链滴

排序算法

作者：SLiMyLove
原文链接：https://ld246.com/article/1542931019100
来源网站：链滴

1. 选择排序

```
import java.util.Arrays;

/**
 * 选择排序：最基本的排序方法，从前往后的排序
 * 从第一个位置开始，依次和剩余的数字进行比较
 * 如果小则交换；大，则不做操作。
 * 比较完毕，换此时数组的第二个位置，依次和3到结尾的数字进行比较
 * ……
 * 直到倒数第二个位置，和最后一个位置的数比较完毕，排序完成
 */
public class SelectSort {
    public static void main(String[] args) {
        // int[] a = {1, 2, 4, 5, 7, 4, 5 ,3 ,9 ,0};
        int[] a = {10, 20, 15, 0, 6, 7, 2, 1, 65, 55};
        System.out.println(Arrays.toString(a));
        // 调用选择排序
        selectSort(a);
        System.out.println(Arrays.toString(a));
    }

    private static void selectSort(int[] numbers) {
        int size = numbers.length;
        int tmp = 0;
        // 外层循环：含义：从第一个位置开始，以此和剩余未排序的位置比较
        for (int i = 0; i < size -1; i++) {
            // 内层循环：含义：选择的位置之后未排序的数据 i + 1
            for (int j = i + 1; j < size; j++) {
                // 如果后一个数比前一个数的数值小，则交换两者的位置
                if (numbers[j] < numbers[i]) {
                    tmp = numbers[i];
                    numbers[i] = numbers[j];
                    numbers[j] = tmp;
                }
            }
        }
    }
}
```

2. 冒泡排序

```
import java.util.Arrays;

/**
 * 冒泡排序：依次对相邻的两个值进行比较，从后往前的排序
 */
public class BubbleSort {
    public static void main(String[] args) {
        // int[] a = {1, 2, 4, 5, 7, 4, 5 ,3 ,9 ,0};
        int[] a = {10, 20, 15, 0, 6, 7, 2, 1, 65, 55};
        System.out.println(Arrays.toString(a));
```

```
        // 调用冒泡排序
        bubbleSort(a);
        System.out.println(Arrays.toString(a));
    }

    private static void bubbleSort(int[] numbers) {
        int size = numbers.length;
        int tmp;
        boolean changed;
        do {
            changed = false;
            size -= 1;
            for (int i = 0; i < size; i++) {
                if (numbers[i] > numbers[i + 1]) {
                    tmp = numbers[i];
                    numbers[i] = numbers[i + 1];
                    numbers[i + 1] = tmp;
                    changed = true;
                }
            }
        } while (changed); //如果上一次的循环没有排序，则表明已经排序完成，直接结束
    }
}
```

3. 快速排序

```
import java.util.Arrays;

/**
 * 快速排序：
 * 1. 从序列中挑出一个元素，作为"基准"(pivot).(基准不是固定不动的，也会被交换)
 * 2. 把所有比基准值小的元素放在基准前面，所有比基准值大的元素放在基准的后面（相同的数可以任一边），这个称为分区(partition)操作。
 * 3. 对每个分区递归地进行步骤1~2，递归的结束条件是序列的大小是0或1，这时整体已经被排好序。
 */
public class QuickSort {
    public static void main(String[] args) {
        // int[] arr = new int[]{1, 4, 8, 2, 55, 3, 4, 8, 6, 4, 0, 11, 34, 90, 23, 54, 77, 9, 2, 9, 4, 10};
        int[] arr = {10, 20, 15, 0, 6, 7, 2, 1, 65, 55};
        System.out.println(Arrays.toString(arr));
        qSort(arr, 0, arr.length - 1);
        System.out.println(Arrays.toString(arr));
    }

    private static void qSort(int[] arr, int head, int tail) {
        // 结束条件，1.头和尾的哨兵相遇
        // 2. 数组为空或者长度小于等于1
        if (head >= tail || arr == null || arr.length <= 1) {
            return;
        }
        // 去中间的数值为基准值
        int i = head, j = tail, pivot = arr[(head + tail) / 2];
        // 只有当i <= j的时候，即头哨兵小于尾哨兵，才会循环
```

```
        while (i <= j) {
            // 从左往右，找到第一个大于基准值的数值所在的位置
            while (arr[i] < pivot) {
                ++i;
            }
            // 从右往左，找到第一个小于基准值的数值所在的位置
            while (arr[j] > pivot) {
                --j;
            }
            // 如果左右哨兵的位置仍然是左右分布，没有重合或者交错，进行数据的交换
            // 如果两个哨兵中间就间隔一个数，在if内，分别自增和自减，会导致i == j
            // 如果两个哨兵中间没有间隔，是相邻的，在if内，分别自增和自减，会导致 i > j
            if (i < j) {
                int t = arr[i];
                arr[i] = arr[j];
                arr[j] = t;
                ++i;
                --j;
            } else if (i == j) {// 如果左右哨兵遇到了，增大i的位置,为了下一步递归的使用
                ++i;
            }
        }
        // 递归判断左侧
        qSort(arr, head, j);
        // 递归判断右侧
        qSort(arr, i, tail);
    }
}
```

4. 归并排序

```
import java.util.Arrays;

/**
 * 归并排序
 * 归并排序算法主要依赖归并(Merge)操作。归并操作指的是将两个已经排序的序列合并成一个序列操作
 */
public class MergeSort {
    public static void main(String[] args) {
        // int[] a = {1, 2, 4, 5, 7, 4, 5 ,3 ,9 ,0};
        // 偶数个
        // int[] a = {10, 20, 15, 0, 6, 7, 2, 1, 65, 55};
        // 奇数个
        int[] a = {10, 20, 15, 6, 7, 2, 1, 65, 55};
        System.out.println(Arrays.toString(a));
        // 调用归并排序 递归的方法实现
        mergeSort(a);
        System.out.println(Arrays.toString(a));
    }

    private static void mergeSort(int[] arr) {
        int len = arr.length;
        int[] result = new int[len];
```

```
        merge_sort_recursive(arr, result, 0, len - 1);
    }

    private static void merge_sort_recursive(int[] arr, int[] result, int start, int end) {
        // 当待递归的数据长度等于1的时候（也就是将数据切分为一个一个的个体），递归开始回溯，
行 归并 的操作
        if (start >= end)
            return;

        // 每次都将数组拆分为两个，在进行下一级的递归
        // 确定递归每部分的start和end
        int len = end - start, mid = (len >> 1) + start;
        int start1 = start, end1 = mid;
        int start2 = mid + 1, end2 = end;

        // 开始递归，直到待递归的长度等于1的时候（也就是将数据切分为一个一个的个体），
        merge_sort_recursive(arr, result, start1, end1);
        merge_sort_recursive(arr, result, start2, end2);

        // 归并 操作。这一步的操作在递归的作用下，实际上每一部分都是一个有序的数组，所以才可
下方简单的比较就可以
        int k = start;
        // 给对应的result数组赋值
        while (start1 <= end1 && start2 <= end2)
            result[k++] = arr[start1] < arr[start2] ? arr[start1++] : arr[start2++];
        // 如果第一部分有剩余的
        while (start1 <= end1)
            result[k++] = arr[start1++];
        // 如果第二部分有剩余的
        while (start2 <= end2)
            result[k++] = arr[start2++];

        // 将排序后的值重新赋值给arr的同样的位置
        for (k = start; k <= end; k++)
            arr[k] = result[k];
    }
}
```

5. 堆排序

```
import java.util.Arrays;

/**
 * 堆排序：
 * 堆排序就是把
 * 1） 数组堆化，并进行最大堆调整，结果就是堆的堆顶 是数组最大的那个值；并且子节点永远小于
节点
 * 2） 将最大堆堆顶的最大数取出，
 * 3） 将剩余的堆继续调整为最大堆，再次将堆顶的最大数取出，
 * 4） 这个过程持续到剩余数只有一个时结束。在堆中定义以下几种操作：
 *      1. 最大堆调整（Max-Heapify）：将堆的末端子节点作调整，使得子节点永远小于父节点
 *      2. 创建最大堆（Build-Max-Heap）：将堆所有数据重新排序，使其成为最大堆
 *      3. 堆排序（Heap-Sort）：移除位在第一个数据的根节点，并做最大堆调整的递归运算
 */
```

```
public class HeapSort {
    public static void main(String[] args) {
        // int[] arr = new int[]{3,5,3,0,8,6,1,5,8,6,2,4,9,4,7,0,1,8,9,7,3,1,2,5,9,7,4,0,2,6};
        int[] arr = {10, 20, 15, 0, 6, 7, 2, 1, 65, 55};
        System.out.println(Arrays.toString(arr));
        // 调用堆排序
        heapSort(arr);
        System.out.println(Arrays.toString(arr));
    }

    /**
     * 堆排序的主要入口方法，共两步。
     *
     * @param arr
     */
    private static void heapSort(int[] arr) {
        /*
         *  第一步：将数组堆化
         *  beginIndex = 第一个非叶子节点。
         *  从第一个非叶子节点开始即可。无需从最后一个叶子节点开始。
         *  叶子节点可以看作已符合堆要求的节点，根节点就是它自己且自己以下值为最大。
         */
        // len 最大下标值
        int len = arr.length - 1;
        // 第一个非叶子节点的下标（计算方法是：第一个非叶子节点是最后一个叶子节点的父节点）
        int beginIndex = (len - 1) >> 1;
        for (int i = beginIndex; i >= 0; i--) {
            maxHeapify(arr, i, len);
        }

        /*
         * 第二步：对堆化数据排序
         * 每次都是移出最顶层的根节点A[0]，与最尾部节点位置调换，同时遍历长度 - 1。
         * 然后从新整理被换到根节点的末尾元素，使其符合堆的特性。
         * 直至未排序的堆长度为 0。
         */
        for (int i = len; i > 0; i--) {
            swap(arr, 0, i);
            maxHeapify(arr, 0, i - 1);
        }
    }

    private static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    /**
     * 调整索引为 index 处的数据，使其符合堆的特性。
     *
     * @param arr
     * @param index 需要堆化处理的数据的索引
     * @param len   未排序的堆（数组）的长度
```

```
     */
    private static void maxHeapify(int[] arr, int index, int len) {
        int li = (index << 1) + 1; // 左子节点索引
        int ri = li + 1;           // 右子节点索引
        int cMax = li;             // 子节点值最大索引，默认左子节点。

        if (li > len) return;       // 左子节点索引超出计算范围，直接返回。
        if (ri <= len && arr[ri] > arr[li]) // 先判断左右子节点，哪个较大。
            cMax = ri;
        if (arr[cMax] > arr[index]) {
            swap(arr, cMax, index);      // 如果父节点被子节点调换，
            maxHeapify(arr, cMax, len);  // 则需要继续判断换下后的父节点是否符合堆的特性。
        }
    }
}
```