PK吧！排序算法

今日目标:

1：能说出各种排序的过程

2：完成对应的题目

排序算法(重要)

在这一章节中我们要来学习排序算法，这对绝大部分从事软件行业的人来说都不陌生，可能你接触的第一个跟算法相关的东西就是排序，在很多的编程语言中都提供了排序的API可直接使用，在日常的软件研发过程中我们也经常使用到排序，当然了排序的算法有很多，甚至有些排序算法我们都没听过，在我们本章节的课程中只讲其中最经典最常用的排序算法：冒泡排序，选择排序，插入排序，归并排序，快速排序，计数排序，桶排序，基数排序；在这些排序算法中如果按照时间复杂度来分类大致可以分为三类：$O(n^2)$：冒泡排序，选择排序，插入排序；$O(n*logn)$：归并排序，快速排序，希尔排序；$O(n)$：计数排序，基数排序，桶排序。

我们要学习这么多的排序算法，除了学习其原理，实现其代码外还要评判出各种排序算法之间性能，效率。那我们应该从哪几个方面来分析一个排序算法是好是坏呢？所以在正式进入排序算法之前，我们先来说说排序算法的评判标准。

评判排序算法好坏的标准

对于众多的排序算法我们要将它们做一个对比需要从如下三个方面着手：

1：时间复杂度

时间复杂度其实就代表了一个算法执行的效率，我们在**分析排序算法的时间复杂度时要分别给出最好情况、最坏情况、平均情况下的时间复杂度。**为什么要区分这三种时间复杂度呢？第一，有些排序算法会区分，为了好对比，所以我们最好都做一下区分。第二，对于要排序的数据，有的接近有序，有的完全无序。有序度不同的数据，对于排序的执行时间肯定是有影响的，我们要知道排序算法在不同数据下的性能表现。

在之前的章节中学习复杂度的分析时我们说过，复杂度反映的是一个算法随着n的变化的一个增长趋势，在表示的时候往往会忽略表达式中的系数，低阶，常量，但是实际的软件开发中，我们排序的可能是 50个、100 个、1000 个这样规模很小的数据，所以，**在对同一阶时间复杂度的排序算法性能对比的时候，我们就要把系数、常数、低阶也考虑进来。**

在本章节中讲的都是排序算法中的不同实现，其中有些排序算法是基于数据比较的排序算法，这些算法在执行过程中会涉及到比较元素大小，然后元素的交换或者移动，所以**在分析基于比较的排序算法时要将元素比较/交换/移动的次数也考虑进来。**

2：空间复杂度

空间复杂度在一个层面代表了算法对存储空间的消耗程度，我们可以简单的理解为算法的内存消耗，在这里我们还引入另外一个概念：**in-place** 和 **out-place**；其中in-place可以称为原地排序就是特指空间复杂度为O(1)的排序算法，算法只占用常数内存，不占用额外内存，而out-place的算法需要占用额外内存。

3：算法稳定性

如果我们只用上面提到的时间复杂度和空间复杂度来度量一个排序算法其实是不够的，针对排序算法，还有一个指标就是：**稳定性。**

所谓排序算法的稳定性指的是：如果待排序的序列中存在值相等的元素，经过排序之后，相等元素之间原有的先后顺序不变。

举个例子，有一组数据：3 7 2 7 5 8 9，我们按照大小排序之后的数据为：2 3 5 7 7 8 9，在这组数据中有两个7，如果经过某种排序算法后两个7的前后顺序没有发生改变则称该算法是稳定的排序算法，否则称为该算法是不稳定的排序算法。

在我们后续每学习一个排序算法我们都应该使用刚刚所讲的这个三个评判规则去分析该算法，好，接下来我们就依次的对每一个排序算法进行学习。

冒泡排序

原理

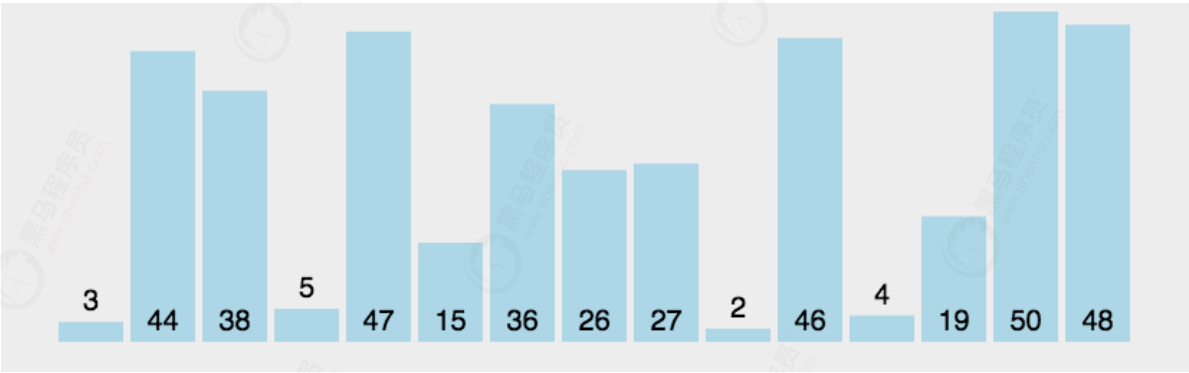
冒泡排序(Bubble Sort)是一种简单的排序算法，它通过依次比较两个相邻的的元素，看两个元素是否满足大小关系要求，如果不满足则交换两个元素。每一次冒泡会让至少一个元素移动到它应该在的位置上，这样n次冒泡就完成了n个数据的排序工作。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

接下来对整个算法的过程进行描述：

- 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两个步骤，除了最后一个；
- 重复前三步，直到排序完成。

为了便于大家理解，我们以一副动图的形式展现给大家(注：以下动图来自网络，下同)

链接：https://mmbiz.qpic.cn/mmbiz_gif/QCu849YTaIOOdfiakqsTRHKk9icjqQZJYuffv5BticjiaK3BNNtdH6dRFglibdwgA9w2oR6QZTadJeZHdOsicqyjasPg/640?tp=webp&wxfrom=5&wx_lazy=1



实现

理解了冒泡排序的原理后代码实现如下：

```
/**
 * 冒泡排序算法
 * 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
 *
 * 步骤：
 *  1：比较相邻的元素。如果第一个比第二个大，就交换它们两个；
    2：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；
    3：针对所有的元素重复以上的步骤，除了最后一个；
    4：重复步骤1~3，直到排序完成。
 */
public class BubbleSort {

    public static void bubbleSort(int[] data) {
        int len = data.length;
        //特殊情况
        if (len < 2) {
            return;
        }
        //开始冒泡
        for (int i=0;i<len;i++) {
            //一次循环就是一次冒泡,一次冒泡会把当前最大的元素挪到后面
            for (int j=0;j < len - i -1;j++) {
                //前一个数据大于后一个则交换
                if (data[j] > data[j+1]) {
                    swap(data,j,j+1);
                }
            }
        }
    }
    private static void swap(int[] array,int i,int j) {
        if (i == j) {
            return;
        }
        array[i] ^= array[j];
        array[j] ^= array[i];
        array[i] ^= array[j];
       /* data[j] = data[j] + data[j+1];
        data[j+1] = data[j] - data[j+1];
        data[j] = data[j] - data[j+1];*/
    }

    public static void main(String[] args) {
        int[] data = new int[]{5,2,6,5,9,0,3};
        System.out.println("排序前:"+ Arrays.toString(data));
        bubbleSort(data);
        System.out.println("排序后:"+ Arrays.toString(data));
    }

}
```

实际上，这里的冒泡排序算法还可以继续优化：因为当某次冒泡时发现已经没有数据需要进行交换时，说明所有元素都已经达到有序状态了，此时就不用再执行后续的冒泡操作了，接下来对之前的冒泡进行优化的代码实现如下：

```
    //冒泡
    public static void bubbleSort(int[] data) {
        int len = data.length;
        //特殊情况
```

```
        if (len < 2) {
            return;
        }
        //开始冒泡
        for (int i=0;i<len;i++) {
            //一次循环就是一次冒泡,一次冒泡会把当前最大的元素挪到后面

            //是否需要提前结束冒泡的标识
            boolean flag = true;

            for (int j=0;j < len - i -1;j++) {
                //前一个数据大于后一个则交换
                if (data[j] > data[j+1]) {
                    swap(data,j,j+1);
                    flag = false;
                }
            }
            //在当前这次冒泡中如果所有元素都不需要进行交换则证明所有元素都已有序
            if (flag) {
                break;
            }
        }
    }
    private static void swap(int[] array,int i,int j) {
        if (i == j) {
            return;
        }
        array[i] ^= array[j];
        array[j] ^= array[i];
        array[i] ^= array[j];
       /* data[j] = data[j] + data[j+1];
        data[j+1] = data[j] - data[j+1];
        data[j] = data[j] - data[j+1];*/
    }

    public static void main(String[] args) {
        int[] data = new int[]{5,2,6,9,0,3,10,13,15};
        System.out.println("排序前:"+ Arrays.toString(data));
        bubbleSort(data);
        System.out.println("排序后:"+ Arrays.toString(data));
    }
```

总结

对于冒泡排序我们要使用之前学习的三个标准来进行评判：

1：冒泡排序的时间复杂度是多少?

最好情况下，要排序的数据已经是有序的了，我们只需要进行一次冒泡操作，就可以结束了，所以**最好情况时间复杂度是 O(n)**。而最坏的情况是，要排序的数据刚好是倒序排列的，我们需要进行 n 次冒泡操作，所以**最坏情况时间复杂度为 O(n2)**。

那对于平均情况下的复杂度呢，对于包含 n 个数据的数组，这 n 个数据就有 n! 种排列方式，每种排列方式冒泡排序执行的时间肯定是不一样的，我们可以运用概率论方法来分析平均情况复杂度，只不过这里涉及到的数学推理和计算会稍显复杂，在这里我们可以换一种思路，使用一种并不严格但是很有用的分析方式：*有序度分析方式*，在这里需要提出两个概念：*有序度和逆序度*

有序度指的是数组中已经具有有序关系的元素对的个数，比如一组数据：2 1 3 4，其中已经存在有序关系的数据对分别为：(1,3),(1,4),(2,3),(2,4),(3,4)，所以该数列的有序度为：5

同理，对于一个倒叙排列的数列，比如：6 5 4 3 2 1有序度为0，对于一个完全有序的数列，比如：1 2 3 4 5 6，有序度为：n*(n-1)/2,也就是15。这种完全有序的有序度我们可以称之为满有序度，那逆序度的定义刚好跟有序度的定义相反，因此我们可以得到一个公式：*逆序度=满有序度-有序度**，而我们排序的过程就是增加有序度，减少逆序度的过程，直到到达满有序度，说明排序完成。

对于冒泡排序而言，其中涉及两个操作：比较和交换，我们每交换一次数据，有序度就加1，因为一个数列一旦给出满有序度是固定的，有序度也是固定的，因此不管算法怎么改进，交换次数总确定的，即为逆序度，也就是：n*(n-1)/2 - 有序度；

因此对于一个包含n个数据的数组进行冒泡排序，平均交换的次数是多少呢？最坏的情况下有序度为0，要进行n* (n-1)/2 次交换，最好的情况下是满有序度不需要进行交换，因此平均情况下我们可以取一个中间值：n*(n-1)/4换句话说，平均情况下，需要 n*(n-1)/4 次交换操作，比较操作肯定要比交换操作多，而复杂度的上限是 O(n2)，所以**平均情况下的时间复杂度就是 O(n2)。**

2：冒泡排序的空间复杂度是多少?

冒泡的过程只涉及相邻数据的交换操作，只需要常量级的临时空间，所以它的空间复杂度为 O(1)，是一种in-place排序算法。

3：冒泡排序是稳定的排序算法吗?

在冒泡排序中，只有交换才可以改变两个元素的前后顺序。为了保证冒泡排序算法的稳定性，当有相邻的两个元素大小相等的时候，我们不做交换，相同大小的数据在排序前后不会改变顺序，所以冒泡排序是稳定的排序算法。

插入排序

原理

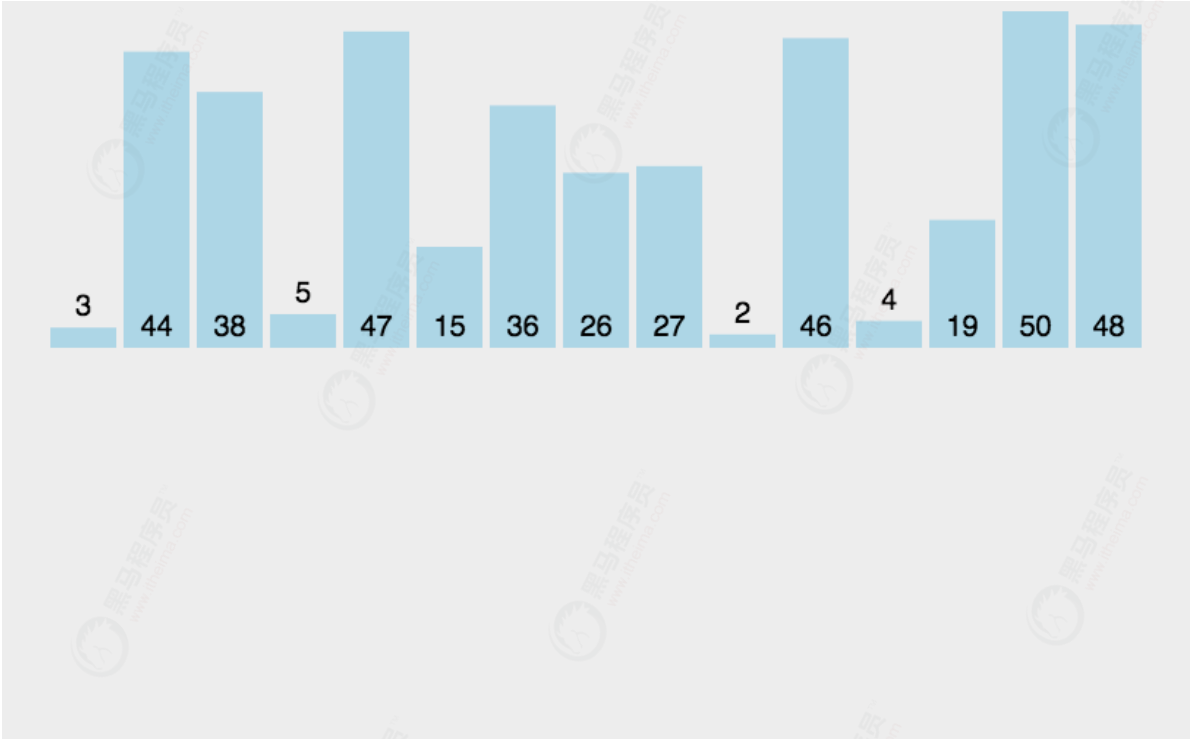
插入排序(Insertion Sort)的原理是：将数组中的数据分为两个区间，已排序区间和未排序区间。初始已排序区间只有一个元素，就是数组的第一个元素。插入算法的核心思想是取未排序区间中的元素，在已排序区间中找到合适的插入位置将其插入，并保证已排序区间数据一直有序。重复这个过程，直到未排序区间中元素为空，算法结束。

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。

动图效果如下：

链接：https://mmbiz.qpic.cn/mmbiz_gif/QCu849YTaIOOdfiakqsTRHKk9icjqQZJYusIFPUq7PlJn7maGNCmlhzTnLCkRcNjulAZk34Elic3oeVka2u4icXWDA/640?tp=webp&wxfrom=5&wx_lazy=1



实现

插入排序的算法实现如下：

```
/**
 * 插入排序算法
 * 插入排序（Insertion-Sort）的算法描述是一种简单直观的排序算法。
 * 我们将数组中的数据分为两个区间，已排序区间和未排序区间。初始已排序区间只有一个元素，就是数组的第一个元素。插入算法的核心思想是取未排序区间中的元素，在已排序区间中找到合适的插入位置将其插入，并保证已排序区间数据一直有序。重复这个过程，直到未排序区间中元素为空，算法结束。
 *
 * 步骤：
 1：从第一个元素开始，该元素可以认为已经被排序；
 2：取出下一个元素，在已经排序的元素序列中从后向前扫描；
 3：如果该元素（已排序）大于新元素，将该元素移到下一位置；
 4：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；
 5：将新元素插入到该位置后；
 6：重复步骤2~5。
 */
public class InsertionSort {

    public static void insertionSort(int[] data) {
        int len = data.length;
        //特殊
        if (len < 2) {
            return;
        }
        //定义前置索引和当前元素
        int preIdx,current;
        //第一个元素默认已排好序
        for (int i=1;i<len;i++) {//i相当于未排序区间的首下标
            //当前元素要插入到已排序区间中某个位置去
            current = data[i];
            //在已经排序的元素序列中从后向前扫描
            preIdx = i-1;
```

```
            //从后向前依次和当前元素data[i]进行比较,找到当前元素要插入的位置,
            while (preIdx >=0 && data[preIdx] > current) {
                //比较过程中如果元素大于当前的元素则将元素后移一位
                data[preIdx+1] = data[preIdx];
                preIdx--;
            }
            //比较过程中如果该元素小于等于当前元素,则将当前元素放在该元素后面
            data[preIdx+1] = current;
        }
    }

    public static void main(String[] args) {
        int[] data = new int[]{5,2,6,5,9,0,3};
        System.out.println("排序前:"+ Arrays.toString(data));
        insertionSort(data);
        System.out.println("排序后:"+ Arrays.toString(data));
    }
}
```

总结

1：插入排序的时间复杂度是多少?

如果要排序的数据已经是有序的，我们并不需要搬移任何数据。如果我们从尾到头在有序数据组里面查找插入位置，每次只需要比较一个数据就能确定插入的位置。所以这种情况下，**最好时间复杂度为O(n)**。注意，这里是从尾到头遍历已经有序的数据。

如果数组是倒序的，每次插入都相当于在数组的第一个位置插入新的数据，所以需要移动大量的数据，所以**最坏情况时间复杂度为 O(n2)**。还记得我们在数组中插入一个数据的平均时间复杂度是多少吗？没错，是 O(n2)。所以，对于插入排序来说，每次插入操作都相当于在数组中插入一个数据，循环执行 n 次插入操作，所以**平均时间复杂度为 O(n2)**。

2：插入排序的空间复杂度是多少?

从实现过程可以很明显地看出，插入排序算法的运行并不需要额外的存储空间，所以**空间复杂度是O(1)**，也就是说，这是一个in-place排序算法。

3：插入排序是稳定的排序算法吗?

在插入排序中，对于值相同的元素，我们可以选择将后面出现的元素，插入到前面出现元素的后面，这样就可以保持原有的前后顺序不变，所以**插入排序是稳定的排序算法**。

选择排序

原理

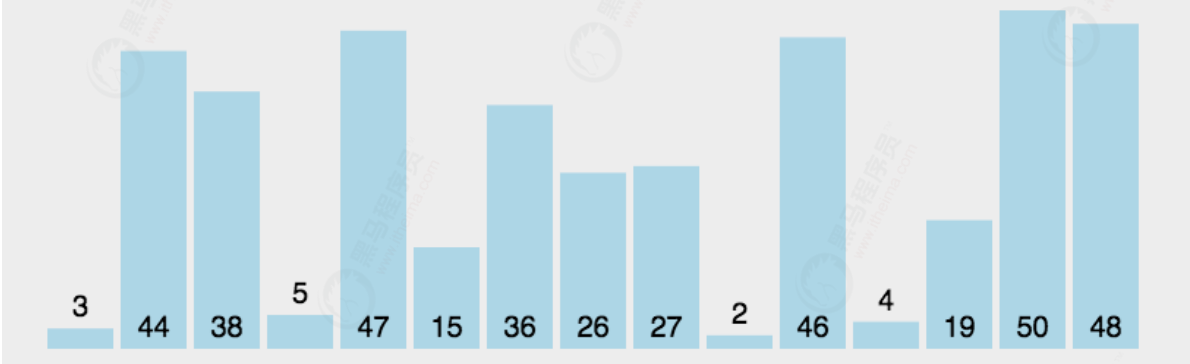
选择排序(Selection Sort)的原理有点类似插入排序，也分已排序区间和未排序区间。但是选择排序每次会从排序区间中找到最小的元素，将其放到已排序区间的末尾。

算法描述如下：

- 初始状态：无序区间为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。

动图效果如下：

链接：https://mmbiz.qpic.cn/mmbiz_gif/QCu849YTaIOOdfiakqsTRHKk9icjqQZJYuROpQscX9fen1nqP1nia2lUADm29QpKHn7lqPn2Aiaic4DoPQ72GYKak6w/640?tp=webp&wxfrom=5&wx_lazy=1



实现

代码实现如下:

```java
/**
 * 选择排序算法
 * 选择排序算法的实现思路有点类似插入排序，也分已排序区间和未排序区间。但是选择排序每次会从未排序区间中找到最小的元素，将其放到已排序区间的末尾
 */
public class SelectionSort {

    public static void selectionSort(int[] data) {
        int len = data.length;
        //特殊
        if (len < 2) {
            return;
        }
        //算法开始
        int minIdx;//无序区最小元素下标
        for (int i=0;i<len;i++) { //i代表了无序区的首下标
            //找到无序区的最小值
            minIdx= i;
            for (int j=i+1;j<len;j++) {
                if (data[j] < data[minIdx]) {
                    minIdx = j;
                }
            }
            //交换最小元素和无序区第一个元素
            swap(data,i,minIdx);
        }
    }

    private static void swap(int[] array,int i,int j) {
        if (i == j) {
            return;
        }
        array[i] ^= array[j];
        array[j] ^= array[i];
        array[i] ^= array[j];
    }

    public static void main(String[] args) {
        int[] data = new int[]{5,2,6,5,9,0,3};
        System.out.println("排序前:"+ Arrays.toString(data));
```

```
        selectionSort(data);
        System.out.println("排序后:"+ Arrays.toString(data));
    }
}
```

总结

1：选择排序的时间复杂度是多少?

结合之前的分析方式分析可知选择排序的**最好情况时间复杂度为O(n2)，最坏情况时间复杂度为:O(n2)，平均情况下的时间复杂度为：O(n2)。**

2：选择排序的空间复杂度是多少?

通过算法的实现我们可以发现，选择排序的**空间复杂度为O(1)，是一个in-place排序算法**

3：选择排序是一个稳定的排序算法吗?

注意：**选择排序不是一个稳定的排序算法**，为什么呢？选择排序每次都要找剩余未排序元素中的最小值，并和未排序区间的第一个元素进行交换位置，这样破坏了稳定性，比如 5，8，5，2，9 这样一组数据，使用选择排序算法来排序的话，第一次找到最小元素 2，与第一个 5 交换位置，那第一个 5 和中间的 5 顺序就变了，所以就不稳定了。正是因此，从稳定性上来说选择排序相对于冒泡排序和插入排序就稍微逊色了。

归并排序

原理

归并排序(Merge Sort)的核心思想还是蛮简单的。如果要排序一个数组，我们先把数组从中间分成前后两部分，然后对前后两部分分别排序，再将排好序的两部分合并在一起，这样整个数组就都有序了。

归并排序使用的是**分治思想**。分治，顾名思义，就是分而治之，将一个大问题分解成小的子问题来解决。小的子问题解决了，大问题也就解决了。从我刚才的描述，你有没有感觉到，分治思想跟我们前面讲的递归思想很像。是的，分治算法一般都是用递归来实现的。分治是一种解决问题的处理思想，递归是一种编程技巧，这两者并不冲突。而对于递归就是要找到递推公式及终止条件，所以我们可以先写出归并排序的递推公式

```
mergeSort(m->n) = merge(mergeSort(m->k),mergeSort(k+1->n));当m=n时终止
```

我们来解释一下这个公式：我们要对m->n之间的数列进行排序，其实可以拆分成对m->k之间的数列进行排序，以及对k+1->n之间的数列排序，然后将连个拍好序的数列进行合并就称为了最终的数列，同样的道理，每一段数列的排序又可以继续往下拆分，形成递归。

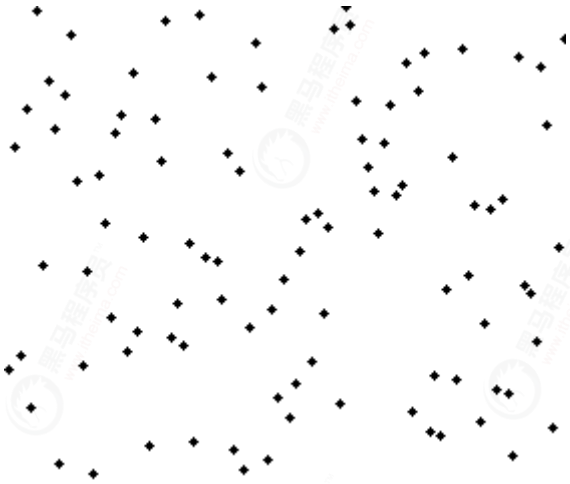
算法描述：

- 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。

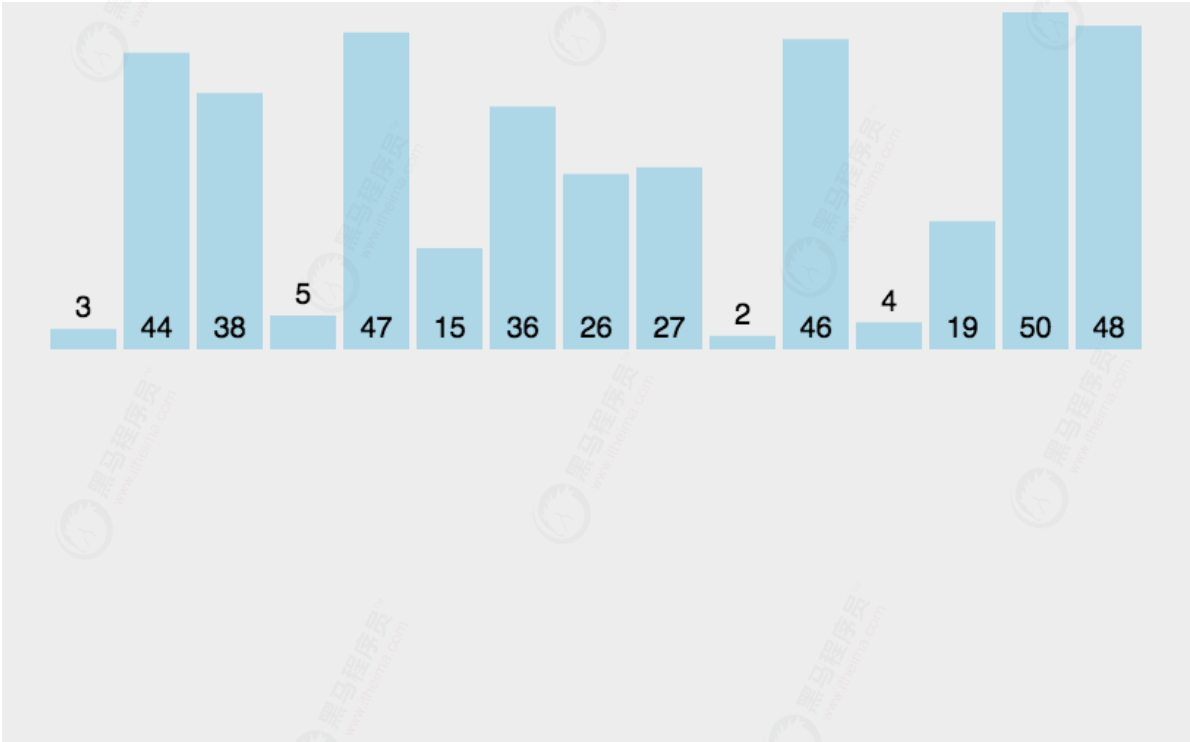
动图效果如下：

链接：

https://mmbiz.qpic.cn/mmbiz_gif/XaklVibwUKn4hqBOkhUQouzu2FDmmen4XHM8pExUK8sHf29rRqpwal4xeqhtJCxt7VDCA0ia5ZveB5rsdKngf6Iw/640?tp=webp&wxfrom=5&wx_lazy=1



https://mmbiz.qpic.cn/mmbiz_gif/QCu849YTaIOOdfiakqsTRHKk9icjqQZJYuEib77DsqHVhsUlM8iayfn5sV1ou3LKGjtDqzhhNPeibLDgyKw6S3Diam4g/640?tp=webp&wxfrom=5&wx_lazy=1



实现

代码实现如下：

```
/**
 * 归并排序算法
 *  归并排序的核心思想还是蛮简单的。如果要排序一个数组，我们先把数组从中间分成前后两部分，
    然后对前后两部分分别排序，再将排好序的两部分合并在一起，这样整个数组就都有序了
 */
public class MergeSort {

    //归并排序
    int[] temp;//存放每层合并后的结果

    public int[] sortArray(int[] nums) {
        temp = new int[nums.length];
        mergeSort(nums,0,nums.length-1);
        return nums;
    }
```

```java
    public void mergeSort(int[] nums,int left,int right) {
        //terminal
        if (left >= right) {
            return;
        }
        //divide
        int mid = (left + right) >> 1;
        mergeSort(nums,left,mid);//对nums数组里[left,mid]部分进行排序。
        mergeSort(nums,mid+1,right);//对nums数组里[mid+1,right]部分进行排序。
        //Conquer 此时[left,mid] [mid+1,right]两个区间内的数据均有序，合并使得[left,right]有序
        //双指针将排好序的数据填充到临时空间temp
        int i=left;
        int j=mid+1;
        int k=0;
        while (i<=mid && j<=right) {
            if (nums[i] <= nums[j]) {
                temp[k++] = nums[i++];
            }else {
                temp[k++] = nums[j++];
            }
        }
        while (i<=mid) {//[left,mid]区间还有剩余
            temp[k++] = nums[i++];
        }
        while (j <= right) {//[mid+1,right]区间还有剩余
            temp[k++] = nums[j++];
        }
        //将临时空间中排好序的数据填充到nums中[left,right]区间
        for (int m=right;m>=left;m--) {
            nums[m] = temp[--k];
        }
    }


    public static void main(String[] args) {
        int[] data = new int[]{5,2,6,5,9,0,3};
        System.out.println("排序前:"+ Arrays.toString(data));
        int[] mergeSort = new MergeSort().sortArray(data);
        System.out.println("排序后:"+ Arrays.toString(mergeSort));
    }
}
```

总结

1：归并排序的时间复杂度是多少?

时间复杂度：O(nlogn)。由于归并排序每次都将当前待排序的序列折半成两个子序列递归调用，然后再合并两个有序的子序列，而每次合并两个有序的子序列需要O(n) 的时间复杂度，所以我们可以列出归并排序运行时间 T(n) 的递归表达式

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

根据主定理我们可以得出归并排序的时间复杂度为 O(n * logn)。

> 主定理

归并排序的时间复杂度为： $O(n*logn)$

从我们的原理分析和伪代码可以看出，归并排序的执行效率与要排序的原始数组的有序程度无关，所以其时间复杂度是非常稳定的，不管是最好情况、最坏情况，还是平均情况，时间复杂度都是O(nlogn)

2：归并排序的空间复杂度是多少?

归并排序的空间复杂度是多少呢？是：$O(n)$，因为归并排序的合并函数，在合并两个有序数组为一个有序数组时，需要借助额外的存储空间。这一点你应该很容易理解，但是如果我们继续按照分析递归时间复杂度的方法，通过递推公式来求解，那整个归并过程需要的空间复杂度就是 O(nlogn)。不过，类似分析时间复杂度那样来分析空间复杂度，这个思路对吗?

实际上，递归代码的空间复杂度并不能像时间复杂度那样累加。刚刚我们忘记了最重要的一点，那就是，尽管每次合并操作都需要申请额外的内存空间，但在合并完成之后，临时开辟的内存空间就被释放掉了。在任意时刻，CPU 只会有一个函数在执行，也就只会有一个临时的内存空间在使用。临时内存空间最大也不会超过 n 个数据的大小，所以**空间复杂度是 O(n)**，因此归并排序并不是一种in-place排序算法而是一种out-place排序算法。

3：归并排序是稳定的排序算法吗?

归并排序算法稳定还是不稳定取决于合并函数merge()。也就是两个有序子数组合并成一个有序数组的那部分代码，通过分析merge函数我们发现，**归并排序也是一个稳定的排序算法。**

快速排序

原理

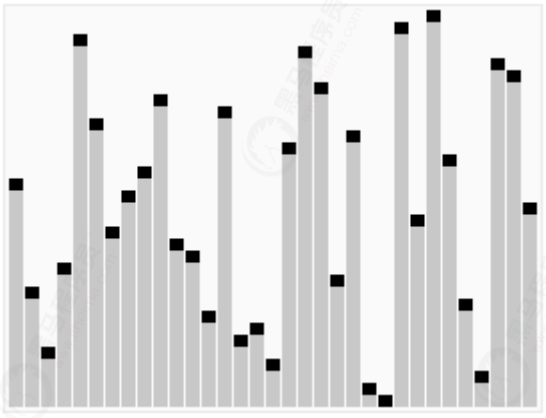
快速排序(Quick Sort)算法，简称快排，利用的也是分治的思想，初步看起来有点像归并排序，但是其实思路完全不一样，快排的思路是：如果要对m->n之间的数列进行排序，我们选择m->n之间的任意一个元素数据作为分区点(Pivot)，然后我们遍历m->n之间的所有元素，将小于pivot的元素放到左边，大于pivot的元素放到右边，pivot放到中间，这样整个数列就被分成三部分了，m->k-1 之间的元素是小于pivot的，中间是pivot，k+1->n之间的元素是大于pivot的。然后再根据分治递归的思想处理两边区间的的元素数列，直到区间缩小为1，就说明整个数列都已有序了。

算法描述如下：

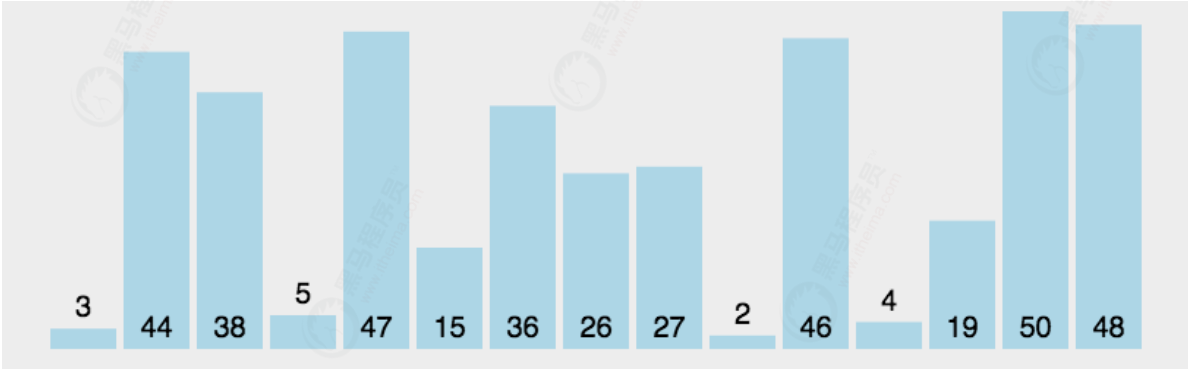
- 从数列中挑出一个元素，称为 “基准”（**pivot**）；
- 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
- 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

动图效果如下：

连接：https://mmbiz.qpic.cn/mmbiz_gif/XaklVibwUKn4hqBOkhUQouzu2FDmmen4Xn9ornBuYJeibehOnHAh3DibRoXHbfdpT6hhtkr4yFGViaWpb7rfvxfpiaw/640?tp=webp&wxfrom=5&wx_lazy=1



https://mmbiz.qpic.cn/mmbiz_gif/QCu849YTaIOOdfiakqsTRHKk9icjqQZJYuGKg8QiaSGDcDuicwJPIPniceyQYUz4uch8XvGsHOB2MUdhDsrvhwibibrKA/640?tp=webp&wxfrom=5&wx_lazy=1



实现

代码实现如下：

```
/**
 * https://visualgo.net
 * 快速排序算法
 *  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；
    3：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。
 *
 */
public class QuickSort {
    /**
     * 快排
     * 借助递归和分区的思想来实现
     * @param arr
     * @param begin
     * @param end
     */
    public int[] sortArray(int[] nums) {
        //特殊
        if (nums.length < 2) {
            return nums;
        }
        quickSort(nums,0,nums.length-1);
        return nums;
```

```
    }
    public void quickSort(int[] nums,int left,int right) {
        //terminal
        if (left >=right) {
            return;
        }
        //current logic partition
        int position = partition(nums,left,right);
        //recur
        quickSort(nums,left,position-1);
        quickSort(nums,position+1,right);
    }
    /*public int partition(int[] nums,int left,int right) {
        int pivotIdx = new Random().nextInt(right - left +1) + left;//预设基准数据的下标：可以是开头，结尾，随机
        swap(nums,pivotIdx,right);//将基准数据放置在right位置,也可放在left位置
        //begin partition
        int pivot = nums[right];
        int j = left;
        for (int i=left;i<right;i++) {
            if (nums[i] <= pivot) {
                swap(nums,i,j);
                j++;
            }
        }
        //[left,j)是小于pivot的数据,[j,right-1)是大于pivot的数据
        //将基准数据放到它应该在的位置
        swap(nums,j,right);
        return j;
    }*/

    public int partition(int[] nums,int left,int right) {
        //预设基准数据的下标：可以是开头，结尾，随机
        int pivotIdx = new Random().nextInt(right - left +1) + left;
        swap(nums,pivotIdx,left);//选择left位置
        //begin partition
        int pivot = nums[left];
        int i = left + 1;
        int j = right;

        while (true) {
            //i从左向右,遇到比pivot小的跳过,直到找到比pivot大的元素,停下
            while (i <= right && nums[i] < pivot) {
                i++;
            }
            //j从右向左,遇到比pivot大的跳过,直到找到比pivot小的元素,停下
            while (j > left && nums[j] > pivot) {
                j--;
            }
            if (i >= j) {
                break;
            }
            //交换i,j
            swap(nums,i,j);
            i++;
            j--;
        }
        //将pivot放到j的位置
```

```
81          swap(nums,left,j);
82          return j;
83      }
84      private void swap(int[] array,int i,int j) {
85          if (i == j) {
86              return;
87          }
88          array[i] ^= array[j];
89          array[j] ^= array[i];
90          array[i] ^= array[j];
91      }
92
93
94      public static void main(String[] args) {
95          int[] data = new int[10000000];
96          for (int i=0;i<10000000;i++) {
97              data[i] = new Random().nextInt(10000001);
98          }
99          System.out.print("排序前:"+ Arrays.toString(data)+"\n");
100         long start = System.currentTimeMillis();
101         new QuickSort().sortArray(data);
102         long end = System.currentTimeMillis();
103         System.out.print("排序后:"+ Arrays.toString(data)+"\n");
104         System.out.println("耗时:"+(end-start)+"ms");
105     }
106 }
```

总结

1：快速排序的时间复杂度是多少?

快排也是用递归来实现的。对于递归代码的时间复杂度，前面总结的公式，这里也还是适用的。如果每次分区操作，都能正好把数组分成大小接近相等的两个小区间，那快排的时间复杂度递推求解公式跟归并是相同的。所以，快排的时间复杂度也是 O(nlogn)。但是，公式成立的前提是每次分区操作，我们选择的 pivot 都很合适，正好能将大区间对等地一分为二。但实际上这种情况是很难实现的。我举一个比较极端的例子。如果数组中的数据原来已经是有序的了，比如 1，3，5，6，8。如果我们每次选择最后一个元素作为 pivot，那每次分区得到的两个区间都是不均等的。我们需要进行大约n 次分区操作，才能完成快排的整个过程。每次分区我们平均要扫描大约 n/2 个元素，这种情况下，快排的时间复杂度就从 O(nlogn) 退化成了 O(n2)。我们刚刚讲了两个极端情况下的时间复杂度，一个是分区极其均衡，一个是分区极其不均衡。它们分别对应快排的最好情况时间复杂度和最坏情况时间复杂度。那快排的平均情况时间复杂度是多少呢?

在此处我们发现继续套用递归时间复杂度的递推公式会变得非常的复杂，求解过程非常的繁琐，虽然可以求解出来，但是并不推荐这样去做，当然了递归的时间复杂度求解除了递推公式外，还有递归树的求解，但由于目前还未学习树，因此这里不在讨论，这里我们可以直接得出结论：**快排的时间复杂度最好以及平均情况下的复杂度都是O(nlogn)，只有在极端情况下会变成O(n2)**，当然了我们也有对应的方式来降低这种概率，像我们刚刚所说的快排在极端情况下的时间复杂度退化为O(n2)，主要原因是因为分区点选取的不够合理。那如何来选取合适的分区点呢?

最理想的分区点是：被分区点分开的两个分区中，数据的数量差不多。如果很粗暴地直接选择第一个或者最后一个数据作为分区点，不考虑数据的特点，肯定会出现之前讲的那样，在某些情况下，排序的最坏情况时间复杂度是 O(n)。为了提高排序算法的性能，我们也要尽可能地让每次分区都比较平均。这里介绍两个比较常用的方式：

- **三数取中法**：我们从区间的首、尾、中间，分别取出一个数，然后对比大小，取这 3 个数的中间值作为分区点。这样每间隔某个固定的长度，取数据出来比较，将中间值作为分区点的分区

算法，肯定要比单纯取某一个数据更好。但是，如果要排序的数组比较大，那“三数取中”可能就不够了，可能要“五数取中”或者“十数取中”。

- **随机法**：随机法就是每次从要排序的区间中，随机选择一个元素作为分区点。这种方法并不能保证每次分区点都选的比较好，但是从概率的角度来看，也不大可能会出现每次分区点都选的很差的情况，所以平均情况下，这样选的分区点是比较好的。时间复杂度退化为最糟糕的 O(n) 的情况，出现的可能性不大。

2：快速排序的空间复杂度是多少？

通过快排的代码实现我们发现，快排不需要额外的存储空间，所有的操作都能在既定的空间内完成，因此快排的**空间复杂度为O(1)**，也就是说快排是一种in-place的排序算法。

3：快速排序是稳定的排序算法吗？

因为分区的过程涉及交换操作，如果数组中有两个相同的元素，比如序列 6，8，7，6，3，5，9，4，在经过第一次分区操作之后，两个 6 的相对先后顺序就会改变。所以，**快速排序并不是一个稳定的排序算法。**

4：快排和归并的异同

首先快排和归并都用到了分治递归的思想，在快排中对应的叫分区操作，递推公式和递归代码也非常相似，但是归并排序的处理过程是由下到上的由局部到整体，先处理子问题，然后再合并。而快排正好相反，它的处理过程 是由上到下由整体到局部，先分区，然后再处理子问题。归并排序虽然是稳定的、时间复杂度为 O(nlogn) 的排序算法，但是它是一种out-place排序算法。主要原因是合并函数无法在原地(数组内)执行。快速排序通过设计巧妙的原地(数组内)分区函数，可以实现原地排序，解决了归并排序占用太多内存的问题。

截至目前我们已经学习了：冒泡，插入，选择，归并，快排；并且分析了每种排序算法的原理、时间复杂度、空间复杂度、稳定性等。就复杂度而言：冒泡，插入，选择都是O(n2)，归并和快排是：O(nlogn)；接下来我们要来学习三种时间复杂度是 O(n) 的排序算法：桶排序、计数排序、基数排序。因为这些排序算法的时间复杂度是线性的，所以我们把这类排序算法叫作**线性排序（Linear sort）**。之所以能做到线性的时间复杂度，主要原因是这三个算法是非基于比较的排序算法，都不涉及元素之间的比较操作。

实战题目

215. 数组中的第K个最大元素

```java
class Solution {
    public int findKthLargest(int[] nums, int k) {
        int len = nums.length;
        int left = 0;
        int right = len -1;
        int targetIndex = len - k;
        //开始分区
        while (true) {
            int pivotIdx = partition(nums,left,right);
            if (targetIndex == pivotIdx) {
                break;
            }else if (pivotIdx > targetIndex) {
                right = pivotIdx -1;
            }else {
                left = pivotIdx +1;
            }
```

```
        }
        return nums[targetIndex];
    }
    Random random = new Random();
    public int partition(int[] nums,int left,int right) {
        //为了让递归输相对平衡,随机选择pivot
        int pivotIdx = random.nextInt(right - left +1) + left;
        swap(nums,pivotIdx,left);
        int pivot = nums[left];
        int i= left+1;
        int j= right;
        while (true) {
            while (i<=right && nums[i] < pivot) {
                i++;
            }
            while (j>left && nums[j] > pivot) {
                j--;
            }
            if (i>=j) {
                break;
            }
            swap(nums,i,j);
            i++;
            j--;
        }
        swap(nums,left,j);
        return j;
    }

    public void swap(int[] nums,int i,int j) {
        if (i==j) {
            return;
        }
        nums[i] ^= nums[j];
        nums[j] ^= nums[i];
        nums[i] ^= nums[j];
    }
}
```

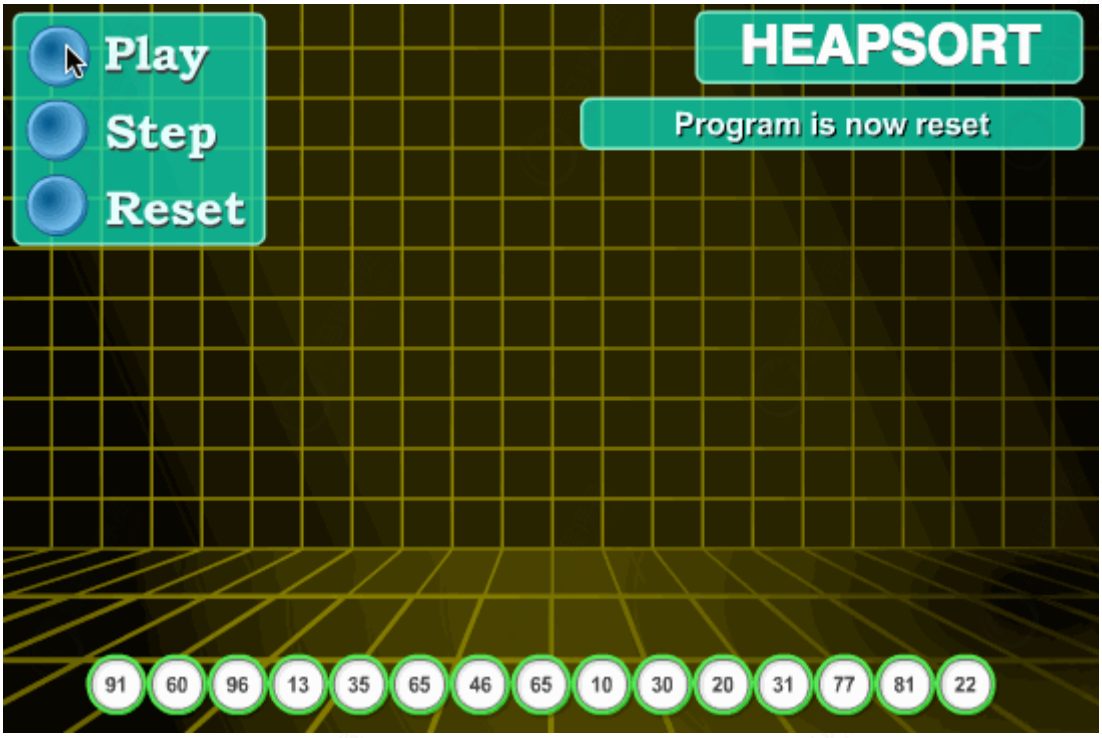
堆排序

原理

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。

算法描述如下：

- 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



实现

```java
package com.itheima.sort;

import java.util.Arrays;

/**
 * Created by 传智教育*黑马程序员.
 */
public class HeapSort {


    public int[] sortArray(int[] nums) {
        headSort(nums);
        return nums;
    }

    //step3:堆排序
    public void headSort(int[] nums) {
        int len = nums.length -1;//区间最后一个位置(下标)
        //先针对nums构建大顶堆
        buildMaxHeap(nums,len);
        /*
            将堆顶元素和堆最后一个元素交换,有序数据多了一个,堆中数据少一个,
            然后针对交换过来的堆顶元素进行堆化调整
            依次操作知道堆中只剩一个元素(只需进行len-1 次交换),怎整体数据有序
        */
        for (int i=len;i>=1;i--) {
            swap(nums,0,i);//堆中少一个,堆的边界-1
            len--;
            //堆化调整
            maxIfyDown(nums,0,len);
        }

    }

```

```java
    //step2:针对nums构建大顶堆
    public void buildMaxHeap(int[] nums,int end) {
        //为了能使用从上到下的堆化调整,我们从倒数第二层最后一个元素开始进行调整
        for (int i= (end-1)>>1;i>=0;i--) {
            maxIfyDown(nums,i,end);
        }
    }
    //step1:在区间[start,end]进行从到到下的堆化调整
    public void maxIfyDown(int[] nums,int start,int end) {
        while (start <= (end-1)>>1) {
            int lson = (start<<1) + 1;
            int rson = lson +1;
            //找自己,左孩子,右孩子中的最大值替换自己
            int large = start;
            if (lson<=end &&  nums[lson] > nums[large] ) {
                large = lson;
            }
            if (rson<=end && nums[rson] > nums[large]) {
                large = rson;
            }
            //如果最大值来自左右孩子,交换
            if (large != start) {
                swap(nums,start,large);
                //到下一层继续
                start = large;
            }else {
                break;
            }
        }

    }
    public void swap(int[] nums,int i,int j) {
        if (i==j) {
            return;
        }
        nums[i] ^= nums[j];
        nums[j] ^= nums[i];
        nums[i] ^= nums[j];
    }

    public static void main(String[] args) {
        int[] data = new int[]{5,2,6,9,0,3};
        System.out.println("排序前:"+ Arrays.toString(data));
        new HeapSort().sortArray(data);
        System.out.println("排序后:"+ Arrays.toString(data));
    }

}

```

总结

时间复杂度：最好，最坏，平均情况下都为：O(n * log n)

空间复杂度：O(1)

稳定性：非稳定的排序算法

归并-快排-堆排比较

通过如下测试代码进行测试比较：

```
package com.itheima.sort;

import java.util.Random;

/**
 * Created by 传智教育*黑马程序员.
 */
public class PerformanceTest {

    static Random random = new Random();

    public static int[] genData(int size) {
        int[] data = new int[size];
        for (int i=0;i<size;i++) {
            data[i] = random.nextInt(size+1);
        }
        return data;
    }

    //10次排序,每次排序的数据是500w,求时间的平均值
    public static void main(String[] args) {
        long sum = 0L;
        for (int i=0;i<10;i++) {
            int[] data = genData(5000000);
            long start = System.currentTimeMillis();
            new MergeSort().sortArray(data);//归并949
            sum += System.currentTimeMillis()-start;
        }
        System.out.println("归并:10次排序-每次500w数据平均耗时为:"+sum / 10L+"毫秒");


        sum = 0L;
        for (int i=0;i<10;i++) {
            int[] data = genData(5000000);
            long start = System.currentTimeMillis();
            //new MergeSort().sortArray(data);//归并949
            new QuickSort().sortArray(data);//快排 1006
            //new HeapSort().sortArray(data);//堆排 1459
            sum += System.currentTimeMillis()-start;
        }
        System.out.println("快速排序:10次排序-每次500w数据平均耗时为:"+sum / 10L+"毫秒");

        sum = 0L;
        for (int i=0;i<10;i++) {
            int[] data = genData(5000000);
            long start = System.currentTimeMillis();
            //new MergeSort().sortArray(data);//归并949
            //new QuickSort().sortArray(data);//快排 1006
            new HeapSort().sortArray(data);//堆排 1459
```

```
            sum += System.currentTimeMillis()-start;
        }
        System.out.println("堆排序:10次排序-每次500w数据平均耗时为:"+sum / 10L+"毫秒");
    }



}

```

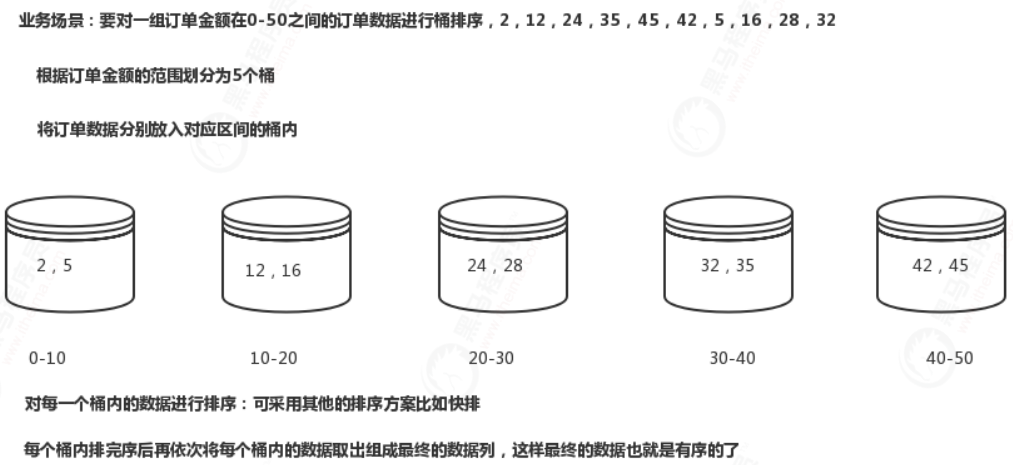
输出：仅供参考

```
归并:10次排序-每次500w数据平均耗时为:964毫秒
快速排序:10次排序-每次500w数据平均耗时为:780毫秒
堆排序:10次排序-每次500w数据平均耗时为:1451毫秒
```

桶排序

原理

桶排序(Bucket Sort)顾名思义，会用到“桶”，桶我们可以将其想象成一个容器，核心思想是将要排序的数据分到几个有序的桶里，每个桶里的数据再单独进行排序。桶内排完序之后，再把每个桶里的数据按照顺序依次取出，组成的序列就是有序的了，换句话说：桶排序是将待排序集合中处于同一个值域的元素存入同一个桶中，也就是根据元素值特性将集合拆分为多个区域，则拆分后形成的多个桶，从值域上看是处于有序状态的。对每个桶中元素进行排序，则所有桶中元素构成的集合是已排序的。



桶排序过程中存在两个关键环节：

- 元素值域的划分，也就是元素到桶的映射规则。映射规则需要根据待排序集合的元素分布特性进行选择，若规则设计的过于模糊、宽泛，则可能导致待排序集合中所有元素全部映射到一个桶上，若映射规则设计的过于具体、严苛，则可能导致待排序集合中每一个元素值映射到一个桶上。
- 从待排序集合中元素映射到各个桶上的过程，并不存在元素的比较和交换操作，在对各个桶中元素进行排序时，可以自主选择合适的排序算法，每个桶内的排序算法的复杂度和稳定性，决定了最终的算法的复杂度和稳定性

那么桶排序的时间复杂度是多少呢？我们可以建议分析一下：

如果要排序的数据有 n 个，我们把它们均匀地划分到 m 个桶内，每个桶里就有 k=n/m 个元素。假设每个桶内部使用快速排序，时间复杂度为 O(k * logk)。m 个桶排序的时间复杂度就是 O(m * k * logk)，因为 k=n/m，所以整个桶排序的时间复杂度就是 O(n*log(n/m))。当桶的个数 m 接近数据个数 n 时，log(n/m) 就是一个非常小的常量，这个时候桶排序的时间复杂度接近 O(n)。

桶排序看起来是如此的优秀，那是不是可以替代我们之前讲到的排序算法呢？答案是否定的。首先，要排序的数据需要很容易就能划分成 m 个桶，并且，桶与桶之间有着天然的大小顺序。这样每个桶内的数据都排序完之后，桶与桶之间的数据不需要再进行排序。其次，数据在各个桶之间的分布是比较均匀的。如果数据经过桶的划分之后，有些桶里的数据非常多，有些非常少，很不平均，那桶内数据排序的时间复杂度就不是常量级了。在极端情况下，如果数据都被划分到一个桶里，那就退化为 O(nlogn) 的排序算法了。

桶排序比较适合用在非内存排序中。所谓的非内存排序就是说数据存储在外部磁盘中，数据量比较大，内 存有限，无法将数据全部加载到内存中。此外由桶排序的过程可知，当待排序集合中存在元素值相差较大时，对映射规则的选择是一个挑战，有时可能导致元素集中分布在某一个桶中或者绝大多数桶是空桶的现象，对算法的时间复杂度或空间复杂度有较大影响，所以桶排序适用于元素值分布较为集中的序列，或者说待排序的元素能够均匀分布在某一个范围[MIN, MAX]之间。

接下来以一个面试题为例来说明：比如说我们有 10GB 的订单数据，我们希望按订单金额（假设金额都是正整数）进行排序，但是我们的内存有限，只有几百 MB，没办法一次性把 10GB 的数据都加载到内存中。这个时候该怎么办呢？

现在我来讲一下，如何借助桶排序的处理思想来解决这个问题。我们可以先扫描一遍文件，看订单金额所处的数据范围。假设经过扫描之后我们得到，订单金额最小是 1 元，最大是 10 万元。我们将所有订单根据金额划分到 100 个桶里，第一个桶我们存储金额在 1 元到 1000 元之内的订单，第二桶存储金额在 1001 元到 2000 元之内的订单，以此类推。每一个桶对应一个文件，并且按照金额范围的大小顺序编号命名（00，01，02...99）。

理想的情况下，如果订单金额在 1 到 10 万之间均匀分布，那订单会被均匀划分到 100 个文件中，每个小文件中存储大约 100MB 的订单数据，我们就可以将这 100 个小文件依次放到内存中，用快排来排序。等所有文件都排好序之后，我们只需要按照文件编号，从小到大依次读取每个小文件中的订单数据，并将其写入到一个文件中，那这个文件中存储的就是按照金额从小到大排序的订单数据了。

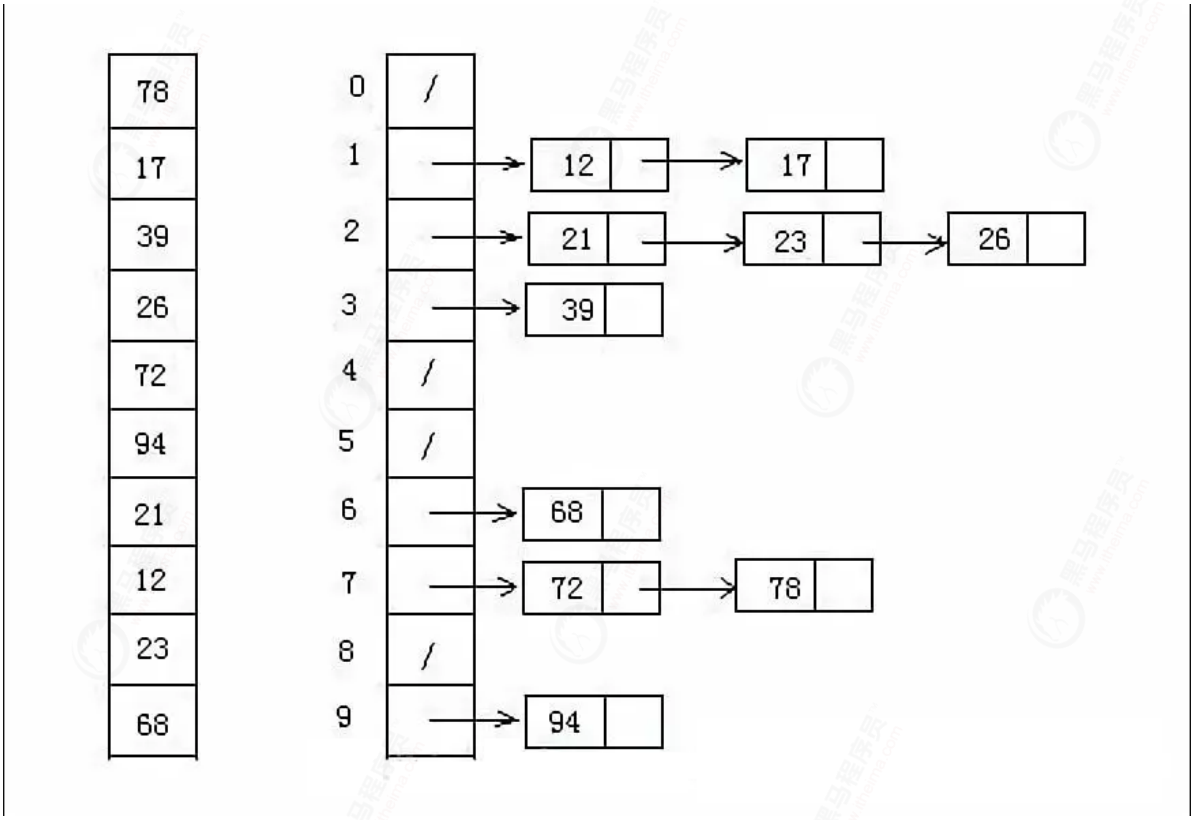
不过，你可能也发现了，订单按照金额在 1 元到 10 万元之间并不一定是均匀分布的 ，所以 10GB 订单数据是无法均匀地被划分到 100 个文件中的。有可能某个金额区间的数据特别多，划分之后对应的文件就会很大，没法一次性读入内存。这又该怎么办呢？针对这些划分之后还是比较大的文件，我们可以继续划分，比如，订单金额在 1 元到 1000 元之间的比较多，我们就将这个区间继续划分为 10 个小区间，1 元到 100 元，101 元到 200 元，201 元到 300 元...901 元到 1000 元。如果划分之后，101 元到 200 元之间的订单还是太多，无法一次性读入内存，那就继续再划分，直到所有的文件都能读入内存为止。

接下来为了能对该算法做具体的实现，我们对该算法进一步做具体的描述：

- 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。

图片演示：



实现

对该算法具体实现如下：

```
/**
 * 桶排序
 */
public class BucketSort {
    /**
     * 桶排序
     * @param array         待排序集合
     * @param bucketSize  桶中元素类型的个数即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）
     * @return              排好序后的集合
     */
    public static List<Integer> bucketSort(List<Integer> array,int bucketSize){
        if(array == null || array.size() < 2 || bucketSize < 1){
            return array;
        }
        //找出集合中元素的最大值,最小值
        int max = array.get(0);
        int min = array.get(0);
        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);
            }
        }
        //计算桶的个数    最大值-最小值代表了集合中元素取值范围区间
        int bucketCount = (max - min )/bucketSize +1;
```

```
        //按序创建桶,创建一个List,List带下标是有序的,List中的每一个元素是一个桶,也用List表示
        List<List<Integer>> bucketList = new ArrayList<>();
        for(int i=0;i< bucketCount;i++){
            bucketList.add(new ArrayList<Integer>());
        }
        //将待排序的集合依次添加到对应的桶中
        for(int j=0; j< array.size();j++){
            int bucketIndex = (array.get(j)-min)/bucketSize;
            bucketList.get(bucketIndex).add(array.get(j));
        }
        //对每一个桶中的数据进行排序(可以使用别的排序方式),然后再将桶中的数据依次取出存放到一个最终的集合中
        //创建最终的集合
        List<Integer> resultList = new ArrayList<>();
        for(int j=0;j < bucketList.size(); j++){

            List<Integer> everyBucket = bucketList.get(j);
            //如果桶内有元素
            if(everyBucket.size()>0){

                //递归的使用桶排序为每一个桶进行排序
                //当某次桶排序待排序集合都分配到一个桶中时,缩小桶的范围以获得更多的桶
                if(bucketCount ==1){
                    bucketSize--;
                }
                List<Integer> temp = bucketSort(everyBucket, bucketSize);
                for(int i=0;i<temp.size();i++){
                    resultList.add(temp.get(i));
                }
            }
        }
        return resultList;
    }

    /**
     * 测试桶排序
     */
    @Test
    public void testBucketSort(){
        List<Integer> list = new ArrayList<>();
        list.add(5);
        list.add(2);
        list.add(2);
        list.add(6);
        list.add(9);
        list.add(0);
        list.add(3);
        list.add(4);
        System.out.println(list);
        List<Integer> bucketSort = bucketSort(list, 2);
        System.out.println(bucketSort);
    }
}
```

总结

1：桶排序的时间复杂度是多少?

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，如果我们将待排序元素映射到某一个桶的映射规则做的很好的话，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。我们一般对每个桶内的元素进行排序时采用快排也可以采用递归桶排序，通过我们刚开始的分析，当我们对每个桶采用快排时如果桶的个数接近数据规模n时，**复杂度为O(n)**，如果在极端情况下复杂度退化为O(n* log n)。

2：桶排序的空间复杂度是多少?

由于需要申请额外的空间来保存元素，并申请额外的空间来存储每个桶，所以**空间复杂度为O(N+M)**，其中M代表桶的个数。所以桶排序虽然快，但是它是采用了用空间换时间的做法。

3：桶排序是稳定的排序算法吗?

桶排序是否稳定取决于对每一个桶内元素排序的算法的稳定性，如果我们对桶内元素使用快排时桶排序就是一个不稳定的排序算法。

计数排序

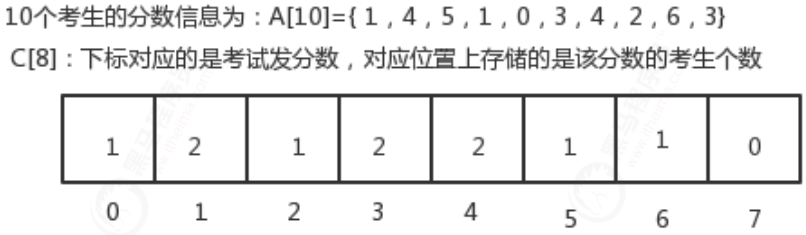
原理

计数排序(Counting Sort) 使用了一个额外的数组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。其实计数排序其实是桶排序的一种特殊情况。当要排序的 n 个数据，所处的范围并不大的时候，比如最大值是 m，我们就可以把数据划分成 m 个桶(其实是个数组)。每个桶内的数据值都是相同的，省掉了桶内排序的时间。每个桶内存储的也不是待排序的数据而是待排序数组A中值等于某个值的元素个数，接下来我们以一个例子来说明

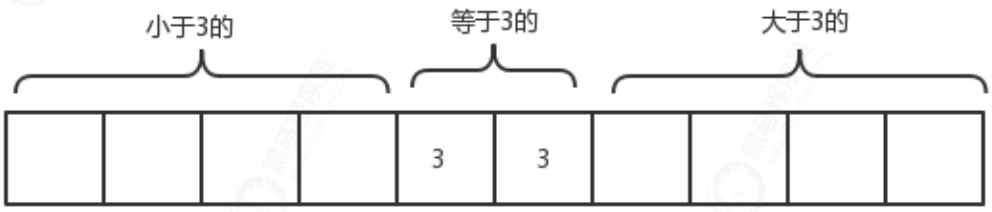
我们都经历过高考，我们查分数的时候，系统会显示我们的成绩以及所在省的排名。如果你所在的省有100 万考生，如何通过成绩快速排序得出名次呢?

我们都知道高考的满分是 750分，最小是 0 分，这个数据的范围很小，所以我们可以分成 751 个桶，对应分 数从 0 分到750分。根据考生的成绩，我们将这 100 万考生划分到这 751 个桶里。桶内的数据都是分数相同的考生，所以并不需要再进行排序。我们只需要依次扫描每个桶，将桶内的考生依次输出到一个数组中，就实现了 50 万考生的排序，那具体如何来做呢?

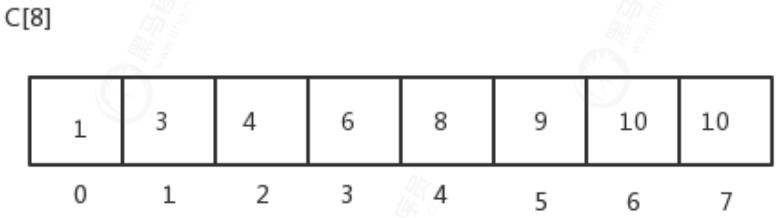
在这里为了方便理解和说明，我们假设有10考生，大家的分数在0-7分之间，这10个考生的成绩我们存放在一个数组A[10]中，分别为：1，4，5，1，0，3，4，2，6，3；因为成绩的分布是在0-7之间，我们使用一个大小为8的数组C[8]代表8个桶，数组的下标对应的是考生的分数，数组C中存储的并不是考生信息，而是对应下标分数的考生个数，我们只需要编译一遍A[10]这样就可以得到C[8]的值如下：



从图中我们可以看出：分数为3的考生有2个，小于3分的考生有4个，所以，成绩为3分的考生在最终排序好的有序数组R[10]中会保存在下标为：4，5的位置上



那接下来就是计算出每个分数的考生在最终的有序数组中的存储位置，这个处理方案很是巧妙，下面是处理思路：对数组C[8]顺序求和，就变成了下面这个样子，C[i]里存储的就是分数小于等于i的考生个数



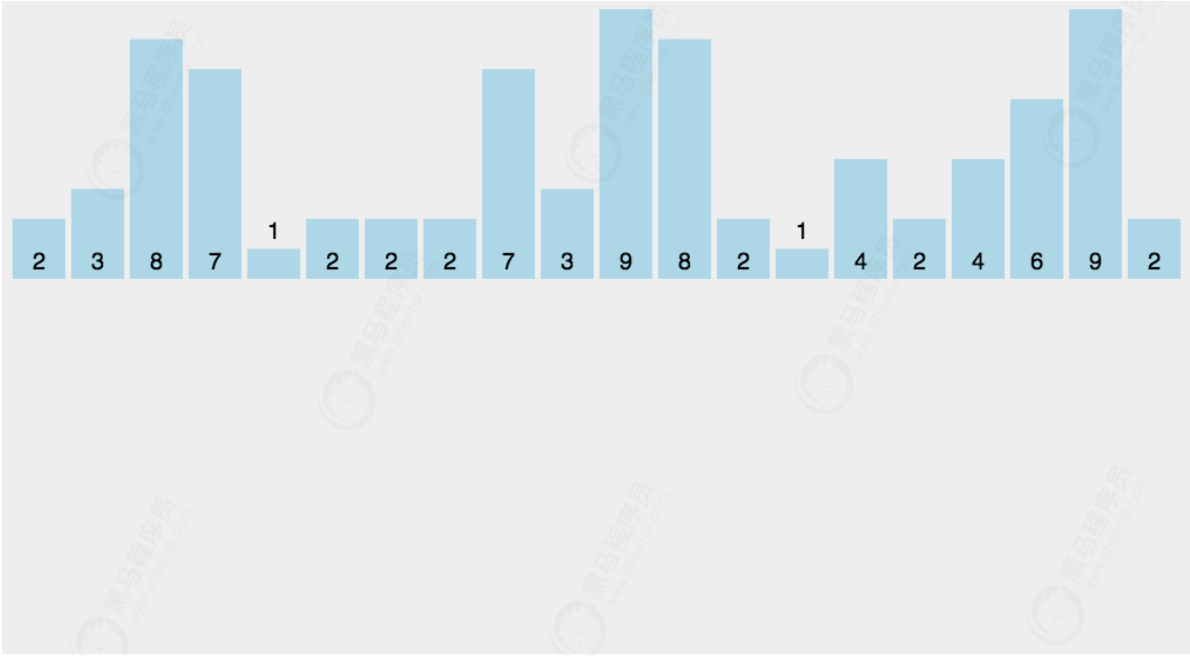
下面就是计数排序中稍微复杂度一点的地方了，我们从后到前依次扫描待排序数组 A。比如，当扫描到元素 3 时，我们可以从数组 C 中取出下标为 3 的值 6，也就是说，到目前为止，包括自己在内，分数小于等于 3 的考生有 6 个，也就是说 3 是最终有序数组 R 中的第 6个元素（也就是数组 R 中下标为5的位置）。当 3 放入到数组 R 中后，小于等于 3 的元素就只剩下了 5 个了，所以相应的 C[3] 要减 1，变成 5。

以此类推，当我们扫描到第 2 个分数为 3 的考生的时候，就会把它放入数组 R 中的第 5 个元素的位置（也就是下标为 4的位置）。当我们扫描完整个数组 A 后，数组 R 内的数据就是按照分数从小到大有序排列的了。

当然了这只是大致的思路，在代码的具体实现中我们还需要根据实际的情况来做出一些调整

动图连接：

https://mmbiz.qpic.cn/mmbiz_gif/QCu849YTaIOOdfiakqsTRHKk9icjqQZJYu0t2QuZRYMJqzEAUiaiagwpngltGHrJyegsZCwr7GpxQoRcSpTmypS3ag/640?tp=webp&wxfrom=5&wx_lazy=1



实现

计数排序实现如下：

```java
/**
 * 计数排序
 *  1:找出待排序的数组中最大和最小的元素；
    2: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；
    3: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）；
    4: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。
 */
public class CountingSort {

    public void countingSort(int[] array){
        //求出待排序数组的最大值,最小值,找出取值区间
        int max = array[0];
        int min = array[0];
        for(int i=0;i<array.length;i++){
            if( array[i] > max){
                max = array[i];
            }
            if(array[i] < min){
                min = array[i];
            }
        }
        //定义一个额外的数组C
        int bucketSize = max - min +1;
        int[] bucket = new int[bucketSize];
        //统计对应元素的个数,数组的下标不是单纯的值
        for(int i=0;i< array.length;i++){
            int bucketIndex = array[i] - min;
            bucket[bucketIndex]+=1;
        }
        //对数组C内元素进行累加
        for(int i=1;i<bucket.length;i++){
            bucket[i] = bucket[i] + bucket[i-1];
        }
        //创建临时数组R 存储最终有序的数据列表
```

```
        int[] temp = new int[array.length];
        //逆序扫描待排序数组 可保证元素的稳定性
        for(int i= array.length -1;i >=0;i--){
            int bucketIndex = array[i] - min;
            temp[bucket[bucketIndex]-1]  = array[i];
            bucket[bucketIndex] -=1;
        }
        /*for(int i= 0;i < array.length;i++){
            int bucketIndex = array[i] - min;
            temp[bucket[bucketIndex]-1]  = array[i];
            bucket[bucketIndex] -=1;
        }*/
        //将临时数据列表依次放入原始数组
        for(int i=0;i<temp.length;i++){
            array[i] = temp[i];
        }
    }

    /**
     * 测试
     */
    @Test
    public void testCountingSort(){
        //准备一个int数组
        int[] array = new int[8];
        array[0] = 5;
        array[1] = 2;
        array[2] = 6;
        array[3] = 9;
        array[4] = 0;
        array[5] = 3;
        array[6] = 3;
        array[7] = 4;
        //进行排序
        System.out.println(Arrays.toString(array));
        countingSort(array);
        //输出排序结果
        System.out.println(Arrays.toString(array));
    }
}
```

总结

1：计数排序的时间复杂度是多少?

通过代码的实现过程我们发现计数排序不涉及元素的比较，不涉及桶内元素(数组C)的排序，只有对待排序数组和用于计数数组的遍历操作，因此**计数排序的时间复杂度是O(n+k)，其中k是桶的个数即待排序的数据范围**，是一种线性排序算法。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k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。

2：计数排序的空间复杂度是多少?

在计数排序的过程中需要创建额外的桶空间(数组C)来计数，因此我们可以得知 **计数排序的口空间复杂度为：O(n+K)，其中n是数据规模大小，K是计数排序中需要的桶的个数**，其实也就是用来计数的数组C的长度，之前我们提到过它取决于待排序数组中数据的范围。

3：计数排序是稳定的排序算法吗?

在计数排序中核心操作中我们是逆序的去扫描待排序数组，这样仍然可以使待排序数组中值相同但是位置靠后的元素在最终的已排序数组中保持着相同的位置关系，因此**计数排序是一个稳定的排序算法。**

4：计数排序的适用场景?

计数排序只能用在数据范围不大的场景中，如果数据范围 k 比要排序的数据 n 大很多，就不适合用计数排序了。而且，计数排序只能给非负整数排序，如果要排序的数据是其他类型的，要将其在不改变相对大小的情况下，转化为非负整数。

比如，还是拿分数这个例子。如果分数精确到小数后一位，我们就需要将所有的分数都先乘以10，转化成整数，然后再放到桶内。再比如，如果要排序的数据中有负数，数据的范围是[-100, 100]，那我们就需要先对每个数据都加 100，转化成非负整数。

基数排序

原理

基数排序(Radix Sort)也是非比较的排序算法，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 。以LSD为例我们可以先按照低位排序，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。

我们以一个例子来说明基数排序，假设我们有 100万个手机号码，希望将这 100万个手机号码从小到大排序，你有什么比较快速的排序方法呢?

如果使用我们之前讲的快排，时间复杂度可以做到 O(nlogn)，咱们今天所讲的O(n)复杂度的桶排序、计数排序可以使用吗？手机号码有 11 位，范围太大，显然不适合用这两种排序算法。针对这个排序问题，有没有时间复杂度是 O(n) 的算法呢?

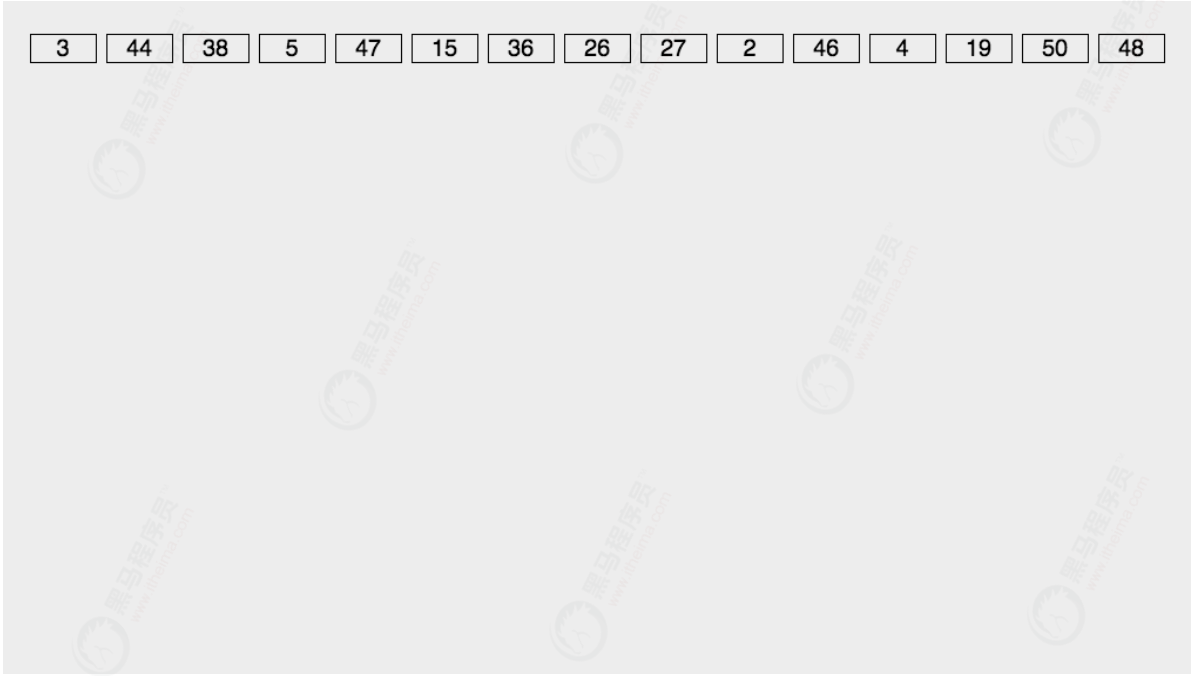
现在我们就来介绍如何使用基数排序解决刚刚这个问题。这个问题里有这样的规律：假设要比较两个手机号码 a，b 的大小，如果在前面几位中，a 手机号码已经比 b 手机号码大了，那后面的几位就不用看了。借助稳定排序算法，这里有一个巧妙的实现思路。先按照最后一位来排序手机号码，然后，再按照倒数第二位重新排序，以此类推，最后按照第一位重新排序。经过 11 次排序之后，手机号码就都有序了。根据每一位来排序，我们可以用刚讲过的桶排序或者计数排序，它们的时间复杂度可以做到 O(n)。如果要排序的数据有 k 位，那我们就需要 k 次桶排序或者计数排序，总的时间复杂度是 O(k*n)。当k 不大的时候，比如手机号码排序的例子，k 最大就是 11，所以基数排序的时间复杂度就近似于 O(n)。实际上，有时候要排序的数据并不都是等长的，那这个时候就需要对不够位数的元素进行补位，比如位数不够可以高位补0，因为补0不影响原有大小顺序，补足位数之后可以继续使用基数排序了。

如果我们要用该排序算法对一个数组中的数据进行排序，我们先对该算法进行一个描述：

- 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）；

动图连接：

https://mmbiz.qpic.cn/mmbiz_gif/QCu849YTaIOOdfiakqsTRHKk9icjqQZJYueqC3UMMkJF6NJjLNIm8XicOJhLI7FiadqfV7saxBAPM0Tmd7LBSicOr9g/640?tp=webp&wxfrom=5&wx_lazy=1



实现

对一个int数组进行基数排序的实现如下：

```
/**
 * 基数排序
 *  1：取得数组中的最大数，并取得位数；
    2：arr为原始数组，从最低位开始取每个位组成radix数组；
    3：对radix进行计数排序（利用计数排序适用于小范围数的特点）；
 */
public class RadixSort {

    /**
     * 基数排序
     * @param array
     * @return
     */
    public 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;
    }


    @Test
    public void testRadixSort(){
        //准备一个int数组
        int[] array = new int[9];
        array[0] = 25;
        array[1] = 2;
        array[2] = 36;
        array[3] = 9;
        array[4] = 10;
        array[5] = 3;
        array[6] = 4;
        array[7] = 3;
        array[8] = 135;
        //进行排序
        System.out.println(Arrays.toString(array));
        array = radixSort(array);
        //输出排序结果
        System.out.println(Arrays.toString(array));
    }
}
```

总结

1：基数排序的时间复杂度是多少?

通过之前的分析可知，基数排序的时间复杂度是O(k* n)，其中n是数据规模，k是待排序数列中数据的最大长度，所以当k不是特别大的时候，**基数排序的时间复杂度就近似于O(n)。**

2：基数排序的空间复杂度是多少?

基数排序也需要额外的存储空间，因此**基数排序的空间复杂度是：O(n+k)，其中n是数据规模，k是待排序数列中数据的最大长度。**

3：基数排序是稳定的排序算法吗?

通过基数排序的实现过程可知，在对每一位进行排序时我们采用的都是稳定的排序算法，因此**基数排序也是稳定的排序算法。**

4：基数排序是适用场景?

基数排序对要排序的数据是有要求的，需要可以分割出独立的“位”来比较，而且位之间有递进的关系，如果 a 数据的高位比 b 数据大，那剩下的低位就不用比较了。除此之外，每一位的数据范围不能太大，要可以用线性排序算法来排序，否则，基数排序的时间复杂度就无法做到 O(n) 了。

小结

接下来我们以一幅图的形式来总结一下各种排序算法。

排序算法	平均情况复杂度	最好情况复杂度	最坏情况复杂度	空间复杂度	排序方式	稳定性	备注
冒泡排序	O(n2)	O(n)	O(n2)	O(1)	in-place	稳定	
选择排序	O(n2)	O(n2)	O(n2)	O(1)	in-place	不稳定	
插入排序	O(n2)	O(n)	O(n2)	O(1)	in-place	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(n2)	O(1)	in-place	不稳定	
桶排序	O(n)	O(n)	O(n*log n)	O(n+k)	out-place	不稳定	桶排序的复杂度和稳定性取决于用何种排序算法为每一个桶进行排序，在此以快排为例为每个桶进行排序，k为桶的个数
计数排序	O(n+k)	O(n+k)	O(n+k)	O(n+k)	out-place	稳定	其中n为数据规模，k为计数排序中需要的桶的个数，其实也就是用来计数的数组C的长度，它取决于待排序数组中数据的范围
基数排序	O(n*k)	O(n*k)	O(n*k)	O(n+k)	out-place	稳定	其中n是数据规模，k是待排序数列中数据的最大长度，当k不是特别大时基数排序的时间复杂度近似为线性时间复杂度

进阶题目：

147. 对链表进行插入排序

905. 按奇偶排序数组

922. 按奇偶排序数组 II

451. 根据字符出现频率排序

1122. 数组的相对排序

面试题 10.01. 合并排序的数组

148. 排序链表

912. 排序数组

剑指 Offer 51. 数组中的逆序对

剑指 Offer 45. 把数组排成最小的数