# 十大经典排序算法最强总结（含JAVA代码实现）

> **最近几天在研究排序算法，看了很多博客，发现网上有的文章中对排序算法解释的并不是很透彻，而且有很多代码都是错误的，例如有的文章中在"桶排序"算法中对每个桶进行排序直接使用了Collection.sort()函数，这样虽然能达到效果，但对于算法研究来讲是不可以的。所以我根据这几天看的文章，整理了一个较为完整的排序算法总结，本文中的所有算法均有JAVA实现，经本人调试无误后才发出，如有错误，请各位前辈指出。**

### 0、排序算法说明

#### 0.1 排序的定义

对一序列对象根据某个关键字进行排序。

#### 0.2 术语说明

- **稳定**：如果a原本在b前面，而a=b，排序之后a仍然在b的前面；
- **不稳定**：如果a原本在b的前面，而a=b，排序之后a可能会出现在b的后面；
- **内排序**：所有排序操作都在内存中完成；
- **外排序**：由于数据太大，因此把数据放在磁盘中，而排序通过磁盘和内存的数据传输才能进行；
- **时间复杂度**： 一个算法执行所耗费的时间。
- **空间复杂度**：运行完一个程序所需内存的大小。

#### 0.3 算法总结

| 排序算法 | 平均时间复杂度 | 最好情况 | 最坏情况 | 空间复杂度 | 排序方式 | 稳定性 |
|---|---|---|---|---|---|---|
| 冒泡排序 | $O(n^2)$ | $O(n)$ | $O(n^2)$ | $O(1)$ | In-place | 稳定 |
| 选择排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | In-place | 不稳定 |
| 插入排序 | $O(n^2)$ | $O(n)$ | $O(n^2)$ | $O(1)$ | In-place | 稳定 |
| 希尔排序 | $O(n \log n)$ | $O(n \log^2 n)$ | $O(n \log^2 n)$ | $O(1)$ | In-place | 不稳定 |
| 归并排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n)$ | Out-place | 稳定 |
| 快速排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$ | $O(\log n)$ | In-place | 不稳定 |
| 堆排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | In-place | 不稳定 |
| 计数排序 | $O(n + k)$ | $O(n + k)$ | $O(n + k)$ | $O(k)$ | Out-place | 稳定 |
| 桶排序 | $O(n + k)$ | $O(n + k)$ | $O(n^2)$ | $O(n + k)$ | Out-place | 稳定 |
| 基数排序 | $O(n \times k)$ | $O(n \times k)$ | $O(n \times k)$ | $O(n + k)$ | Out-place | 稳定 |

**图片名词解释：**

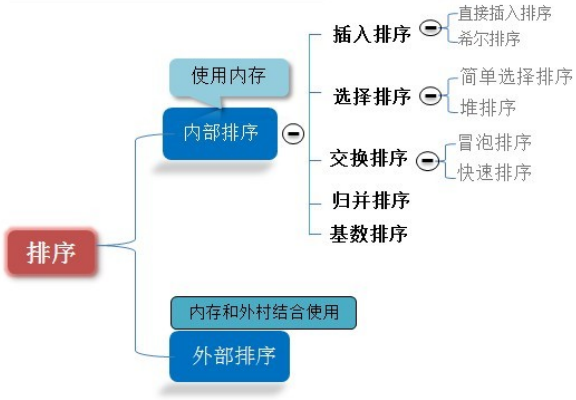
- n：数据规模
- k："桶"的个数
- In-place：占用常数内存，不占用额外内存
- Out-place：占用额外内存





昵称：郭耀华
园龄：3年2个月
粉丝：446
关注：2
+加关注

### 0.5 算法分类




### 0.6 比较和非比较的区别

常见的**快速排序、归并排序、堆排序、冒泡排序**等属于**比较排序**。**在排序的最终结果里，元素之间的次序依赖于它们之间的比较。每个数都必须和其他数进行比较，才能确定自己的位置。**

在**冒泡排序**之类的排序中，问题规模为n，又因为需要比较n次，所以平均时间复杂度为O(n²)。在**归并排序、快速排序**之类的排序中，问题规模通过**分治法**消减为logN次，所以时间复杂度平均**O(nlogn)**。

比较排序的优势是，适用于各种规模的数据，也不在乎数据的分布，都能进行排序。可以说，**比较排序适用于一切需要排序的情况。**

**计数排序、基数排序、桶排序**则属于**非比较排序**。非比较排序是通过确定每个元素之前，应该有多少个元素来排序。针对数组arr，计算arr[i]之前有多少个元素，则唯一确定了arr[i]在排序后数组中的位置。

非比较排序只要确定每个元素之前的已有的元素个数即可，所有一次遍历即可解决。算法时间复杂度**O(n)**。

**非比较排序时间复杂度底，但由于非比较排序需要占用空间来确定唯一位置。所以对数据规模和数据分布有一定的要求。**

---

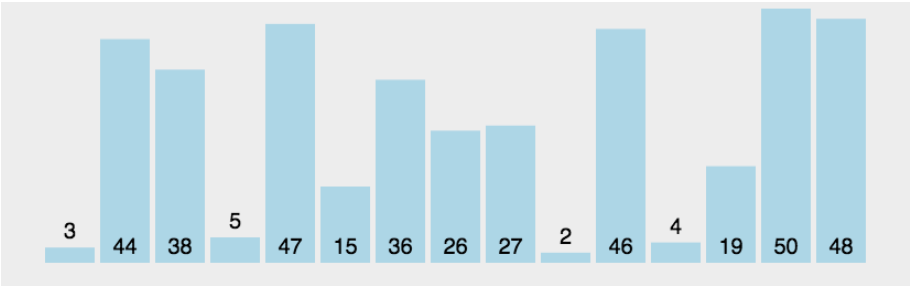
## 1、冒泡排序（Bubble Sort）

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，如果它们的顺序错误就把它们交换过来。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。

### 1.1 算法描述

- 比较相邻的元素。如果第一个比第二个大，就交换它们两个；
- 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，这样在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；
- 重复步骤1~3，直到排序完成。

### 1.2 动图演示




### 1.3 代码实现

```
1     /**
2      * 冒泡排序
3      *
4      * @param array
5      * @return
6      */
7     public static int[] bubbleSort(int[] array) {
8         if (array.length == 0)
9             return array;
10        for (int i = 0; i < array.length; i++)
11            for (int j = 0; j < array.length - 1 - i; j++)
12                if (array[j + 1] < array[j]) {
13                    int temp = array[j + 1];
```

```
                array[j + 1] = array[j];
                array[j] = temp;
            }
        return array;
    }
```

#### 1.4 算法分析

**最佳情况：T(n) = O(n) 最差情况：T(n) = O($n^2$) 平均情况：T(n) = O($n^2$)**

### 2、选择排序（Selection Sort）

表现**最稳定的排序算法之一**，因为**无论什么数据进去都是O(n2)的时间复杂度**，所以用到它的时候，数据规模越小越好。唯一的好处可能就是不占用额外的内存空间了吧。理论上讲，选择排序可能也是平时排序一般人想到的最多的排序方法了吧。

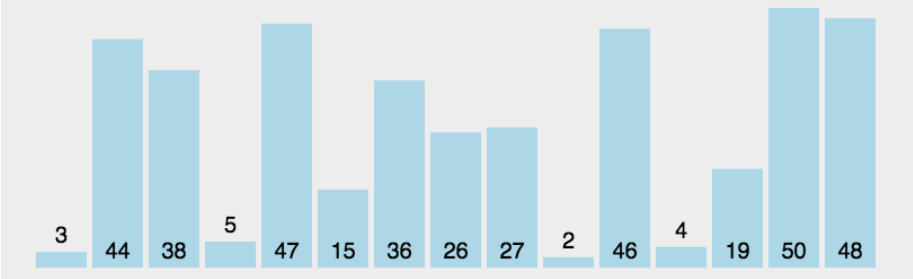
选择排序(Selection-sort)是一种简单直观的排序算法。它的工作原理：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

#### 2.1 算法描述

n个记录的直接选择排序可经过n-1趟直接选择排序得到有序结果。具体算法描述如下：

- 初始状态：无序区为R[1..n]，有序区为空；
- 第i趟排序(i=1,2,3…n-1)开始时，当前有序区和无序区分别为R[1..i-1]和R(i..n）。该趟排序从当前无序区中-选出关键字最小的记录 R[k]，将它与无序区的第1个记录R交换，使R[1..i]和R[i+1..n)分别变为记录个数增加1个的新有序区和记录个数减少1个的新无序区；
- n-1趟结束，数组有序化了。

#### 2.2 动图演示




#### 2.3 代码实现

```
/**
 * 选择排序
 * @param array
 * @return
 */
public static int[] selectionSort(int[] array) {
    if (array.length == 0)
        return array;
    for (int i = 0; i < array.length; i++) {
        int minIndex = i;
        for (int j = i; j < array.length; j++) {
            if (array[j] < array[minIndex]) //找到最小的数
                minIndex = j; //将最小数的索引保存
        }
        int temp = array[minIndex];
        array[minIndex] = array[i];
        array[i] = temp;
    }
    return array;
}
```

#### 2.4 算法分析

**最佳情况：T(n) = O($n^2$) 最差情况：T(n) = O($n^2$) 平均情况：T(n) = O($n^2$)**

### 3、插入排序（Insertion Sort）

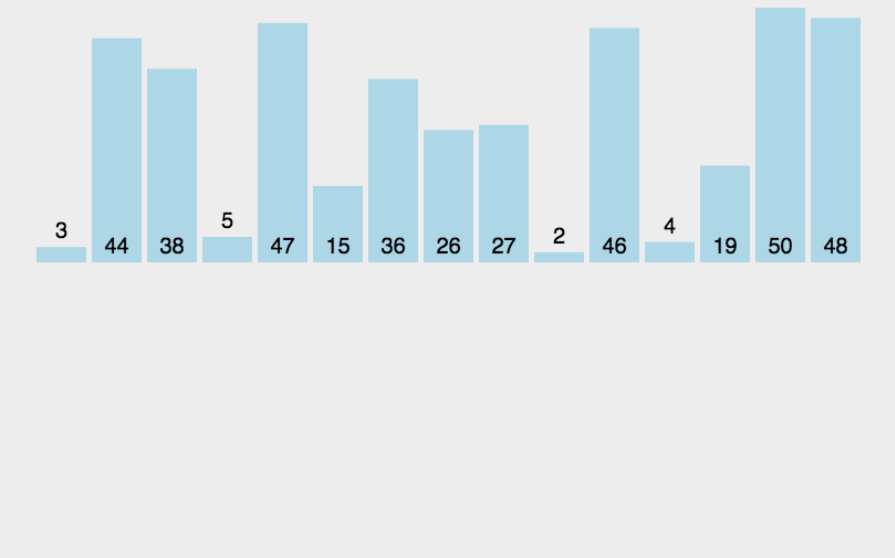
插入排序（Insertion-Sort）的算法描述是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。插入排序在实现上，通常采用in-place排序（即只需用到O(1)的额外空间的排序），因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

#### 3.1 算法描述

一般来说，插入排序都采用in-place在数组上实现。具体算法描述如下：

- 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后；
- 重复步骤2~5。

#### 3.2 动图演示




#### 3.2 代码实现

```
/**
 * 插入排序
 * @param array
 * @return
 */
public static int[] insertionSort(int[] array) {
    if (array.length == 0)
        return array;
    int current;
    for (int i = 0; i < array.length - 1; i++) {
        current = array[i + 1];
        int preIndex = i;
        while (preIndex >= 0 && current < array[preIndex]) {
            array[preIndex + 1] = array[preIndex];
            preIndex--;
        }
        array[preIndex + 1] = current;
    }
    return array;
}
```

#### 3.4 算法分析

**最佳情况：T(n) = O(n) 最坏情况：T(n) = O($n^2$) 平均情况：T(n) = O($n^2$)**

### 4、希尔排序（Shell Sort）

希尔排序是希尔（Donald Shell）于1959年提出的一种排序算法。希尔排序也是一种插入排序，它是简单插入排序经过改进之后的一个更高效的版本，也称为缩小增量排序，同时该算法是冲破O($n^2$）的第一批算法之一。它与插入排序的不同之处在于，它会优先比较距离较远的元素。希尔排序又叫缩小增量排序。

**希尔排序是把记录按下表的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰被分成一组，算法便终止。**

#### 4.1 算法描述

我们来看下希尔排序的基本步骤，在此我们选择增量gap=length/2，缩小增量继续以gap = gap/2的方式，这种增量选择我们可以用一个序列来表示，**{n/2,(n/2)/2...1}**，称为**增量序列**。希尔排序的增量序列的选择与证明是个数学难题，我们选择的这个增量序列是比较常用的，也是希尔建议的增量，称为希尔增量，但其实这个增量序列不是最优的。此处我们做示例使用希尔增量。

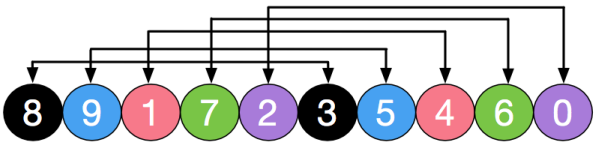
先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，具体算法描述：

- 选择一个增量序列t1，t2，…，tk，其中ti>tj，tk=1；
- 按增量序列个数k，对序列进行k 趟排序；
- 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。

#### 4.2 过程演示

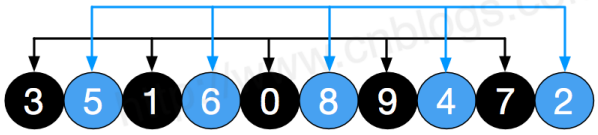
原始数组 以下数据元素颜色相同为一组


初始增量 gap=length/2=5，意味着整个数组被分为5组，[8,3] [9,5] [1,4] [7,6] [2,0]



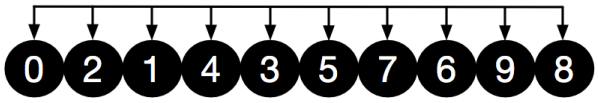

对这5组分别进行直接插入排序，结果如下，可以看到，像3，5，6这些小元素都被调到前面了，

然后缩小增量 gap=5/2=2,数组被分为2组 [3,1,0,9,7] [5,6,8,4,2]




对以上2组再分别进行直接插入排序，结果如下，可以看到，此时整个数组的有序程度更进一步啦。

再缩小增量gap=2/2=1，此时，整个数组为1组[0,2,1,4,3,5,7,6,9,8]，如下




经过上面的“宏观调控”，整个数组的有序化程度成果喜人。

此时，仅仅需要对以上数列简单微调，无需大量移动操作即可完成整个数组的排序。


#### 4.3 代码实现

```
/**
 * 希尔排序
 *
 * @param array
 * @return
```





4. Re:十大经典排序算
A代码实现）
你写的代码 冒泡排序能达

5. Re:【深度学习】深入
alization批标准化
数学公式完全不理解/捂脸

阅读排行榜

1. 【深度学习】深入理解
ation批标准化(210930)
2. 深度学习——优化器算
解（BGD、SGD、MBGD
m、NAG、Adagrad、A
op、Adam）(142460)
3. 十大经典排序算法最强
码实现）(129525)
4. NLP之——Word2Vec
5. Win10 Anaconda下
环境搭建详细教程（包含
装过程）(71901)
6. 【深度学习】一文读懂
失函数（Loss Function）
7. 【深度学习】目标检测
N、Fast R-CNN、Faste
YOLO、SSD、RetinaNe
8. 深度学习——卷积神经
（LeNet-5、AlexNet、
6、GoogLeNet、ResNe
9. 【深度学习】数据降维
9)
10. 【NLP】Attention
型）学习总结(1

```
     */
    public static int[] ShellSort(int[] array) {
        int len = array.length;
        int temp, gap = len / 2;
        while (gap > 0) {
            for (int i = gap; i < len; i++) {
                temp = array[i];
                int preIndex = i - gap;
                while (preIndex >= 0 && array[preIndex] > temp) {
                    array[preIndex + gap] = array[preIndex];
                    preIndex -= gap;
                }
                array[preIndex + gap] = temp;
            }
            gap /= 2;
        }
        return array;
    }
```

#### 4.4 算法分析

**最佳情况：T(n) = O(nlog$_2$ n) 最坏情况：T(n) = O(nlog$_2$ n) 平均情况：T(n) =O(nlog$_2$n)**

### 5、归并排序（Merge Sort）

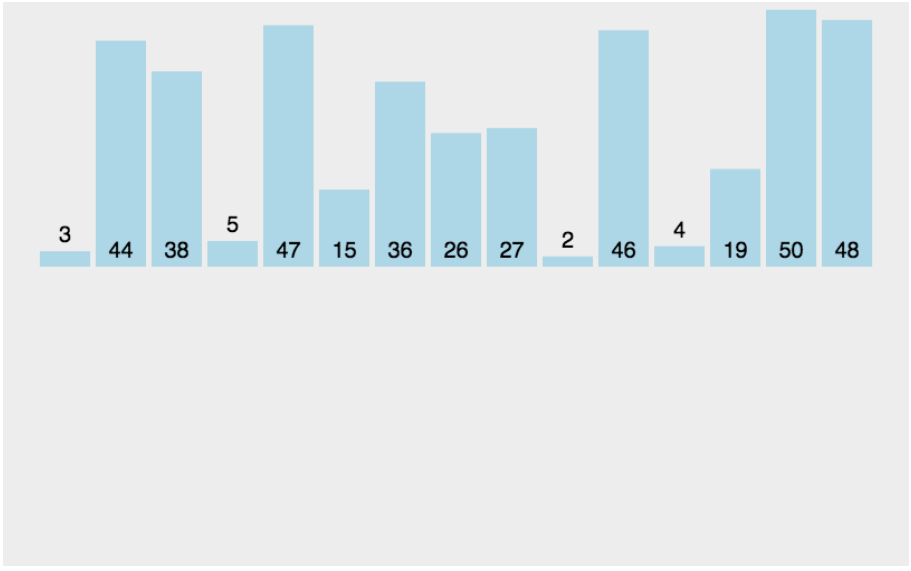
和选择排序一样，归并排序的性能不受输入数据的影响，但表现比选择排序好的多，因为始终都是O(n log n）的时间复杂度。代价是需要额外的内存空间。

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。归并排序是一种稳定的排序方法。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为2-路归并。

#### 5.1 算法描述

- 把长度为n的输入序列分成两个长度为n/2的子序列；
- 对这两个子序列分别采用归并排序；
- 将两个排序好的子序列合并成一个最终的排序序列。

#### 5.2 动图演示




#### 5.3 代码实现

```
    /**
     * 归并排序
     *
     * @param array
     * @return
     */
    public static int[] MergeSort(int[] array) {
        if (array.length < 2) return array;
        int mid = array.length / 2;
```





评论排行榜
1. 十大经典排序算法最强总结（含JAVA代码实现）(48)
2. 【深度学习】深入理解Batch Normalization批标准化(22)
3. 深度学习——优化器算法总结（BGD、SGD、MBGD、Momentum、NAG、Adagrad、Adadelta、RMSprop、Adam）(8)
4. NLP之——Word2Vec
5. 【NLP】Attention Model（注意力模型）学习总结(3)
6. Win10 Anaconda下TensorFlow环境搭建详细教程（包含安装过程）(3)
7. 【深度学习】目标检测：R-CNN、Fast R-CNN、Faster R-CNN、YOLO、SSD、RetinaNet
8. SmileyFace——基于人脸表情、眼检测、面部识别程序(3
9. 深度学习工作站攒机指
10. 平均精度均值(mAP)：目标检测模型性能统计量(2)
推荐排行榜
1. 十大经典排序算法最强总结（含JAVA代码实现）(387)
2. 【深度学习】深入理解Batch Normalization批标准化(263)
3. 深度学习——优化器算法总结（BGD、SGD、MBGD、Momentum、NAG、Adagrad、Adadelta、RMSprop、Adam）(108)
4. NLP之——Word2Vec
5. 【深度学习】目标检测：R-CNN、Fast R-CNN、Faster R-CNN、YOLO、SSD、

```
        int[] left = Arrays.copyOfRange(array, 0, mid);
        int[] right = Arrays.copyOfRange(array, mid, array.length);
        return merge(MergeSort(left), MergeSort(right));
    }
    /**
     * 归并排序—将两段排序好的数组结合成一个排序数组
     *
     * @param left
     * @param right
     * @return
     */
    public static int[] merge(int[] left, int[] right) {
        int[] result = new int[left.length + right.length];
        for (int index = 0, i = 0, j = 0; index < result.length; index++) {
            if (i >= left.length)
                result[index] = right[j++];
            else if (j >= right.length)
                result[index] = left[i++];
            else if (left[i] > right[j])
                result[index] = right[j++];
            else
                result[index] = left[i++];
        }
        return result;
    }
```

#### 5. 4 算法分析

**最佳情况：T(n) = O(n) 最差情况：T(n) = O(nlogn) 平均情况：T(n) = O(nlogn)**

### 6、快速排序（Quick Sort）

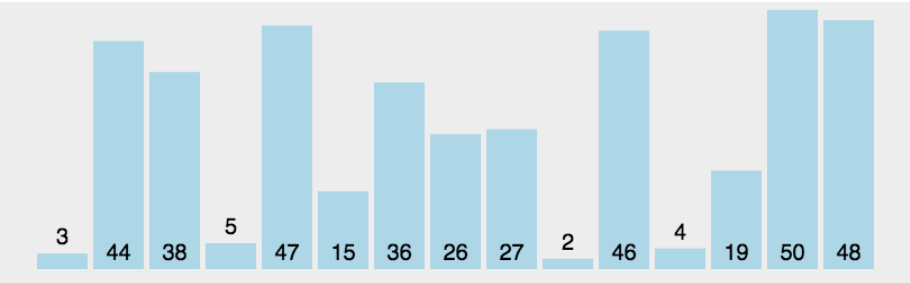
快速排序的基本思想：通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

#### 6.1 算法描述

快速排序使用分治法来把一个串（list）分为两个子串（sub-lists）。具体算法描述如下：

- 从数列中挑出一个元素，称为 “基准”（**pivot**）；
- 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
- 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

#### 5.2 动图演示




#### 5.3 代码实现

```
    /**
     * 快速排序方法
     * @param array
     * @param start
     * @param end
     * @return
     */
    public static int[] QuickSort(int[] array, int start, int end) {
        if (array.length < 1 || start < 0 || end >= array.length || start > end) return null;
        int smallIndex = partition(array, start, end);
        if (smallIndex > start)
            QuickSort(array, start, smallIndex - 1);
        if (smallIndex < end)
```

```
            QuickSort(array, smallIndex + 1, end);
        return array;
    }
    /**
     * 快速排序算法——partition
     * @param array
     * @param start
     * @param end
     * @return
     */
    public static int partition(int[] array, int start, int end) {
        int pivot = (int) (start + Math.random() * (end - start + 1));
        int smallIndex = start - 1;
        swap(array, pivot, end);
        for (int i = start; i <= end; i++)
            if (array[i] <= array[end]) {
                smallIndex++;
                if (i > smallIndex)
                    swap(array, i, smallIndex);
            }
        return smallIndex;
    }

    /**
     * 交换数组内两个元素
     * @param array
     * @param i
     * @param j
     */
    public static void swap(int[] array, int i, int j) {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
```

#### 5.4 算法分析

**最佳情况：T(n) = O(nlogn)　最差情况：T(n) = O($n^2$)　平均情况：T(n) = O(nlogn)**

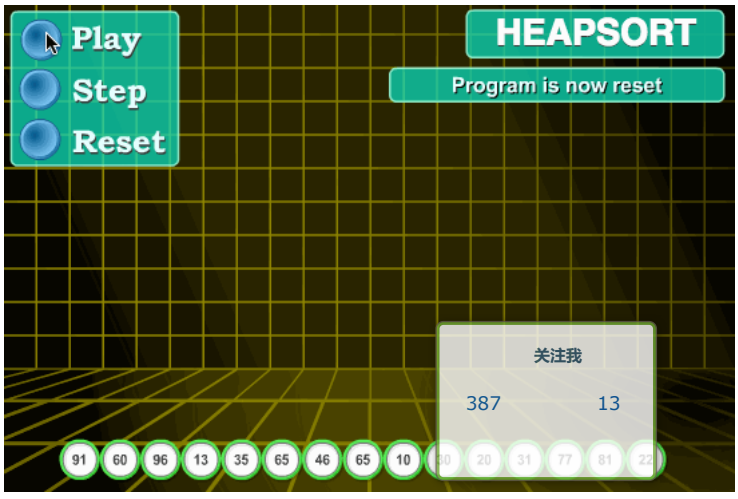
### 7、堆排序（Heap Sort）

堆排序（Heapsort）是指利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。

#### 7.1 算法描述

- 将初始待排序关键字序列(R1,R2….Rn)构建成大顶堆，此堆为初始的无序区；
- 将堆顶元素R[1]与最后一个元素R[n]交换，此时得到新的无序区(R1,R2,……Rn-1)和新的有序区(Rn),且满足R[1,2…n-1] <=R[n]；
- 由于交换后新的堆顶R[1]可能违反堆的性质，因此需要对当前无序区(R1,R2,……Rn-1)调整为新堆，然后再次将R[1]与无序区最后一个元素交换，得到新的无序区(R1,R2….Rn-2)和新的有序区(Rn-1,Rn)。不断重复此过程直到有序区的元素个数为n-1，则整个排序过程完成。

#### 7.2 动图演示



### 7.3 代码实现

注意：这里用到了完全二叉树的部分性质：详情见《数据结构二叉树知识点总结》

```
//声明全局变量，用于记录数组array的长度；
static int len;
    /**
     * 堆排序算法
     *
     * @param array
     * @return
     */
    public static int[] HeapSort(int[] array) {
        len = array.length;
        if (len < 1) return array;
        //1.构建一个最大堆
        buildMaxHeap(array);
        //2.循环将堆首位（最大值）与末位交换，然后在重新调整最大堆
        while (len > 0) {
            swap(array, 0, len - 1);
            len--;
            adjustHeap(array, 0);
        }
        return array;
    }
    /**
     * 建立最大堆
     *
     * @param array
     */
    public static void buildMaxHeap(int[] array) {
        //从最后一个非叶子节点开始向上构造最大堆
        for (int i = (len/2 - 1); i >= 0; i--) { //感谢 @让我发会呆 网友的提醒，此处应该为 i = (len/2 - 1)
            adjustHeap(array, i);
        }
    }
    /**
     * 调整使之成为最大堆
     *
     * @param array
     * @param i
     */
    public static void adjustHeap(int[] array, int i) {
        int maxIndex = i;
        //如果有左子树，且左子树大于父节点，则将最大指针指向左子树
        if (i * 2 < len && array[i * 2] > array[maxIndex])
            maxIndex = i * 2;
        //如果有右子树，且右子树大于父节点，则将最大指针指向右子树
        if (i * 2 + 1 < len && array[i * 2 + 1] > array[maxIndex])
            maxIndex = i * 2 + 1;
        //如果父节点不是最大值，则将父节点与最大值交换，并且递归调整与父节点交换的位置。
        if (maxIndex != i) {
            swap(array, maxIndex, i);
            adjustHeap(array, maxIndex);
        }
    }
```

### 7.4 算法分析

**最佳情况：T(n) = O(nlogn) 最差情况：T(n) = O(nlogn) 平均情况：T(n) = O(nlogn)**

## 8、计数排序（Counting Sort）

计数排序的核心在于将输入的数据值转化为键存储在额外开辟的数组空间中。 作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数。

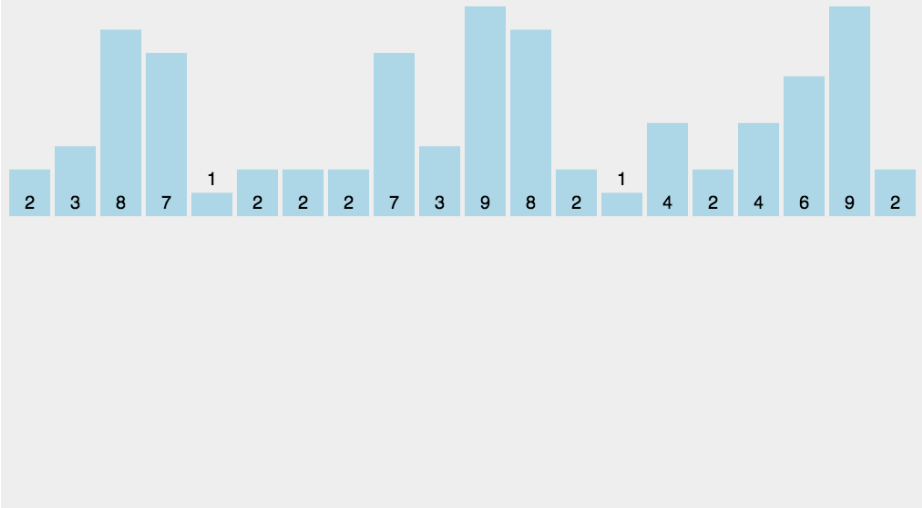
计数排序(Counting sort)是一种稳定的排序算法。计数排序使用一个额外的数组C，其中第i个元素是待排序数组A中值等于i的元素的个数。然后根据数组C来将A中的元素排到正确的位置。它只能对整数进行排序。



### 8.1 算法描述

- 找出待排序的数组中最大和最小的元素；
- 统计数组中每个值为i的元素出现的次数，存入数组C的第i项；
- 对所有的计数累加（从C中的第一个元素开始，每一项和前一项相加）；
- 反向填充目标数组：将每个元素i放在新数组的第C(i)项，每放一个元素就将C(i)减去1。

### 8.2 动图演示




### 8.3 代码实现

```
/**
     * 计数排序
     *
     * @param array
     * @return
     */
    public static int[] CountingSort(int[] array) {
        if (array.length == 0) return array;
        int bias, min = array[0], max = array[0];
        for (int i = 1; i < array.length; i++) {
            if (array[i] > max)
                max = array[i];
            if (array[i] < min)
                min = array[i];
        }
        bias = 0 - min;
        int[] bucket = new int[max - min + 1];
        Arrays.fill(bucket, 0);
        for (int i = 0; i < array.length; i++) {
            bucket[array[i] + bias]++;
        }
        int index = 0, i = 0;
        while (index < array.length) {
            if (bucket[i] != 0) {
                array[index] = i - bias;
                bucket[i]--;
                index++;
            } else
                i++;
        }
        return array;
    }
```

### 8.4 算法分析

当输入的元素是n 个0到k之间的整数时，它的运行时间是 O(n + k)。计数排序不是比较排序，排序的速度快于任何比较排序算法。由于用来计数的数组C的长度取决于待排序数组中数据的范围（等于待排序数组的最大值与最小值的差加上1），这使得计数排序对于数据范围很大的数组，需要大量时间和内存。

**最佳情况：T(n) = O(n+k) 最差情况：T(n) = O(n+k) 平均情况：T(n) = O(n+k)**



## 9、桶排序（Bucket Sort）

桶排序是计数排序的升级版。它利用了函数的映射关系，高效与否的关键就在于这个映射函数的确定。

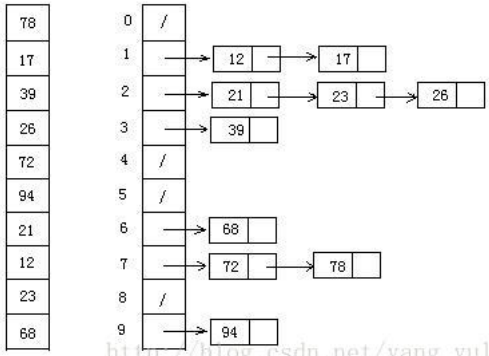
桶排序 (Bucket sort)的工作的原理：假设输入数据服从均匀分布，将数据分到有限数量的桶里，每个桶再分别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排

**9.1 算法描述**

- 人为设置一个BucketSize，作为每个桶所能放置多少个不同数值（例如当BucketSize==5时，该桶可以存放 {1,2,3,4,5} 这几种数字，但是容量不限，即可以存放100个3）；
- 遍历输入数据，并且把数据一个一个放到对应的桶里去；
- 对每个不是空的桶进行排序，可以使用其它排序方法，也可以递归使用桶排序；
- 从不是空的桶里把排好序的数据拼接起来。

**注意，如果递归使用桶排序为各个桶排序，则当桶数量为1时要手动减小BucketSize增加下一循环桶的数量，否则会陷入死循环，导致内存溢出。**

**9.2 图片演示**





**9.3 代码实现**

```
/**
 * 桶排序
 *
 * @param array
 * @param bucketSize
 * @return
 */
public static ArrayList<Integer> BucketSort(ArrayList<Integer> array, int bucketSize) {
    if (array == null || array.size() < 2)
        return array;
    int max = array.get(0), min = array.get(0);
    // 找到最大值最小值
    for (int i = 0; i < array.size(); i++) {
        if (array.get(i) > max)
            max = array.get(i);
        if (array.get(i) < min)
            min = array.get(i);
    }
    int bucketCount = (max - min) / bucketSize + 1;
    ArrayList<ArrayList<Integer>> bucketArr = new ArrayList<>(bucketCount);
    ArrayList<Integer> resultArr = new ArrayList<>();
    for (int i = 0; i < bucketCount; i++) {
        bucketArr.add(new ArrayList<Integer>());
    }
    for (int i = 0; i < array.size(); i++) {
        bucketArr.get((array.get(i) - min) / bucketSize).add(array.get(i));
    }
    for (int i = 0; i < bucketCount; i++) {
        if (bucketSize == 1) { // 如果带排序数组中有重复数字时  感谢 @见风任然是风 朋友指出错误
            for (int j = 0; j < bucketArr.get(i).size(); j++)
                resultArr.add(bucketArr.get(i).get(j));
        } else {
            if (bucketCount == 1)
                bucketSize--;
            ArrayList<Integer> temp = BucketSort(bucketArr.get(i), bucketSize);
            for (int j = 0; j < temp.size(); j++)
                resultArr.add(temp.get(j));
        }
    }
```

```
        return resultArr;
    }
```

#### 9.4 算法分析

桶排序最好情况下使用线性时间O(n)，桶排序的时间复杂度，取决与对各个桶之间数据进行排序的时间复杂度，因为其它部分的时间复杂度都为O(n)。很显然，桶划分的越小，各个桶之间的数据越少，排序所用的时间也会越少。但相应的空间消耗就会增大。

**最佳情况：T(n) = O(n+k) 最差情况：T(n) = O(n+k) 平均情况：T(n) = O($n^2$)**

### 10、基数排序（Radix Sort）

基数排序也是非比较的排序算法，对每一位进行排序，从最低位开始排序，复杂度为O(kn),为数组长度，k为数组中的数的最大的位数；

基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。基数排序基于分别排序，分别收集，所以是稳定的。

#### 10.1 算法描述

- 取得数组中的最大数，并取得位数；
- arr为原始数组，从最低位开始取每个位组成radix数组；
- 对radix进行计数排序（利用计数排序适用于小范围数的特点）；

#### 10.2 动图演示


#### 10.3 代码实现

```
/**
 * 基数排序
 * @param array
 * @return
 */
public static int[] RadixSort(int[] array) {
    if (array == null || array.length < 2)
        return array;
    // 1.先算出最大数的位数;
    int max = array[0];
    for (int i = 1; i < array.length; i++) {
        max = Math.max(max, array[i]);
    }
    int maxDigit = 0;
    while (max != 0) {
        max /= 10;
        maxDigit++;
    }
    int mod = 10, div = 1;
    ArrayList<ArrayList<Integer>> bucketList = new ArrayList<ArrayList<Integer>>();
    for (int i = 0; i < 10; i++)
        bucketList.add(new ArrayList<Integer>());
    for (int i = 0; i < maxDigit; i++, mod *= 10, div *= 10) {
        for (int j = 0; j < array.length; j++) {
```

```
                int num = (array[j] % mod) / div;
                bucketList.get(num).add(array[j]);
            }
            int index = 0;
            for (int j = 0; j < bucketList.size(); j++) {
                for (int k = 0; k < bucketList.get(j).size(); k++)
                    array[index++] = bucketList.get(j).get(k);
                bucketList.get(j).clear();
            }
        }
        return array;
    }
```

#### 10.4 算法分析

**最佳情况：T(n) = O(n * k)　最差情况：T(n) = O(n * k)　平均情况：T(n) = O(n * k)**

基数排序有两种方法：

MSD 从高位开始进行排序 LSD 从低位开始进行排序

#### 基数排序 vs 计数排序 vs 桶排序

这三种排序算法都利用了桶的概念，但对桶的使用方法上有明显差异：

- 基数排序：根据键值的每位数字来分配桶
- 计数排序：每个桶只存储单一键值
- 桶排序：每个桶存储一定范围的数值




分类： JAVA

« 上一篇： 数据结构二叉树知识点总结
» 下一篇： 面试历程记录

posted @ 2018-03-19 10:14 郭耀华 阅读(129544) 评论(48) 编辑 收藏



评论列表

#1楼 2018-03-19 14:07 ITDragon龙

已推荐，相对于算法的魅力，我更在意这图是怎么做出来的。

支持(16) 反对(0)

#2楼 2018-03-19 14:10 Sam Xiao

你这个动图还真有技术。比这个算法的技术要高，动态的效果过度的很平滑。

支持(5) 反对(0)

#3楼 2018-03-20 09:48 我是13

赞

支持(1) 反对(0)

#4楼 2018-03-20 11:25 wdwwtzy

不错 挺有意思

支持(1) 反对(0)

#5楼 2018-03-20 13:47 穷苦书生

动态图666

支持(1) 反对(0)

#6楼 2018-03-20 17:38 影子の爱人

@ ITDragon龙

引用

已推荐，相对于算法的魅力，我更在意这图是怎么做出来的。

同求

支持(0) 反对(0)

#7楼 2018-03-22 22:14 Python学习者

mark

支持(0) 反对(0)

#8楼 2018-03-23 14:09 不知道该说啥

动图怎么做出来的?

支持(0) 反对(0)

#9楼 2018-03-28 13:44 弗洛伊德77

动图，挺不错的

支持(0) 反对(0)

#10楼 2018-03-29 16:55 见风任然是风

```
ArrayList<Integer> arr=new ArrayList<Integer>();
        arr.add(19);
        arr.add(27);
        arr.add(11);
        arr.add(37);
        arr.add(14);
```

```
        arr.add(27);
System.out.println(BucketSort(arr,5));

//桶排序报错
```

支持(0) 反对(0)

#11楼 2018-04-04 15:49 让我发会呆

在堆排序中，建立最大堆的方法里，for (int i = (len/2- 1); i >= 0; i--) {},
for循环这样写会不会更好一点，i的左子树和右子树分别2i+1和2(i+1)。

支持(0) 反对(0)

#12楼 [楼主] 2018-04-05 23:19 郭耀华

@ 让我发会呆
你说的对，是我的失误~谢谢提醒

支持(0) 反对(0)

#13楼 [楼主] 2018-04-05 23:34 郭耀华

@ 见风任然是风
确实是我写的有问题，已修正~感谢指出。

```
/**
 * 桶排序
 *
 * @param array
 * @param bucketSize
 * @return
 */
public static ArrayList<Integer> BucketSort(ArrayList<Integer> array, int bucketSize) {
    if (array == null || array.size() < 2)
        return array;
    int max = array.get(0), min = array.get(0);
    // 找到最大值最小值
    for (int i = 0; i < array.size(); i++) {
        if (array.get(i) > max)
            max = array.get(i);
        if (array.get(i) < min)
            min = array.get(i);
    }
    int bucketCount = (max - min) / bucketSize + 1;
    ArrayList<ArrayList<Integer>> bucketArr = new ArrayList<>(bucketCount);
    ArrayList<Integer> resultArr = new ArrayList<>();
    for (int i = 0; i < bucketCount; i++) {
        bucketArr.add(new ArrayList<Integer>());
    }
    for (int i = 0; i < array.size(); i++) {
        bucketArr.get((array.get(i) - min) / bucketSize).add(array.get(i));
    }
    for (int i = 0; i < bucketCount; i++) {
        if (bucketSize == 1) { // 如果带排序数组中有重复数字时
            for (int j = 0; j < bucketArr.get(i).size(); j++)
                resultArr.add(bucketArr.get(i).get(j));
        } else {
            if (bucketCount == 1)
                bucketSize--;
            ArrayList<Integer> temp = BucketSort(bucketArr.get(i), bucketSize);
            for (int j = 0; j < temp.size(); j++)
                resultArr.add(temp.get(j));
        }
    }
    return resultArr;
}
```

支持(2) 反对(0)

#14楼 2018-04-26 23:23 tp_16b

为动图点赞~ 哈哈

支持(1) 反对(0)

#15楼 2018-05-10 13:22 qwy156

堆排序 把i = (len/2 - 1)改了，后面的逻辑是不是也该改一下啊。

支持(0) 反对(0)

#16楼 2018-06-27 22:43 蓝鲸也是鲸

给例子 能不能给全啊

支持(0) 反对(0)

#17楼 [楼主] 2018-06-28 11:08 郭耀华

@ 蓝鲸也是鲸
请问哪里不全?

支持(0) 反对(0)

#18楼 2018-11-30 15:08 沟渠映明月

图nice

支持(0) 反对(0)

#19楼 2019-03-17 10:10 lyx22

楼主你的堆排序那里的左子树的坐标是不是有问题，应该是i*2+1吧?

支持(2) 反对(0)

#20楼 2019-03-27 08:36 停也蹉跎0.0行也颠簸

归并排序代码不对

支持(1) 反对(0)

#21楼 2019-04-01 15:55 浮生尽头皆小町

堆排序中，左子树与右子树的坐标不对吧。。。。应该是i*2+1和i*2+2吧

支持(0) 反对(0)

#22楼 2019-04-08 00:51 lichaoyu

niuniu

支持(0) 反对(0)

#23楼 2019-05-25 21:49 Gordon_running

快速排序 基数的生成，以及下面的 for 循环能够加一个注释么，没怎么看懂，其他的还在慢慢看，希望 博主 能够 解答一下下，Thanks♪(･ω･)ﾉ

支持(0) 反对(0)

#24楼 [楼主] 2019-05-25 21:57 郭耀华

@ Gordon_running
快排算法的描述中已经做出了解释:
1.从数列中挑出一个元素，称为 “基准”（pivot）；
2.重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作;

```
// 这个就是选取基准（注意：pivot指的是index，可理解为指针），在start和end两个index之间随机生成一个数。
int pivot = (int) (start + Math.random() * (end - start + 1));

下面for循环的作用就是将start和end这个区间的元素按照pivot指针指向的值，将小于pivot元素值的放置在pivot左边，大于的放在右边。
```

支持(0) 反对(0)



#25楼 2019-05-25 22:02 Gordon_running

@ 郭耀华
谢谢博主的 真诚回答!

不过，我还是有些疑惑的地方，就是下面这些代码部分，希望博主能够看两分钟，给点注释，我拿着笔和纸，按照代码手动模拟一遍，卡到这个地方了。

就是下面的代码部分：

```
for (int i = start; i <= end; i++)
            if (array[i] <= array[end]) {
                smallIndex++;
                if (i > smallIndex)
                    swap(array, i, smallIndex);
            }
```

支持(0) 反对(0)

#26楼 2019-05-25 23:43 Gordon_running

@ 郭耀华
博主，你好，我又用一组 数据 重新走了一遍流程，这个真的是巧妙，不过理解起来确实挺困难的。现在已经理解了，这一段代码是没问题的。是我理解的方式不正确。

支持(0) 反对(0)

#27楼 2019-05-26 22:28 Gordon_running

博主，你好，
对于堆排序，下标可以从0 开始，对于 中间某个节点坐标为 i ，
那么 它 的左子树 下标为 2*i +1 ， 它的右子树 下标为 2*i+2，

对于堆排序，下标如果从1 开始，对于中间某个节点坐标为 i ，
那么 它 的左子树 下标为 2*i ， 它 的右子树 下标为 2*i+1

你的堆排序的 maxHeap（）中，i 可以从 n/2,开始，即堆下标从1 开始 ，
adjustHeap 中下标不用变，
如果堆排序中 maxHeap（）中， i 从 n/2-1 开始， 即对下表从0 开始，那么
adjustHeap 中的左子树下标需要改为 2*i +1，右子树下标需要改为 2*i +2

代码分别如下：
1，下标从 1 开始的代码：

```
+ View Code
```

2，下标从 0 开始 的代码：

```
+ View Code
```

支持(1) 反对(1)

#28楼 2019-06-01 10:11 Gordon_running

https://github.com/hustcc/JS-Sorting-Algorithm
https://www.runoob.com/w3cnote/ten-sorting-algorithm.html
看这两篇博客，跟咱们这个是非常像，不知道是不是他们抄咱们这里的┬┬﹏┬┬

支持(0) 反对(0)

#29楼 2019-07-04 09:18 KG5

选择排序算法修改为如下，for 可以少循环几次。其时就是自己不用跟自己比较。

```
public void selectionSort(int[] array) {
for (int i = 0; i < array.length; i++) {
int index = i;
for (int j = i + 1; j < array.length; j++) {
```

```
if (array[j] < array[index]) {
index = j;
}
}
int min = array[index];
array[index] = array[i];
array[i] = min;
}
}
```

支持(1) 反对(0)

#30楼 2019-08-13 14:32 晚英

好棒!

支持(1) 反对(0)

#31楼 2019-08-23 11:11 JoUU

牛逼!

支持(0) 反对(0)

#32楼 2019-08-27 17:51 天朝攻城狮

冒泡排序中有个小问题，缺少判断排序完成的标识，这个会导致算法最快的时间复杂度退化成n2

支持(1) 反对(0)

#33楼 2019-09-20 15:21 hello_415600

@ Gordon_running
世界很精彩，楼主：快速排序， 这个代码 貌似右问题啊。
下面for循环的作用就是将start和end这个区间的元素按照pivot指针指向的值，将小于pivot元素值的放置在pivot左边，大于的放在右边。 ？ 这个能做到？？

```
for (int i = start; i <= end; i++)
if (array[i] <= array[end]) {
smallIndex++;
if (i > smallIndex)
swap(array, i, smallIndex);
}
```

支持(1) 反对(0)

#34楼 2019-09-28 17:18 boaz问心

@ Gordon_running

```
if(2*i+1 <len && array[maxIndex] < array[2*i+1])
maxIndex = 2*i+1;
if(2*i+2 <len && array[maxIndex] < array[2*i+2])
maxIndex = 2*i+2;
```

这个地方评论里面是正确的
但是原文中是错的，原文是

```
if(2*i <len && array[maxIndex] < array[2*i])
maxIndex = 2*i;
if(2*i+2 <len && array[maxIndex] < array[2*i+2])
maxIndex = 2*i+2;
```

望能改正

支持(0) 反对(0)

#35楼 2019-09-28 17:20 boaz问心

```
if(2*i+1 <len && array[maxIndex] < array[2*i+1])
maxIndex = 2*i+1;
if(2*i+2 <len && array[maxIndex] < array[2*i+2])
maxIndex = 2*i+2;
```

这个地方评论里面是正确的
但是原文中是错的，原文是
if(2*i <len && array[maxIndex] < array[2*i])
maxIndex = 2*i;
if(2*i+1 <len && array[maxIndex] < array[2*i+1])
maxIndex = 2*i+1;
望能改正

支持(1) 反对(0)

#36楼 2019-10-10 16:55 dlxzy

博主，归并排序和桶排序有问题吧？我这边跑下来似乎没有排序啊！你能给我个完整的例子试试吗？

支持(1) 反对(0)

#37楼 2019-10-21 18:47 风清夏至

堆排序的adjustHeap方法中，左子树的下标应该是i*2+1,右子树的下标是i*2+2（因为结合上下文，下标是从0开始的），即：

if(i * 2 + 1 < len && array[i * 2 + 1] > array[maxIndex])

maxIndex = i * 2 + 1;

if(i * 2 + 2 < len && array[i * 2 + 2] > array[maxIndex])

maxIndex = i * 2 + 2;

支持(0) 反对(0)

#38楼 2019-11-14 19:00 我喜欢帽子123

选择排序的j为什么不从i+1开始呢，从i开始，自己和自己比较好像没有必要，求指教！

支持(1) 反对(0)

#39楼 2019-12-23 17:26 代码拯救不了世界

为什么开始的图片上写选择排序的稳定性是不稳定，但是在下面的介绍中却说选择排序是最稳定的排序之一呢？

支持(0) 反对(0)

#40楼 2019-12-24 09:33 代码拯救不了世界

@ hello_415600
楼主的这个快速排序就是有问题的

支持(0) 反对(0)

#41楼 2020-04-11 15:40 安然黑妮兔

very good

支持(0) 反对(0)

#42楼 2020-05-29 18:27 Damonon

楼主快排有问题，缺少了交换基准的步骤
smallIndex++;
swap(smallIndex, end);

支持(0) 反对(0)

#43楼 2020-06-03 14:19 LemonTreeKjy

归并排序代码错了

支持(1) 反对(0)

#44楼 2020-06-11 17:07 rorooo

博主，您的归并算法这么写，空间复杂度太高了，O(n^2)了吧。归并算法空间复杂度应该是O(n)才对。是不是应该写成：

```
    private static int[] mergeSort2(int[] arr) {
        return mergeSort(arr, 0, arr.length - 1);
```

```
    }

    private static int[] mergeSort(int[] arr, int from, int to) {
        if ((to - from) < 1)
            return arr;
        int mid = (from + to) / 2;
        mergeSort(arr, from, mid);
        mergeSort(arr, mid + 1, to);
        return merge2(arr, from, mid, mid + 1, to);
    }

    private static int[] merge2(int[] arr, int lfrom, int lto, int rfrom, int rto) {
        int li = lfrom, ri = rfrom;
        int[] res = new int[rto + 1];
        while (li <= lto || ri <= rto) {
            res[li + ri - rfrom] = li <= lto && (ri > rto || arr[li] < arr[ri]) ? arr[li++] : arr[ri++];
        }
        System.arraycopy(res, lfrom, arr, lfrom, rto - lfrom + 1);
        return arr;
    }
`
```

支持(0) 反对(0)

#45楼 2020-06-20 00:27 面朝大海春暖坏开

if (bucketCount == 1)等于1说明数据都是相差值小于bucketSize的（并不一定相同），无需处理，
需要考虑该桶是否有数据
bucketSize--;这一句不应该放在bucketCount == 1条件下吧?
而且桶排序中这样写没啥效果（该桶位里的数据就是很相似的）会浪费内存
bucketSize--;
不如写成bucketSize /= 2;

支持(0) 反对(0)

#46楼 2020-06-20 18:58 面朝大海春暖坏开

@郭耀华
堆排序中使用len/2- 1会导致第一次创建大顶堆的时候最后一个子树首次未调整，但是之后调整的时候会默认该子树已调整（存在该子树顺序未调整风险），除非swap交换的时候作比较，第一次交换时如果堆顶大于末尾再交换，或者直接使用len/2即可

支持(0) 反对(0)

#47楼 2020-08-10 11:14 一缕月光

你写的代码 冒泡排序能达到o(n) 吗

支持(0) 反对(0)

#48楼 2020-08-11 09:26 一缕月光

# 你写的代码 冒泡排序能达到o(n) 吗

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

**注册用户登录后才能发表评论，请 登录 或 注册， 访问 网站首页。**