# 十大排序算法代码：

## 冒泡排序

**package** 冒泡排序;

以从小到大的排序举例：

每相邻两个数据进行两两比较，始终使小的数在前面

/\*\*

\* 冒牌排序算法详解

\*

\* **@author** LaoK 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。

\*/

**public** **class** BubbleSort {

**public** **static** **void** main(String[] args) {

**int** arr[] = { 95, 45, 52, 12, 37, 68, 59 };

System.***out***.println("排序结果为：");

**for** (**int** i : *bubbleSort*(arr)) {

System.***out***.print(i + " ,");

}

}

// 从小到大

**public** **static** **int**[] bubbleSort(**int** arr[]) {

**int** count = 0;

**for** (**int** i = 0; i < arr.length - 1; i++) {

**for** (**int** j = 0; j < arr.length - 1; j++) {

**if** (arr[j] > arr[j + 1]) {

**int** temp = arr[j]; // 相邻元素相互交换

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

count++;

}

}

System.***out***.println("系统运行次数：" + count);

**return** arr;

}

}

## 快速排序

选择一个数（一般就是第一个数）作为标准，和其他的数进行比较，比他小的放在他的左边，比他大的放在他的右边，然后在进行同样的排序。

**package** 快速排序;

**public** **class** QuickSort {

**public** **static** **void** quickSort(**int** arr[], **int** \_left, **int** \_right) {

**int** left = \_left;

**int** right = \_right;

**int** temp = 0;

**if** (left <= right) { // 待排序的元素至少有两个的情况

temp = arr[left]; // 待排序的第一个元素作为基准元素

**while** (left != right) { // 从左右两边交替扫描，直到left = right

**while** (right > left && arr[right] >= temp)

right--; // 从右往左扫描，找到第一个比基准元素小的元素

arr[left] = arr[right]; // 找到这种元素arr[right]后与arr[left]交换

**while** (left < right && arr[left] <= temp)

left++; // 从左往右扫描，找到第一个比基准元素大的元素

arr[right] = arr[left]; // 找到这种元素arr[left]后，与arr[right]交换

}

arr[right] = temp; // 基准元素归位

*quickSort*(arr, \_left, left - 1); // 对基准元素左边的元素进行递归排序

*quickSort*(arr, right + 1, \_right); // 对基准元素右边的进行递归排序

}

}

**public** **static** **void** main(String[] args) {

**int** array[] = { 10, 5, 3, 1, 7, 2, 8 };

System.***out***.println("排序之前：");

**for** (**int** element : array) {

System.***out***.print(element + " ");

}

*quickSort*(array, 0, array.length - 1);

System.***out***.println("\n排序之后：");

**for** (**int** element : array) {

System.***out***.print(element + " ");

}

}

}

## 选择排序

假设第一个数为最大（或最小的数），然后从剩余的数据中与之进行比较，若有更大的则与之交换，继续进行比较，然后进行第二次，第三次的循环

**package** 选择排序;

/\*\*

\* 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。

\*

\* **@author** LaoK

\* 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。

\*/

**public** **class** SelectionSort {

**public** **static** **void** main(String[] args) {

**int** arr[] = { 95, 45, 52, 12, 37, 68, 59 };

System.***out***.println("排序结果为：");

**for** (**int** i : *selectionSort*(arr)) {

System.***out***.print(i + " ");

}

}

**public** **static** **int**[] selectionSort(**int**[] arr) {

**int** len = arr.length;

**int** minIndex, temp;

**for** (**int** i = 0; i < len - 1; i++) {

minIndex = i;

**for** (**int** j = i + 1; j < len; j++) {

**if** (arr[j] < arr[minIndex]) { // 寻找最小的数

minIndex = j; // 将最小数的索引保存

}

}

temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

}

**return** arr;

}

}

## 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后；
* 重复步骤2~5。

**package** 插入排序;

/\*\*

\* 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。

\*

\* **@author** LaoK

\*

\*/

**public** **class** InsertionSort {

**public** **static** **void** main(String[] args) {

**int** arr[] = { 95, 45, 52, 12, 37, 68, 59 };

System.***out***.println("排序结果为：");

**for** (**int** i : *insertionSort*(arr)) {

System.***out***.print(i + " ");

}

}

// 从小到大

**public** **static** **int**[] insertionSort(**int** arr[]) {

**int** len = arr.length;

**int** preIndex, current;

**for** (**int** i = 1; i < len; i++) {

preIndex = i - 1;

current = arr[i];

**while** (preIndex >= 0 && arr[preIndex] > current) {

arr[preIndex + 1] = arr[preIndex];

preIndex--;

}

arr[preIndex + 1] = current;

}

**return** arr;

}

}

## 归并排序

采用可分治法，

· 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

**package** 归并排序;

**import** java.util.Arrays;

**public** **class** MergeSort {

**public** **static** **void** main(String[] args) {

**int**[] arr = { 9, 8, 7, 6, 5, 4, 3, 2, 1 };

*sort*(arr);

System.***out***.println(Arrays.*toString*(arr));

}

**public** **static** **void** sort(**int**[] arr) {

**int**[] temp = **new** **int**[arr.length];// 在排序前，先建好一个长度等于原数组长度的临时数组，避免递归中频繁开辟空间

*sort*(arr, 0, arr.length - 1, temp);

}

**private** **static** **void** sort(**int**[] arr, **int** left, **int** right, **int**[] temp) {

**if** (left < right) {

**int** mid = (left + right) / 2;

*sort*(arr, left, mid, temp);// 左边归并排序，使得左子序列有序

*sort*(arr, mid + 1, right, temp);// 右边归并排序，使得右子序列有序

*merge*(arr, left, mid, right, temp);// 将两个有序子数组合并操作

}

}

**private** **static** **void** merge(**int**[] arr, **int** left, **int** mid, **int** right, **int**[] temp) {

**int** i = left;// 左序列指针

**int** j = mid + 1;// 右序列指针

**int** t = 0;// 临时数组指针

**while** (i <= mid && j <= right) {

**if** (arr[i] <= arr[j]) {

temp[t++] = arr[i++];

} **else** {

temp[t++] = arr[j++];

}

}

**while** (i <= mid) {// 将左边剩余元素填充进temp中

temp[t++] = arr[i++];

}

**while** (j <= right) {// 将右序列剩余元素填充进temp中

temp[t++] = arr[j++];

}

t = 0;

// 将temp中的元素全部拷贝到原数组中

**while** (left <= right) {

arr[left++] = temp[t++];

}

}

}

## 希尔排序

希尔排序是在简单插入排序的基础上进行了改进，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。**

**package** 希尔排序;

**public** **class** ShellSort {

**public** **static** **void** main(String[] args) {

**int** arr[] = { 95, 45, 52, 12, 37, 68, 59 };

System.***out***.println("排序结果为：");

**for** (**int** i : *shellSort*(arr)) {

System.***out***.print(i + " ");

}

}

**public** **static** **int**[] shellSort(**int**[] arr) {

**for** (**int** gap = arr.length / 2; gap > 0; gap /= 2) {

// 从第gap个元素，逐个对其所在组进行直接插入排序操作

**for** (**int** i = gap; i < arr.length; i++) {

**int** j = i;

**while** (j - gap >= 0 && arr[j] < arr[j - gap]) {

// 插入排序采用交换法

*swap*(arr, j, j - gap);

j -= gap;

}

}

}

**return** arr;

}

**public** **static** **void** swap(**int** arr[], **int** a, **int** b) {

arr[a] = arr[a] + arr[b];

arr[b] = arr[a] - arr[b];

arr[a] = arr[a] - arr[b];

}

}

## 堆排序

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。

**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。

**package** 堆排序;

**import** java.util.Arrays;

**public** **class** HeapSort {

**public** **static** **void** main(String[] args) {

**int**[] arr = { 9, 8, 7, 6, 5, 4, 3, 2, 1 };

System.***out***.println("排序前：");

System.***out***.println(Arrays.*toString*(arr));

*sort*(arr);

System.***out***.println("排序后：");

System.***out***.println(Arrays.*toString*(arr));

}

**public** **static** **void** sort(**int**[] arr) {

// 1.构建大顶堆

**for** (**int** i = arr.length / 2 - 1; i >= 0; i--) {

// 从第一个非叶子结点从下至上，从右至左调整结构

*adjustHeap*(arr, i, arr.length);

}

// 2.调整堆结构+交换堆顶元素与末尾元素

**for** (**int** j = arr.length - 1; j > 0; j--) {

*swap*(arr, 0, j);// 将堆顶元素与末尾元素进行交换

*adjustHeap*(arr, 0, j);// 重新对堆进行调整

}

}

/\*\*

\* 调整大顶堆（仅是调整过程，建立在大顶堆已构建的基础上）

\*

\* **@param** arr

\* **@param** i

\* **@param** length

\*/

**public** **static** **void** adjustHeap(**int**[] arr, **int** i, **int** length) {

**int** temp = arr[i];// 先取出当前元素i

**for** (**int** k = i \* 2 + 1; k < length; k = k \* 2 + 1) {// 从i结点的左子结点开始，也就是2i+1处开始

**if** (k + 1 < length && arr[k] < arr[k + 1]) {// 如果左子结点小于右子结点，k指向右子结点

k++;

}

**if** (arr[k] > temp) {// 如果子节点大于父节点，将子节点值赋给父节点（不用进行交换）

arr[i] = arr[k];

i = k;

} **else** {

**break**;

}

}

arr[i] = temp;// 将temp值放到最终的位置

}

/\*\*

\* 交换元素

\*

\* **@param** arr

\* **@param** a

\* **@param** b

\*/

**public** **static** **void** swap(**int**[] arr, **int** a, **int** b) {

**int** temp = arr[a];

arr[a] = arr[b];

arr[b] = temp;

}

}

## 计数排序

计数排序不是基于比较的排序算法，其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。

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。

**package** 计数排序;

**public** **class** CountingSort {

**private** **static** **int**[] countSort(**int**[] array, **int** k) {

**int**[] C = **new** **int**[k + 1];// 构造C数组

**int** length = array.length, sum = 0;// 获取A数组大小用于构造B数组

**int**[] B = **new** **int**[length];// 构造B数组

**for** (**int** i = 0; i < length; i++) {

C[array[i]] += 1;// 统计A中各元素个数，存入C数组

}

**for** (**int** i = 0; i < k + 1; i++)// 修改C数组

{

sum += C[i];

C[i] = sum;

}

**for** (**int** i = length - 1; i >= 0; i--)// 遍历A数组，构造B数组

{

B[C[array[i]] - 1] = array[i];// 将A中该元素放到排序后数组B中指定的位置

C[array[i]]--;// 将C中该元素-1，方便存放下一个同样大小的元素

}

**return** B;// 将排序好的数组返回，完成排序

}

**public** **static** **void** main(String[] args) {

**int**[] A = **new** **int**[] { 2, 5, 3, 0, 2, 3, 0, 3 };

**int**[] B = *countSort*(A, 5);

**for** (**int** i = 0; i < A.length; i++) {

System.***out***.println((i + 1) + "th:" + B[i]);

}

}

}

## 基数排序

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。

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）；

**package** 基数排序;

**public** **class** RadixSort {

**private** **static** **void** radixSort(**int**[] array, **int** d) {

**int** n = 1;// 代表位数对应的数：1,10,100...

**int** k = 0;// 保存每一位排序后的结果用于下一位的排序输入

**int** length = array.length;

**int**[][] bucket = **new** **int**[10][length];// 排序桶用于保存每次排序后的结果，这一位上排序结果相同的数字放在同一个桶里

**int**[] order = **new** **int**[length];// 用于保存每个桶里有多少个数字

**while** (n < d) {

**for** (**int** num : array) // 将数组array里的每个数字放在相应的桶里

{

**int** digit = (num / n) % 10;

bucket[digit][order[digit]] = num;

order[digit]++;

}

**for** (**int** i = 0; i < length; i++)// 将前一个循环生成的桶里的数据覆盖到原数组中用于保存这一位的排序结果

{

**if** (order[i] != 0)// 这个桶里有数据，从上到下遍历这个桶并将数据保存到原数组中

{

**for** (**int** j = 0; j < order[i]; j++) {

array[k] = bucket[i][j];

k++;

}

}

order[i] = 0;// 将桶里计数器置0，用于下一次位排序

}

n \*= 10;

k = 0;// 将k置0，用于下一轮保存位排序结果

}

}

**public** **static** **void** main(String[] args) {

**int**[] A = **new** **int**[] { 73, 22, 93, 43, 55, 14, 28, 65, 39, 81 };

*radixSort*(A, 100);

**for** (**int** num : A) {

System.***out***.print(num + " ");

}

}

}

## 桶排序

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

* 设置一个定量的数组当作空桶；
* 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；
* 从不是空的桶里把排好序的数据拼接起来。

**package** 桶排序;

**public** **class** BucketSort {

**public** **static** **void** main(String[] args) {

**int** arr[] = { 95, 45, 52, 12, 37, 68, 59 };

System.***out***.println("排序结果为：");

*bucketSort*(arr);

**for** (**int** i : arr) {

System.***out***.print(i + " ");

}

}

**public** **static** **void** bucketSort(**int**[] arr) {

**if** (arr == **null** || arr.length < 2) {

**return**;

} // 常用写法

**int** max = Integer.***MIN\_VALUE***;

**for** (**int** i = 0; i < arr.length; i++) {

max = Math.*max*(max, arr[i]);

}

**int**[] bucket = **new** **int**[max + 1];

**for** (**int** i = 0; i < arr.length; i++) { // 桶数组此下标有数据，数值就加一

bucket[arr[i]]++;

}

**int** i = 0;

**for** (**int** j = 0; j < bucket.length; j++) {

**while** (bucket[j]-- > 0) {

arr[i++] = j;

}

}

}

}