链滴

十大经典排序算法（详解）

作者：Yi-Xing

原文链接：https://ld246.com/article/1574554128193

来源网站：链滴

排序

□□排序算法有很多，包括插入排序，冒泡排序，堆排序，归并排序，选择排序，计数排序，基数排序桶排序，快速排序等。插入排序，堆排序，选择排序，归并排序和快速排序，冒泡排序都是比较排序它们通过对数组中的元素进行比较来实现排序，其他排序算法则是利用非比较的其他方法来获得有关入数组的排序信息。

□□如果有看不懂地方，可以使用数据结构可视化工具usfca帮助理解，也可以做一些练习题。

目录

原文链接：十大经典排序算法（详解）

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

一、冒泡排序

冒泡排序算法的原理如下：

- 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
- 对每一对相邻元素做同样的工作，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。

```
public static void sort(int[] array) {
    System.out.println(Arrays.toString(array));
    int temp;
    for (int i = 1; i < array.length; i++) {
        for (int j = 0; j < array.length-i; j++) {
            if (array[j] > array[j+1]) {
                temp = array[j];
                array[j] = array[j+1];
                array[j+1] = temp;
            }
        }
    }
    System.out.println(Arrays.toString(array));
}
```

二、选择排序

选择排序法的第一层循环从起始元素开始选到倒数第二个元素，主要是在每次进入的第二层循环之

，将外层循环的下标赋值给临时变量，接下来的第二层循环中，如果发现有比这个最小位置处的元素小的元素，则将那个更小的元素的下标赋给临时变量，最后，在二层循环退出后，如果临时变量改变则说明，有比当前外层循环位置更小的元素，需要将这两个元素交换。

```
public static void sort(int[] array) {
    System.out.println(Arrays.toString(array));
    int temp;
    for (int i = 0; i < array.length - 1; i++) {
        int min = i;
        for (int j = i + 1; j < array.length; j++) {
            if (array[min] > array[j]) {
                min = j;
            }
        }
        if (min != i) {
            temp = array[i];
            array[i] = array[min];
            array[min] = temp;
        }
    }
    System.out.println(Arrays.toString(array));
}
```

三、插入排序

如果有一个已经有序的数据序列，要求在这个已经排好的数据序列中插入一个数，但要求插入后此据序列仍然有序，这个时候就要用到一种新的排序方法——插入排序法

1、直接插入排序

对数组排序就是将数组中下标为 0 到倒数第一个的数进行排序。等价于：对数组中下标为 0 到倒数二个的数进行排序，然后把最后一个元素插入到这个有序数组中。

```
public static void sort(int[] array) {
    System.out.println(Arrays.toString(array));
    for (int i = 1; i < array.length; i++) {
        int temp = array[i];
        int j = i;
        while (j > 0 && temp < array[j - 1]) {
            array[j] = array[j - 1];
            j--;
        }
        array[j] = temp;
    }
    System.out.println(Arrays.toString(array));
}
```

2、二分插入排序

二分查找插入排序是直接插入排序的优化。区别是：在有序区中查找新元素插入位置时，为了减少素比较次数提高效率，采用二分查找算法确定插入位置。

```
public static void sort(int[] array) {
    System.out.println(Arrays.toString(array));
    for (int i = 1; i < array.length; i++) {
        int temp = array[i];
        int index = Java.lookup(array, 0, i - 1, temp);
        System.out.println("即将查找的数组" + Arrays.toString(Arrays.copyOf(array, i + 1)));
        System.out.println("将：temp" + temp + "，插入到：index" + index);
        for (int j = i; j > index; j--) {
            array[j] = array[j - 1];
        }
        array[index] = temp;
    }
    System.out.println(Arrays.toString(array));
}

/**
 * 二分查找,如果找不到，则返回小下标
 */
public static int lookup(int[] array, int min, int max, int value) {
    int mid;
    while (min <= max) {
        mid = (max + min) / 2;
        if (value < array[mid]) {
            max = mid - 1;
        } else if (value > array[mid]) {
            min = mid + 1;
        } else {
            return mid;
        }
    }
    return min;
}
```

四、希尔排序

希尔排序是插入排序的一种又称“缩小增量排序”，是直接插入排序算法的一种更高效的改进版本。

希尔排序是把数组按下标的一定增量进行分组，对每组进行直接插入排序。随着增量逐渐减少，每包含的元素越来越多，当增量减至 1 时，整个数组被分成一组，算法便终止。

```
public static void sort(int[] array) {
    System.out.println(Arrays.toString(array));
    int increment = array.length;
    do {
        //增量每次减半
        increment /= 2;
        for (int i = increment; i < array.length; i++) {
            int temp = array[i];
            int j = i;
            while (j >= increment && temp < array[j - increment]) {
                array[j] = array[j - increment];
                j -= increment;
            }
            array[j] = temp;
```

```
            }
        } while (increment != 1);
        System.out.println(Arrays.toString(array));
    }
```

五、快速排序

学习快排前先了解一下分治法：分治法就是将原问题划分成若干个规模较小而结构与原问题一致的问题，递归地解决这些子问题，然后再合并其结果，就得到原问题的解。

分治模式在每一层递归上都有三个步骤：

- 分解：将原问题分解成一系列子问题。
- 解决：递归的的解决各个子问题。若子问题足够小，则直接有解。
- 合并：将子问题的结构合并成原问题的解。

分治的关键点：

- 原问题可以一直分解为形式相同的子问题，当子问题规模较小时，可以自然求解，如一个元素本身序。
- 子问题的解通过合并可以得到原问题的解。
- 子问题的分解以及解的合并一定是比较简单的，否则分解和合并所花费的时间可能会超出暴力解法得不偿失。

快排算法：

- 分解：数组 A[left - right] 被划分为两个子数组 A[left - mid-1] 和 A[mid+1 - right]，使得 A[mid为大小居中的数，左侧 A[left - mid-1]中的每个元素都小于等于它，而右侧 A[mid+1 - right]中的每元素都大于等于它。
- 解决：通过递归调用快速排序，对子数组 A[left - mid-1] 和 A[mid+1 - right]进行排序。
- 合并：因为子数组都是原数组的一部分，索引不需要合并。

1、单向扫描法

用两个指针将数组划分为三个区间，扫描指针左边是确认小于等于主元素（上面说的 mid）的元素扫描指针到末指针之间的元素是未知，末指针右边的元素是确认大于等于主元素（上面说的 mid）的素。

代码实现：

```
    // 第一次 left为 0 ， right为 array.length - 1
    public static void sort(int[] array, int left, int right) {
        if (left < right) {
            int a = oneWay(array, left, right);
            sort(array, left, a - 1);
            sort(array, a + 1, right);
        }
    }

    public static int oneWay(int[] array, int left, int right) {
        // 主元素
```

```
        int scanning=array[left];
        // 头指针即扫描指针
        int start=left+1;
        // 末指针
        int end=right;
        // 用于交换值
        int temp;
        while(start<=end){
            // 头指针指定的元素如果小于等于主元素，则不用移动，以后往后扫描
            if(array[start]<=scanning){
                start++;
            }else{
                temp=array[start];
                array[start]=array[end];
                array[end]=temp;
                end--;
            }
        }
        // 当头指针和末指针指向一个值时候则继续循环
        // 如果指针值小于等于主元素，则头指针向后移动，指向了一个大于主元素的元素，此时应交末指针元素（小于主元素）和主元素进行交换
        // 如果指针大于主元素，则末指针向前移动，指向一个小于主元素的元素，接着讲末指针指向元素和主元素交换
        temp=array[end];
        array[end]=array[left];
        array[left]=temp;
        // 交换后末指针指向的是主元素所以返回末指针下标。
        return end;
    }
```

2、双向扫描法

头尾指针往中间扫描，从左找到大于主元素的元素，从右边找到小于主元素的元素，一直扫描，知头指针大于尾指针。

代码实现：

```
    // 第一次 left为 0 ， right为 array.length - 1
    public static void sort(int[] array, int left, int right) {
        if (left < right) {
            int a = twoWay(array, left, right);
            sort(array, left, a - 1);
            sort(array, a + 1, right);
        }
    }

    public static int twoWay(int[] array, int left, int right) {
        // 主元素
        int scanning = array[left];
        // 头指针
        int start = left + 1;
        // 末指针
        int end = right;
        // 用于交换值
```

```
    int temp;
    while (start <= end) {
        // 先从左开始扫描，如果左边有大于主元素的值，则停止扫描
        while (start <= end && array[start] <= scanning) {
            start++;
        }
        //然后从右开始扫描，如果右边有小于主元素的值，则停止扫描
        while (start <= end && array[end] >= scanning) {
            end--;
        }
        // 将头指针找到的元素和末指针找到的元素互换
        if (start < end) {
            temp = array[start];
            array[start] = array[end];
            array[end] = temp;
        }
    }
    // 和单向扫描的作用相似
    temp = array[left];
    array[left] = array[end];
    array[end] = temp;
    return end;
}
```

3、三指针分区法

三指针分区法主要用于选取主元素时，所选主元素在数组中有多个相同值。若数组中没有主元素的同值，没有必要用三指针扫描分区法。三指针分区法是单向扫描的扩展，它在单向扫描的基础上再加个指针repeat用于指向重复的主元素。

代码实现：

```
// 第一次 left为 0 ， right为 array.length - 1
public static void sort(int[] array, int left, int right) {
    if (left < right) {
        int a = threePoints(array, left, right);
        sort(array, left, a - 1);
        sort(array, a + 1, right);
    }
}

public static int threePoints(int[] array, int left, int right) {
    // 主元素
    int scanning = array[left];
    // 头指针
    int start = left + 1;
    // 末指针
    int end = right;
    // 重复元素指针
    int repeat=left;
    // 用于交换值
    int temp;
    while (start<=end){
        // 当左边的元素小于主元素时，将小的元素放在主元素左边（即交换位置），然后移动两个
```

```
针
            if(array[start]<scanning){
                temp = array[start];
                array[start] = array[repeat];
                array[repeat] = temp;
                start++;
                repeat++;
            // 当左边的元素等于主元素时，只用移动头指针即可，repeat仍然指向最左边的和主元素相
的元素
            }else  if(array[start]==scanning){
                start++;
            // 当左边的元素大于主元素时，则将该元素放到主元素右边。
            }else{
                temp = array[start];
                array[start] = array[end];
                array[end] = temp;
                end--;
            }
        }
        // 由于主元素在扫描的过程中已经被移动了，所以最后直接返回末指针即可。
        return end;
    }
```

4、对主元素的优化：

快速排序的目的是将时间复杂度降到 O(nlogn)，快排每次递归都将数组划分为两份，但是当主元素大时，主元素被移动到最右侧，而右边划分的数组为主元素下标 + 1 到 right 不满足start<=end的件，所以只剩左边划分的数组。如果每次拿到的主元素都是最大的，那么时间复杂度将被提升到 O(n<up>2</sup>)，所以我们确定的主元素数值尽量适中来提升效率。

1）三点中值法

我们可以使用三点中值法来确定主元素来提升效率，在 left、mid、right 之间选一个中值作为主元。

代码实现：

```
    // 确定主元素前，先调用该方法
    public static void threePointMedianMethod(int[] array,int left, int right) {
        // 先计算中间下标
        int mid =(left+right)/2;
        // 主元素的下标（中值）
        int scanning;
        if(array[left]<=array[mid]&& array[left]>=array[right]){
            // 如果左边的元素小于等于中间元素且大于等于右边的元素，则将left设置为主元素
            scanning=left;
        }else if(array[right]<=array[mid]&& array[right]>=array[left]){
            // 理由同上
            scanning=right;
        }else{
            scanning=mid;
        }
        // 为了不影响后面的逻辑，将主元素和第一个元素位置进行互换。
```

```
        int temp = array[scanning];
        array[scanning] = array[left];
        array[left] = temp;
    }
```

2）绝对中值法

使用三点中值法，只是解决了以前确定主元素的随机性，但是选定的中值并不是绝对的中值。如果让主元素是绝对的中值，则使用绝对中值法。

绝对中值法的思路：将数组按每 5 个一组进行分组，对每组进行插入排序，然后将其中值取出再进插入排序，取其中值即可，该中值为绝对中值。

由于确定绝对中值的时间复杂度为 O(n)，代码也比较复杂，所以实际中我们使用的三点中值法居多 Java 中的 JDK 使用的就是三点中值法。

代码实现：

```
    // 确定主元素前，先调用该方法
    public static void absoluteMedianMethod(int[] array, int left, int right) {
        // 先确定当前数组的长度
        int length = right - left + 1;
        // 每个数组5个元素
        int groupLength = length % 5 == 0 ? length / 5 : length / 5 + 1;
        // 存储每组的中值
        int[] groupMedian = new int[groupLength];
        // 用于记录 groupMedian 数组的下标
        int groupMedianIndex = 0;
        // 对每一组进行插入排序
        for (int i = 0; i < groupLength; i++) {
            // 单独处理最后一组，最后一组可能不是5个元素
            if (i == groupLength - 1) {
                // insertionSort方法是一个插入排序
                insertionSort(array, left + i * 5, right);
                // 将最后一组的中值存入数组中，然后下标自增
                groupMedian[groupMedianIndex++] = array[(left + i * 5 + right) / 2];
            } else {
                // 每组5个元素所以 right = left + 4
                insertionSort(array, left + i * 5, left + i * 5 + 4);
                // 将每一组的中值存入数组中，然后下标自增
                groupMedian[groupMedianIndex++] = array[left + i * 5 + 2];
            }
        }
        // 对groupMedian数组进行插入排序
        insertionSort(groupMedian, 0, groupMedian.length);
        // 该数组的中值
        int median=groupMedian[groupMedian.length/2];
        // 找到了该数组中的中值后，由于我们需要的是中值的下标
        // 我们还需要查找该中值的下标
        // 为了不影响后面的逻辑，我们找到下标后还需要将主元素和第一个元素位置进行互换。
    }
```

5、快排的进一步优化

插入排序的时间复杂度为 O(n²)，但是实际复杂度是[n(n-1)]/2；快排的时间复杂度 O(nlogn)，但是实际复杂度为 O(n(logn + 1))。经过计算我们不难发现，当数组的长度小于等于 8 ，插入排序的效率比快排的效率还要高，所以我们可以修改上面快排的sort方法进行优化。

修改后的 sort 方法：

```
public static void sort(int[] array, int left, int right) {
    if (left < right) {
        // 当数组的长度小于等于8的时候使用插入排序
        if (right - left + 1 <= 8) {
            // insertionSort方法是一个插入排序
            insertionSort(array, left, right);
        } else {
            int a = threePoints(array, left, right);
            sort(array, left, a - 1);
            sort(array, a + 1, right);
        }
    }
}
```

通过上面的例子我们可以发现，虽然计算时间复杂度时我们忽略了低阶项和常数因子等，但是当数规模过小的时候，效率还是会受到影响的。

六、归并排序

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的一个非常典型的应用递归的将数组分成两个子序列，先使子序列有序，再将子序列合并成一个有序列，称为二路归并。归排序是一种稳定的排序方法。

归并排序算法：

- 分解：将 n 个元素分成含有 n/2 个元素的子序列。
- 解决：对两个子序列进行递归排序。
- 合并：将两个已排序好的子序列进行合并得到排序的结果。

合并的具体实现：

- 创建一个大小为两个序列长度之和的临时数组。
- 创建两个指针，分别指向两个序列的起始位置。
- 比较两个指针所指向的元素，将小的元素存入临时数组，并移动指针到下一位置。
- 将没有比较完的元素直接拷贝到临时数组中。
- 将临时数组中的元素拷贝到原数组中。

代码实现：

```
// 第一次 left为 0 ， right为 array.length - 1
public static void sort(int[] array, int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        // 对左边进行排序
        sort1(array, left, mid);
```

```
            // 对右边进行排序
            sort1(array, mid + 1, right);
            // 合并
            merge(array, left, mid, right);
        }
    }

    public static void merge(int[] array, int left, int mid, int right) {
        // 创建一个辅助的临时数组
        int[] tempArray = new int[right - left + 1];
        // 记录临时数组的下标
        int tempIndex = 0;
        // 左边的指针
        int leftPointer = left;
        // 右边的指针
        int rightPointer = mid + 1;
        while (leftPointer <= mid && rightPointer <= right) {
            // 如果左边的元素小于等于右边的元素，则将左边的元素，存入临时数组，否则相反
            if (array[leftPointer] <= array[rightPointer]) {
                tempArray[tempIndex++] = array[leftPointer++];
            } else {
                tempArray[tempIndex++] = array[rightPointer++];
            }
        }
        // 将左序列剩下的元素，拷贝到临时数组中
        while (leftPointer <= mid) {
            tempArray[tempIndex++] = array[leftPointer++];
        }
        // 将右序列剩下的元素，拷贝到临时数组中
        while (rightPointer <= right) {
            tempArray[tempIndex++] = array[rightPointer++];
        }
        // 将临时数组拷贝到原数组中
        System.arraycopy(tempArray, 0, array, left, tempArray.length);
    }
```

七、堆排序

堆排序是指利用堆这种数据结构所设计的一种排序算法。堆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等于（或者大于等于）它的父节点，时间复杂度 O(nlgn)。

堆排序算法：

- 基础：首先要知道如何将数组调整为大顶堆和小顶堆。
- 堆化：反向调整使用每个子树都是大顶堆或小顶堆。
- 调整：把堆顶元素和最末元素对调，然后再次堆化该数组。
- 顺序：小顶堆降序，大顶堆升序。

代码实现：

```
    // 堆排序，升序
```

```
public static void sort(int[] array) {
    int temp;
    for (int i = array.length; i > 1; i--) {
        // 先将数组堆化，然后再交换元素
        pile(array, i);
        temp = array[i - 1];
        array[i - 1] = array[0];
        array[0] = temp;
    }
}

// 将数组大顶堆化，第一次length参数传入 array.length
public static void pile(int[] array, int length) {
    // array.length/2-1得到最小的有子节点的节点
    for (int i = length / 2 - 1; i >= 0; i--) {
        bigTopPile(array, i, length);
    }
}


// 大顶堆
public static void bigTopPile(int[] array, int root, int length) {
    //先取出当前元素i
    int temp = array[root];
    //从i结点的左子结点开始，也就是2i+1处开始
    for (int k = root * 2 + 1; k < length; k = k * 2 + 1) {
        //如果左子结点小于右子结点，k指向右子结点
        if (k + 1 < length && array[k] < array[k + 1]) {
            k++;
        }
        //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）
        if (array[k] > temp) {
            array[root] = array[k];
            root = k;
        } else {
            break;
        }
    }
    //将temp值放到最终的位置
    array[root] = temp;
}
```

八、计数排序

计数排序是一个非基于比较的排序算法，它的优势在于在对一定范围内的整数排序时，它的复杂度 O(n+k)（其中 k 是整数的范围），快于任何比较排序算法。当然这是一种牺牲空间换取时间的做法而且当 O(k)>O(n*log(n))的时候其效率反而不如基于比较的排序。

计数排序算法：

- 用辅助数组对数组中出现的数字进行计数。
- 假设元素均大于等于 0，依次扫描原数组，将元素值 k 记录在辅助数组的 k 位上。
- 元素转下标，下标转元素。

- 依次扫描辅助数组，如果为 1，将其插入目标数组的空白处。

代码实现：

方法一：

```
public static void sort(int[] array) {
    // 先找出数组中的最大值
    int max = array[0];
    for (int i = 1; i < array.length; i++) {
        if (max < array[i]) {
            max = array[i];
        }
    }
    int[] temp = new int[max + 1];
    // 将array中的元素以下标的形式存入数组中
    for (int value : array) {
        temp[value]++;
    }
    int index = 0;
    // 遍历temp如果值大于0，则将小标转为元素
    for (int i = 0; i < temp.length; i++) {
        while (temp[i] != 0) {
            array[index] = i;
            temp[i]--;
            index++;
        }
    }
}
```

方法二：

```
public static int[] sort(int[] array) {
    // 先找出数组中的最大值
    int max = array[0];
    for (int i = 1; i < array.length; i++) {
        if (max < array[i]) {
            max = array[i];
        }
    }
    int[] temp = new int[max + 1];
    // 将array中的元素以下标的形式存入数组中
    for (int value : array) {
        temp[value]++;
    }
    // 统计后现在temp[i]就表示小于等于i的值出现的次数，即i应该在array中下标为temp[i]处
    for (int i = 1; i < temp.length; i++) {
        temp[i] += temp[i - 1];
    }
    int[] sort = new int[array.length];
    for (int value : array) {
        // 统计是从1开始的所以需要-1
        sort[temp[value] - 1] = value;
        // 如果有相同的元素，则放在该数前面
```

```
            temp[value]--;
        }
        return sort;
    }
```

解决数组中的负数

假如数组中存在负数，比如对-10 到 10 的数字进行排序，则创建一个长度为 21 的数组，不过每个负责计数的元素变成了“索引-10”，即槽 0 对应-10 的计数，槽 1 对应-9 的计数……以此类推，并出槽的时候记得 +10 就是了。

使用方法二实现负数排序，如果感兴趣可以自己使用方法一实现一下。

```
    public static int[] sort(int[] array) {
        int max = 0;
        // 记录如果数组中存在负数的话
        int min = 0;
        // 先找出数组中的最大值和最小值
        for (int value : array) {
            if (value > max) {
                max = value;
            }
            if (value < min) {
                min = -value;
            }
        }
        int[] temp = new int[max + min + 1];
        // 将array中的元素以下标的形式存入数组中
        for (int value : array) {
            temp[value + min]++;
        }

        // 统计后现在temp[i]就表示小于等于i的值出现的次数，即i应该在array中下标为temp[i]处
        for (int i = 1; i < temp.length; i++) {
            temp[i] += temp[i - 1];
        }
        int[] sort = new int[array.length];
        for (int value : array) {
            // 下标是从0开始的所以需要-1
            sort[temp[value + min] - 1] = value;
            // 如果有相同的元素，则放在该数前面
            temp[value + min]--;
        }
        return sort;
    }
```

计数排序的两个弊端：

- 不擅长处理范围跨度很大的数字排序。这点很好理解，比如范围在-20000 到 20000，仅仅选 10 数字（比如：{-20000,-726...,20000,826...}）进行排序造成很大的空间浪费。
- 浮点型数字不好处理。对于两位小数的浮点，可采用先乘 100 后续再除 100 的方式，但这样非常费空间，比如小数位数多（试想 2 位整数 4 位小数的情况—68.4275）。

九、桶排序

桶排序是计数排序的升级版，工作的原理是将数组分到有限数量的桶子里。每个桶子再个别排序（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序）。当要被排序的数组内的数值是匀分配的时候，桶排序的线性时间 O(n)。

桶排序算法：

- 桶划分：设定桶的元素范围，进行第一次遍历，以获取最大值。
- 入桶：依次将元素放入适合自己的桶中（按桶设定的数字范围）。
- 桶内排序：各个桶之间的元素已经排好序了（桶 0 的元素 <
桶 1 的元素），但是桶内的元素顺序依然混乱，桶内元素的排序方式方法不限，插入、快排等等。

通过分配和收集过程来实现排序。

- 出桶：按顺序拿出就好。

桶排序的要点：

- 在额外空间充足的情况下，尽量增大桶的数量。
- 使用的映射函数能够将输入的 N 个数据均匀的分配到 K 个桶中。
- 对于桶内元素使用何种排序算法对于性能的影响至关重要。
- 当输入的数据可以均匀的分配到每一个桶中时，效率最高。
- 当输入的数据被分配到同一个桶中时，效率最低。
- 不擅于处理负数和小数。

代码实现：

```
public static LinkedList<Integer> sort(int[] array) {
    // 获取数组中的最大值
    int max = array[0];
    for (int i = 1; i < array.length; i++) {
        if (max < array[i]) {
            max = array[i];
        }
    }
    // 创建桶
    LinkedList<Integer>[] bucket = new LinkedList[array.length];
    // 桶下标
    int bucketIndex;
    for (int value : array) {
        // 计算出指定值应该进入哪个桶
        bucketIndex = value * array.length / (max + 1);
        if (bucket[bucketIndex] == null) {
            bucket[bucketIndex] = new LinkedList<>();
            bucket[bucketIndex].add(value);
        } else {
            // 将要存储的下标
            int index = 0;
            for (Integer i : bucket[bucketIndex]) {
                if (value > i) {
                    index++;
                } else {
                    break;
```

```
                }
            }
            bucket[bucketIndex].add(index, value);
        }
    }
    LinkedList<Integer> list = new LinkedList<>();
    for (LinkedList<Integer> bucketList : bucket) {
        if (bucketList != null && bucketList.size() > 0) {
            list.addAll(bucketList);
        }
    }
    return list;
}
```

十、基数排序

基数排序是一种非比较型整数排序算法，其原理是将整数按位数切割成不同的数字，然后按每个位分别比较。由于整数也可以表达字符串（比如名字或日期）和特定格式的浮点数，所以基数排序也不只能使用于整数。

基数排序算法：

- 首先遍历数组，根据个位的数值将它们分配至编号 0 到 9 的桶子中。
- 接下来将这些桶子中的数重新串接起来。
- 然后再次进行分配，这次是根据十位数来分配。
- 持续进行以上的动作直至最高位数为止。
- 最低位优先 LSD 法或最高位优先 MSD 法，LSD 的排序方式由键值的最右边开始，而 MSD 则相反。

基数排序 vs 计数排序 vs 桶排序

- 这三种排序方式不擅于对负数，小数排序。
- 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。
- 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。
- 基数排序：根据键值的每位数字来分配桶。

代码实现：

```
public static void sort(int[] array) {
    int max = array[0];
    // 10个桶
    ArrayList<Integer>[] list = new ArrayList[10];
    // 初始化集合数组
    allArray(list);
    for (int i = 1; i < array.length; i++) {
        if (max < array[i]) {
            max = array[i];
        }
    }
    // 判断max是几位数
```

```
    int length = 0;
    while (max != 0) {
        max /= 10;
        length++;
    }
    for (int i = 0; i < length; i++) {
        // 遍历数组
        for (int value : array) {
            // 得到指定位置上的数
            int num = (int) (value / Math.pow(10, i)) % 10;
            list[num].add(value);
        }
        int index=0;
        // 将集合中的数拷贝到数组中
        for (ArrayList<Integer> arrayList:list){
            for(Integer integer:arrayList){
                array[index]=integer;
                index++;
            }
        }
        System.out.println(Arrays.toString(array));
        // 初始化集合数组
        allArray(list);
    }
}

// 对集合数组初始化
public static void allArray(ArrayList<Integer>[] list) {
    for (int i = 0; i < list.length; i++) {
        list[i] = new ArrayList<>();
    }
}
```