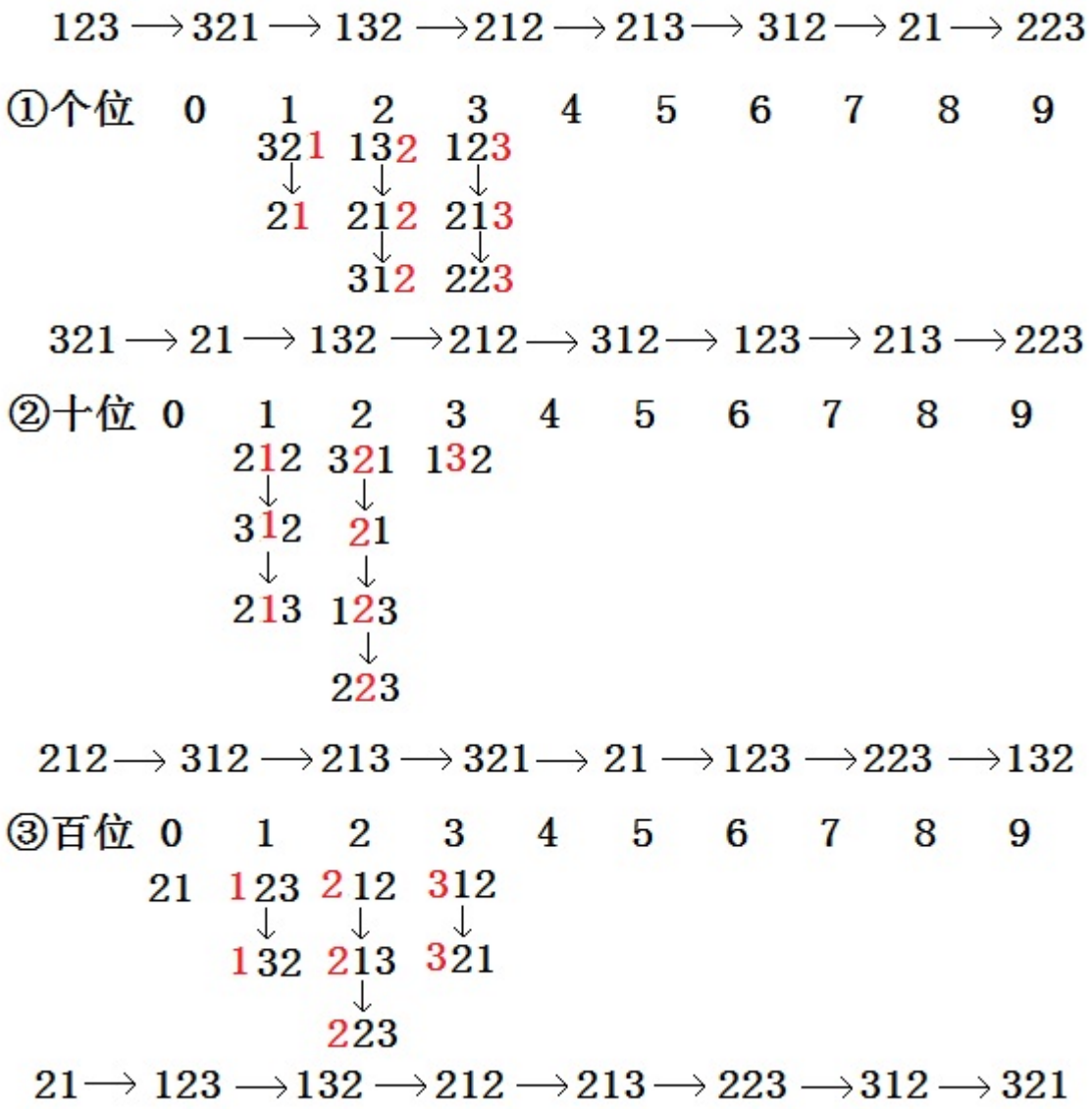
```
        /**
         * 测试
         * @param args
         */
        public static void main(String[] args) {
                Integer[] intgArr = { 5, 9, 1, 4, 1, 2, 6, 3, 8, 0, 7 };
                MergeSort<Integer> insertSort = new MergeSort<Integer>();
                Sort.testSort(insertSort, intgArr);
                Sort.testSort(insertSort, null);
        }
}
```

基数排序

基数排序的主要思路是,将所有待比较数值(注意,必须是正整数)统一为同样的数位长度,数位较短的数前面补零. 然后, 从最低位开始, 依次进行一次稳定排序.这样从最低位排序一直到最高位排序完成以后, 数列就变成一个有序序列.

它的理论比较容易理解，但实现却有一点绕。

基数排序




```
package sort;

import java.util.Arrays;

public class RadixSort {

        /**
         * 取数x上的第d位数字
         * @param x 数
         * @param d 第几位，从低位到高位
         * @return
         */
```

```
public int digit(long x, long d) {

        long pow = 1;
        while (--d > 0) {
                pow *= 10;
        }
        return (int) (x / pow % 10);
}

/**
 * 基数排序实现，以升序排序（下面程序中的位记录器count中，从第0个元素到第9个元素依次用来
 * 记录当前比较位是0的有多少个..是9的有多少个数，而降序时则从第0个元素到第9个元素依次用来
 * 记录当前比较位是9的有多少个..是0的有多少个数）
 * @param arr 待排序数组
 * @param digit 数组中最大数的位数
 * @return
 */
public long[] radixSortAsc(long[] arr) {
        //从低位往高位循环
        for (int d = 1; d <= getMax(arr); d++) {
                //临时数组，用来存放排序过程中的数据
                long[] tmpArray = new long[arr.length];
                //位记数器，从第0个元素到第9个元素依次用来记录当前比较位是0的有多少个..是9的有
                int[] count = new int[10];
                //开始统计0有多少个，并存储在第0位，再统计1有多少个，并存储在第1位..依次统计至
                for (int i = 0; i < arr.length; i++) {
                        count[digit(arr[i], d)] += 1;
                }
                /*
                 * 比如某次经过上面统计后结果为：[0, 2, 3, 3, 0, 0, 0, 0, 0, 0]则经过下面计
                 * [0, 2, 5, 8, 8, 8, 8, 8, 8, 8]但实质上只有如下[0, 2, 5, 8, 0, 0, 0, 0
                 * 非零数才用到，因为其他位不存在，它们分别表示如下：2表示比较位为1的元素可以
                 * 位置，5表示比较位为2的元素可以存放在4、3、2三个(5-2=3)位置，8表示比较位为
                 * 7、6、5三个(8-5=3)位置
                 */
                for (int i = 1; i < 10; i++) {
                        count[i] += count[i - 1];
```

```
                }

                /*
                 * 注，这里只能从数组后往前循环，因为排序时还需保持以前的已排序好的 顺序，不应
                 * 乱原来已排好的序，如果从前往后处理，则会把原来在前面会摆到后面去，因为在处
                 * 元素的位置时，位记数器是从大到到小（count[digit(arr[i], d)]--）的方式来处
                 * 理的，即先存放索引大的元素，再存放索引小的元素，所以需从最后一个元素开始处
                 * 如有这样的一个序列[212,213,312]，如果按照从第一个元素开始循环的话，经过第
                 * 后（个位）排序后，得到这样一个序列[312,212,213]，第一次好像没什么问题，但
                 * 从第二轮开始出现，第二轮排序后，会得到[213,212,312]，这样个位为3的元素本应
                 * 放在最后，但经过第二轮后却排在了前面了，所以出现了问题
                 */
                for (int i = arr.length - 1; i >= 0; i--) {//只能从最后一个元素往前处理
                        //for (int i = 0; i < arr.length; i++) {//不能从第一个元素开始循环
                        tmpArray[count[digit(arr[i], d)] - 1] = arr[i];
                        count[digit(arr[i], d)]--;
                }

                System.arraycopy(tmpArray, 0, arr, 0, tmpArray.length);
        }
        return arr;
}

/**
 * 基数排序实现，以降序排序（下面程序中的位记录器count中，从第0个元素到第9个元素依次用来
 * 记录当前比较位是0的有多少个..是9的有多少个数，而降序时则从第0个元素到第9个元素依次用来
 * 记录当前比较位是9的有多少个..是0的有多少个数）
 * @param arr 待排序数组
 * @return
 */
public long[] radixSortDesc(long[] arr) {
        for (int d = 1; d <= getMax(arr); d++) {
                long[] tmpArray = new long[arr.length];
                //位记数器，从第0个元素到第9个元素依次用来记录当前比较位是9的有多少个..是0的有
                int[] count = new int[10];
                //开始统计0有多少个，并存储在第9位，再统计1有多少个，并存储在第8位..依次统计
                //到9有多少个，并存储在第0位
```

```
                for (int i = 0; i < arr.length; i++) {
                        count[9 - digit(arr[i], d)] += 1;
                }

                for (int i = 1; i < 10; i++) {
                        count[i] += count[i - 1];
                }

                for (int i = arr.length - 1; i >= 0; i--) {
                        tmpArray[count[9 - digit(arr[i], d)] - 1] = arr[i];
                        count[9 - digit(arr[i], d)]--;
                }

                System.arraycopy(tmpArray, 0, arr, 0, tmpArray.length);
        }
        return arr;
    }

    private int getMax(long[] array) {
        int maxlIndex = 0;
        for (int j = 1; j < array.length; j++) {
                if (array[j] > array[maxlIndex]) {
                        maxlIndex = j;
                }
        }
        return String.valueOf(array[maxlIndex]).length();
    }

    public static void main(String[] args) {
        long[] ary = new long[] { 123, 321, 132, 212, 213, 312, 21, 223 };
        RadixSort rs = new RadixSort();
        System.out.println("升 - " + Arrays.toString(rs.radixSortAsc(ary)));

        ary = new long[] { 123, 321, 132, 212, 213, 312, 21, 223 };
        System.out.println("降 - " + Arrays.toString(rs.radixSortDesc(ary)));
    }
}
```

时间复杂度与空间复杂度对比表

各类排序算法时间复杂度和空间复杂度对比表

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		$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	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。

排序算法

常用排序算法分析与实现（Java版）

作者: junJZ_2008
http://jiangzhengjun.javaeye.com

本书由JavaEye提供电子书DIY功能制作并发行。
更多精彩博客电子书，请访问：http://www.javaeye.com/blogs/pdf