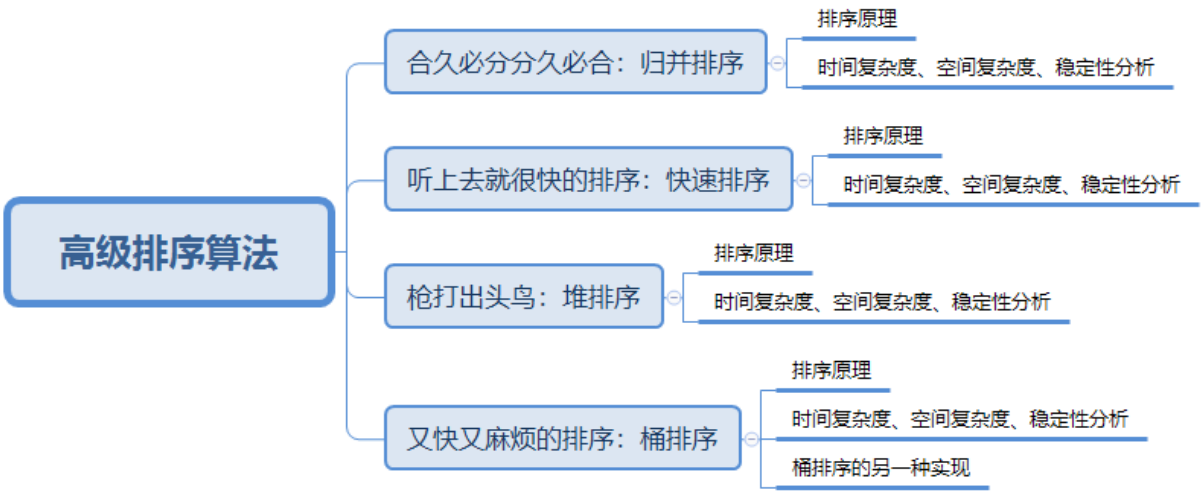


0.回忆杀

首先我们一起来回忆一下之前我们学过的4种基本排序算法：

冒泡排序：最基本的排序算法，排序方式是相邻位比较，反序则互换，时间复杂度是$O(n^2)$，空间复杂度是O(1)，一种稳定的排序算法

选择排序：冒泡排序的升级版，排序方式是在数组中每一次都选定一个标准位，遍历待排序序列，将待排序序列中的最小值放在标准位上，时间复杂度是$O(n^2)$，空间复杂度是O(1)，一种不稳定的排序算法

插值排序：类似于打扑克抽牌的排序算法，排序方式是在保证前面的n个元素是有序的基础上，将拿到的当前元素不断与前面的一个元素比较，反序则互换，直到这个元素落到一个准确的位置为止

时间复杂度是$O(n^2)$，空间复杂度的O(1)，一种稳定的排序算法

希尔排序：一种基于插值排序的排序算法，也是第一个时间复杂度小于$O(n^2)$的排序算法，使用缩小增量法，每一次都让待排序序列接近有序状态，并最终使用一个插值排序对序列进行排序

能够最大限度发挥插值排序的优势，时间复杂度是$O(n^k)$ (1.3 <= k <= 2)，空间复杂度是O(1)，一种不稳定的排序算法

通过上面的回忆，我们发现这4种基本排序算法的时间复杂度都相对比较高，即使是希尔排序算法，它的时间复杂度在最坏情况下也是趋近于$O(n^2)$的

所以，对于排序算法来说，还有提升的空间

接下来我们就来一起研究以下4种高级排序算法，他们的时间复杂度都在$O(n^2)$之下，甚至有的排序算法能够接近最小时间复杂度O(n)

他们分别是：归并排序、快速排序、堆排序、桶排序

1.合久必分分久必合：归并排序

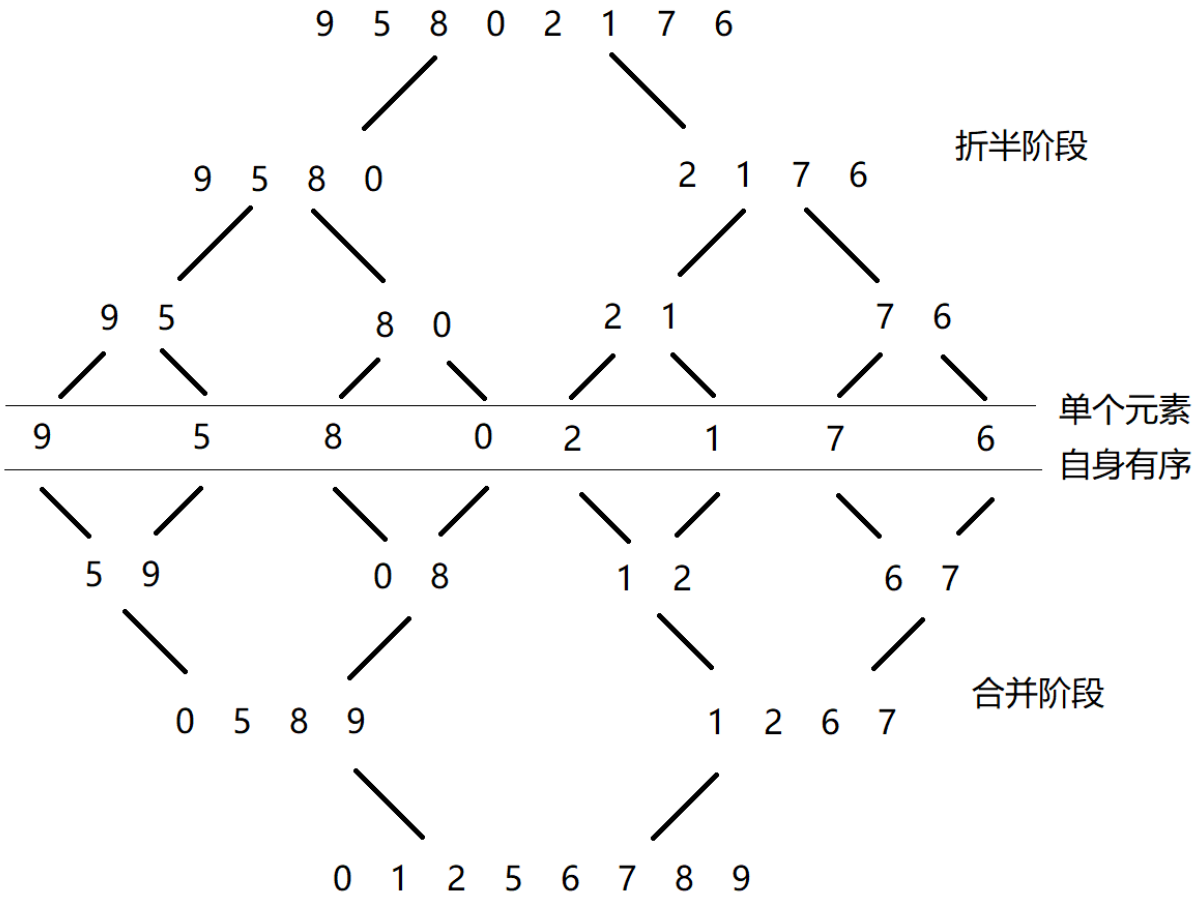
①排序原理

归并排序算法是一种利用了分治算法思想实现的排序算法（关于分治算法的内容，详见第四节的非必读部分），这种排序算法是基于有序数组合并的

归并排序算法的基本思想就是：判断一个待排序序列的长度，只要这个待排序序列的长度是超过1的，那么就将这个待排序序列进行折半

直到一个待排序序列的长度为1的时候，折半停止，并向上进行回溯。回溯过程相当于两个有序数组进行合并的过程。

下面我们通过一张图片来了解归并排序算法的流程



从上述流程图我们可以总结出如下归并排序的规律：

1.在折半阶段中，只要这个序列被拆分的任何一部分的长度依然是大于1的，那么就继续拆分下去

2.一个长度为n的序列，最终会被拆分为n个序列，并且这n个序列的长度都是1，在这n个序列中，单个元素自身是有序的

也就是说，当长度为n的序列被拆分成为n个子序列的时候，这n个子序列都是有序的

3.合并阶段即排序阶段，每一次合并操作都相当于一次有序数组合并，从n个长度为1的有序序列开始，每一次合并都是如此

有序序列合并的时间复杂度是非常低的，归并排序算法也就是通过这种方式降低了自身的时间复杂度

4.从上面的结构可以看出，在对整体序列进行拆分的时候，对于打的序列和对于相对较小的子序列的操作过程是一致的，只是序列的规模变成了原来的二分之一

那么这就说明，拆分阶段的操作，是可以使用递归的结构实现的

5.在合并阶段，我们不得不使用一个额外的长度为n的序列来记录每一次有序序列合并的结果，并将这个结果保存回原始序列中

根据上述规律，我们可以得到如下通过递归实现的归并排序算法代码：

```
/**
 * 归并排序的外壳，在这个方法内部会创建一个临时数组，
 * 用来为内部真正的归并排序提供辅助空间
 * @param array 待排序数组
 */
public void mergeSort(int[] array) {

    //创建辅助空间数组
    int[] tmp = new int[array.length];

    //开始真正的归并排序
    mergeSortInner(array, 0, array.length-1, tmp);

}

/**
 * 使用递归实现的归并排序
 * @param array 待排序数组
 * @param start 数组元素进行归并排序的起点下标
 * @param end 数组元素进行归并排序的终点下标
 * @param tmp 用来作为归并排序操作辅助空间的临时数组，
 *            临时数组长度等于原始数组长度，并且是一个空数组
 */
private void mergeSortInner(int[] array, int start, int end, int[] tmp) {

    //[1]首先判断，如果数组进行归并排序的起点和终点之间，
    //所包含的元素数量不是1，那么就继续划分
    if(end - start > 0) {
        int middle = (start + end) / 2;
        //递归调用，分别对左右两个部分进行归并排序
        mergeSortInner(array, start, middle, tmp);
```

```
        mergeSortInner(array, middle+1, end, tmp);

        //[2]在对左右两个部分分别进行归并排序之后，
        //左右两个部分都已经是有序的了，将左右两个部分进行有序数组合并
        for(int i = start; i <= middle; i++) {  //将左半部分拷贝到临时空间中
            tmp[i] = array[i];
        }
        for(int i = middle+1; i <= end; i++) {  //将右半部分拷贝到临时空间中
            tmp[i] = array[i];
        }

        //对左右两半部分的数组进行有序数组合并操作，最终合并结果合并到原始数组中
        int left = start;  //控制左半部分有序数组拷贝的下标变量
        int right = middle+1;  //控制右半部分有序数组拷贝的下标变量
        int index = start;  //控制原始数组array中，有序部分合并的下标

        while(left <= middle && right <= end) {
            if(tmp[left] < tmp[right]) {
                array[index] = tmp[left];
                left++;
            }else {
                array[index] = tmp[right];
                right++;
            }
            //不管是哪一边的元素落在原始数组中，原始数组的合并下标都要加一
            index++;
        }

        //不管是左半部分没有合并完成，还是右半部分没有合并完成，
        //都将剩余的元素全部直接落在原始数组合并完部分的后面即可
        if(left <= middle) {  //左边没有合并完成
            while(left <= middle) {
                array[index++] = tmp[left++];
            }
        }

        if(right <= end) {  //右半部分没有合并完成
            while(right <= end) {
                array[index++] = tmp[right++];
            }
        }

    }else if(end - start == 0) {
```

```
75          //[3]递归出口：如果待排序数组部分的长度是1，
76          //说明此时的待排序部分只有1个元素，
77          //那么一个元素和自己本身是有序的，此时不需要继续划分了
78          return;
79      }
80
81  }
```

②时间复杂度、空间复杂度、稳定性分析

1.归并排序时间复杂度

首先在分析归并排序算法的时间复杂度之前，我们需要声明一个问题：将两个长度分别为a和b的有序序列进行合并，时间复杂度为O(a+b)

然后我们根据上面的这个规律，对归并排序算法的时间复杂度进行分析：

从上面的图示可见，在合并阶段的每一层，都要将多个有序序列进行合并，但是巧合的是，这多个有序序列的总长度都是n，所在合并阶段每一层的时间复杂度都是n

那么合并阶段总共有多少层呢？我们可以尝试找到一些规律：

当待排序序列的长度是8的时候，我们恰好将有序序列分为3层；如果有序序列中只有4个元素，那么只要两层就可以了；如果有序序列中只有两个元素，一层合并就能够搞定

通过这些“巧合”的数据我们可以得出：当待排序序列的长度是n的时候，合并阶段的层数为$\log_2 n$，即logn

综上所述：合并阶段每一层的时间复杂度是n，并且总共有logn层

所以：归并排序算法的时间复杂度是O(nlogn)

2.归并排序空间复杂度

上文中我们说到，在对有序序列进行合并的时候，我们不得不使用一个长度同样为n的序列来保存每一次合并时有序数组的状态

那么我们有两种选择：

1.在每一次合并的时候都创建两个序列，分别存放两个有序子序列的内容，合并完成的结果存回原始序列中

2.总共给定一个与原始序列等长的辅助空间序列（比如一个空数组），将重复利用这个空序列，将每一次有序合并的两个序列都存放在这个空序列的指定位置上

考虑到实际开发中，在Java中new对象的操作是十分消耗时间和空间的，所以我们选择了第二种方式进行操作

所以：归并排序算法的空间复杂度是O(n)

3.归并排序的稳定性

在归并排序算法的合并阶段，因为每一次合并都是在原始序列中相邻的一部分元素进行合并

所以不可能出现等值元素之间，相对位置发生变化的情况

所以：归并排序是一种稳定的排序算法

2.听上去就很快的排序：快速排序

①排序原理

快速排序算法的思想是这样的：首先我们使用两个下标变量i和j，分别指向待排序序列的起点（i）和终点（j）

然后，我们使用j变量逐步向前移动，并在移动过程中找到第一个比i下标指向元素取值更小的元素，此时j变量停下脚步，i位置上的元素和j位置上的元素进行互换

互换完毕后，换i变量向后移动，同时在移动过程中找到第一个比j变量指向元素更大的值，当找到这个值得时候，i变量停下脚步，i位置上的元素和j位置上的元素进行互换

之后重复上面的两个步骤，变量i和j交互相对移动，直到ij碰面为止。

然后以ij碰面的位置为中间点，将待排序序列一分为二，重复执行上面的步骤

相信各位同学在听了上面的说明应该已经一脸蒙圈状了（请原谅我的词穷|||Orz）

下面我们直接通过一张图片演示的快速排序算法的流程来进行观察，并总结出相关的规律

通过上面的图示我们可以观察得到如下几条比较重要的规律：

1.每次排序，都是下标j首先开始从后向前移动（为什么？）

2.当下标j没有遇见比下标i指向元素更小的元素的时候，j继续移动；同理，当下标i没有遇见比下标j指向元素更大的元素的时候，i继续移动

3.当ij两个下标碰面的时候，ij两个下标共同指向的元素，实际上已经“归位”了，也就是说，通过ij两个下标指向元素的不断交换

在ij碰面的时候，ij共同指向的元素此时的下标，正好是在排序完成之后这个元素应该在的位置

4.当ij碰面的时候，以ij为分界线，ij左边的元素取值都比ij位置上的元素小；ij右边的元素都比ij位置上的元素大

5.在ij碰面之后，对待排序序列进行折半，对折半后两侧的元素进行排序的操作，实际上和对整个序列进行的排序操作方式是相同的

直到折半之后，左右只剩下一个元素，或者没有剩下任何元素为止。也就是说，这个折半和排序的流程，可以使用递归实现

通过上面对上面规律的分析，我们可以得到如下的快速排序代码实现：

```
/**
 * 使用递归结构实现的快速排序
 * @param array 待排序数组
 * @param start 待排序部分的起点
 * @param end 待排序部分的终点
 */
public void quickSort(int[] array, int start, int end) {

    if(end - start > 0) {  //要进行排序的部分中，包含多于一个元素的时候

        //[1]先将中间元素归位
        int i = start;
        int j = end;
        int tmp = 0;  //用来进行交换的临时变量
        while(i < j) {

            while(i < j && array[j] >= array[i]) {
                j--;
            }

            if(i < j) {
                tmp = array[i];
                array[i] = array[j];
                array[j] = tmp;
            }

            while(i < j && array[i] <= array[j]) {
                i++;
            }

            if(i < j) {
                tmp = array[i];
                array[i] = array[j];
                array[j] = tmp;
            }

        }

```

```
39          //[2]当上述循环结束的时候，也就是ij碰面的时候，
40          //说明ij共同指向的元素已经归位了，下面只要根据归位元素，
41          //将数组分成左右两个半侧，分别执行快速排序即可
42          int middle = i;
43  //      int middle = j;  //此时ij相等，中间下标等于谁都是一样的
44
45          //递归调用：分别对左右两个部分的数组元素再次进行快速排序
46          quickSort(array, start, middle-1);  //左半部分快速排序
47          quickSort(array, middle+1, end);  //右半部分快速排序
48
49      }else if(end - start <= 0) {
50          //递归出口：如果待排序序列中的元素数量仅剩1个或者没有元素，
51          //那么依然是一个元素自己和自己有序，不需要继续分解
52          return;
53      }
54
55  }
```

②时间复杂度、空间复杂度、稳定性分析

1.快速排序时间复杂度

通过对快速排序折半过程的分析，我们可以得到如下两点规律：

1.每一轮折半，都能够将一个元素归位

2.一个长度为n的序列能够折半多少回呢？答案是$\log_2 n$次，即logn

所以通过上面两条规则相乘，我们能够推导出，快速排序的时间复杂度是O(nlogn)

注意：值得一说的是，如果在多线程环境下，对每一次折半之后的左右两个部分，在执行递归的时候，分别使用一个线程进行操作

能够大大提升快速排序操作的效率，但是同时也涉及到了线程安全性的问题，有兴趣的同学可以研究一下Java中的Fork/Join多线程操作框架

2.快速排序空间复杂度

在快速排序的折半过程中，我们只要使用一个临时变量，用于ij下标指向元素的的互换即可

所以：快速排序算法的空间复杂度是O(1)

3.快速排序的稳定性

快速排序算法在执行ij相向而行的过程中，很有可能将两个同值的元素的相对顺序进行改变

所以：快速排序是一种不稳定排序

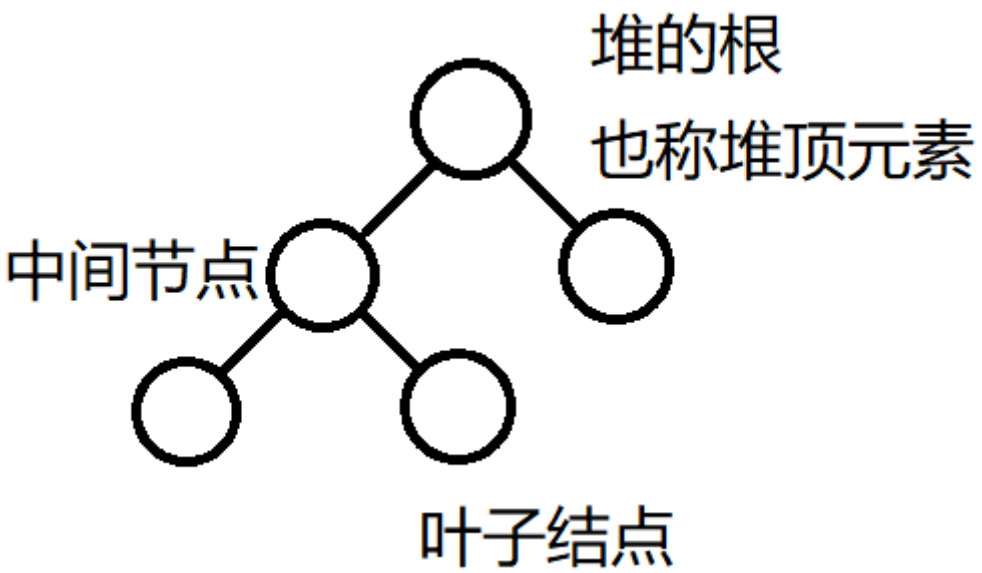
3.枪打出头鸟：堆排序

①排序原理

堆，是一种类似于二叉树的结构。

也就是说，在堆中，每一个待排序序列的元素都可以看做是一个堆的节点

而堆的每一个节点，又有两个子节点，堆结构如下图所示：



我们称图中任何一个节点下方的左右两个分支节点为其左右孩子节点，这个节点本身称之为其左右孩子节点的父节点

我们通过不断调整堆的结构，能够得到大根堆（用于升序排序）或者小根堆（用于降序排序）

也就是说：

大根堆：堆顶元素是整个堆中取值最大的元素

小根堆：堆顶元素是整个堆中取值最小的元素

然后我们通过将堆顶元素与堆中最后的元素进行互换，完成对整个堆的排序操作

有的同学可能会说：我并不会操作二叉树，那么我能够写出来堆排序吗？

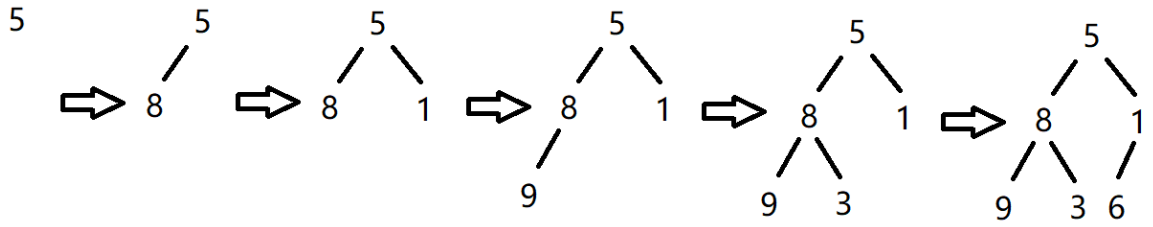
答案是肯定能啊！因为我们使用Java代码实现的堆排序算法并没有真的用到二叉树结构，堆而是使用一组下标关系公式来模拟二叉树结构

所以只要掌握了这几个公式，就能够完成通过一个序列（比如说数组）实现的“虚拟堆”的操作

下面我们将堆排序算法的步骤进行分解，来演示一组堆排序的操作流程：

步骤1：构建堆。也就是将待排序序列中的元素，逐一加入堆结构中，这个过程相当于广度优先创建一个二叉树的过程

构建堆结构，待排序序列：5　8　1　9　3　6

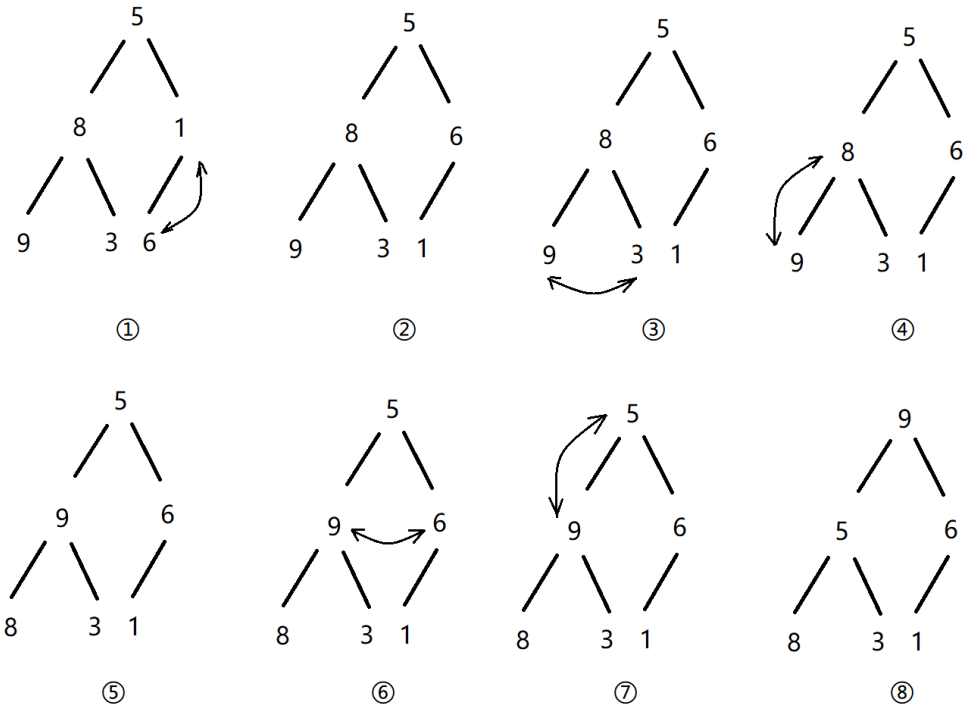


步骤2：从最后一个根节点开始，从这个根节点的左右两个孩子节点中找到一个比较大的，然后和父节点进行比较

如果这个孩子比父节点取值更大，那么这个孩子和父节点进行互换，否则不动；重复上述过程，直到最后一个根节点为止

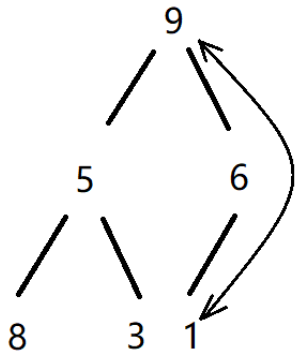
实际上堆顶元素就是最后一个根节点，当对最后一个根节点（堆顶元素）完成上述操作之后，堆顶元素就是整个堆结构中取值最大的元素

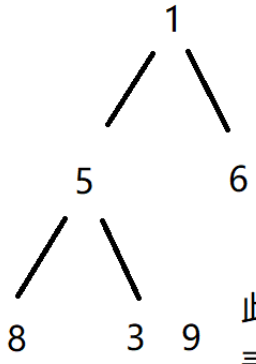
此时我们称这个堆成为了一个大根堆



步骤⑧完成后，整个序列中最大的元素9升至堆顶，成为堆顶元素，此时我们称这个堆为大根堆

步骤3：将堆顶元素和堆中的最后一个元素互换，并断开互换后最后一个元素和堆的链接，此时我们就完成了对待排序序列中最大元素的排序，此时，一个元素就这样有序了。





此时元素9相当于已经从堆结构中移除
表示元素9在序列中已经有序

然后不断重复步骤2与步骤3，直到堆中仅剩余一个元素为止，堆排序完成

上面我们演示了一轮堆排序算法的流程，下面我们通过这个流程，总结出一些堆排序的规律

尤其重要的是，我们要通过上面演示的流程，总结出在序列（比如数组结构）中，用来描述堆结构的两个公式：

1.在堆排序过程中，创建堆结构的流程我们仅执行一次，如果我们使用一个线性存储结构用来描述堆结构

那么这个过程实际上是不需要通过代码实现的，也就是说，这个流程本身只是存在于思想结构和理论过程中的

2.构建大根堆的流程，每执行一次，都能够将当前堆结构中的最大元素升至堆顶，也就是说，每构建一次大根堆，就能够使得一个元素有序

所以，如果一个序列中存在n个元素，那么我们只要执行n-1次构建大根堆的过程就可以了

3.在堆结构中，一个中间节点，要么同时存在左右两个孩子，要么只有一个左孩子，不可能左右孩子同时不存在

因为如果一个节点左右孩子都不存在，那么这个节点就不能够称之为中间节点了，只能够称之为叶子结点

4.我们在创建堆的过程中，实际上就相当于遍历了整个序列，然后将序列中的元素逐个放入堆结构中，所以实际上堆结构中的元素，

按照从上到下，从左到右的顺序，就是序列中元素的顺序

5.根据上面一条规律，我们能够直接推断出来，在堆结构中，具有所有中间节点（即具有左右孩子的节点，包括堆的根节点）所在下标的规律：

中间节点在长度为n的序列中下标的最大值是：n / 2 -1，并且对这个值向下取整

6.因为在堆中，每一个中间节点最多能够具有两个孩子节点，所以如果某一个中间节点如果在序列中的下标为k，那么其左右孩子节点与这个父节点的下标关系为：

左孩子节点下标 = 2k + 1

右孩子节点下标 = 2k + 2

7.通过上面总结的中间节点最大下标和父节点与两个孩子节点的下标关系，我们可以推理出如何判断一个节点是否具有右孩子，即

只要右孩子下标：2k + 2 <= n / 2 + 1，那么说明这个下标为k的父节点是具有右孩子的

8.根据规律4我们可以知道，在堆结构中，按照从上到下、从左到右的顺序，堆结构中的节点的顺序就是序列中节点的顺序

所以，在构建一个大根堆之后，我们在将堆顶元素和堆中最后一个元素进行互换的时候，就是将序列中下标为0的元素和待排序序列中的最后一个元素的互换过程

即序列中下标为0的元素和下标为n / 2 - 1的元素之间的互换过程

9.当序列中的一个元素有序之后，我们只要将待排序序列的最大下标，即n / 2 - 1向前提一位，即-1操作，就可以保证在这个序列中最后的元素，不会再参与到对排序之中

根据如上总结出的所有特点，我们可以得到如下的堆排序的代码实现

```
/**
 * 堆排序中用来构建大根堆的方法
 * @param array 待排序数组
 * @param end 排序范围的终点
 */
private void maxHeap(int[] array, int end) {

    //[1]根据数组的排序范围，计算出最后一个根节点的下标，
    //计算公式：lastFather = (start + end) / 2 - 1，
    //并且(start + end) / 2向上取整
    int lastFather = (0 + end) % 2 == 0 ? (0 + end) / 2 - 1 : (0 + end) / 2;

    //[3]创建一个循环，对数组中所有的根节点都进行如下操作
    for(int father = lastFather; father >= 0; father--) {
        //[2]使用每一个父节点的两个子节点先比较大小，
        //然后用两个子节点中比较大的一个，和根节点比较大小，
        //如果这个子节点比根节点还要大，则互换
        /*
         * 左右孩子节点下标和根节点下标之间的关系公式：
         * leftChild = father * 2 + 1;
         * rightChild = father * 2 + 2;
         */
        int leftChild = father * 2 + 1;
        int rightChild = father * 2 + 2;

        //如果右孩子存在并且右孩子比父节点大，那么由右孩子替换父节点
```

```
        if(rightChild <= end && array[rightChild] > array[father]) {
            int tmp = array[rightChild];
            array[rightChild] = array[father];
            array[father] = tmp;
        }

        //如果左孩子比父节点大，那么由左孩子替换父节点，
        //等价于左孩子比右孩子大，用右孩子替换原有的父节点
        if(array[leftChild] > array[father]) {
            int tmp = array[leftChild];
            array[leftChild] = array[father];
            array[father] = tmp;
        }

    }

}

/**
 * 堆排序算法
 * @param array 待排序数组
 */
public void heapSort(int[] array) {

    //[3]创建一个循环，控制数组的待排序部分的最后下标位
    for(int end = array.length-1; end > 0; end--) {
        //[1]每次都是自顶向下构建大根堆
        maxHeap(array, end);

        //[2]将大根堆的堆顶元素和数组待排序范围内的最后一个元素进行互换
        int tmp = array[0];
        array[0] = array[end];
        array[end] = tmp;
    }

}
```

②时间复杂度、空间复杂度、稳定性分析

1.堆排序时间复杂度

想要总结堆排序的时间复杂度问题，我们需要从序列的长度和在堆中元素进行互换的次数入手并找到相关的规律

在上面的一个案例中，序列的长度为8，那么在这个长度为8的序列构建的堆结构中，我们总共进行了3次互换，分别是：

6和1互换，9和8互换，9和5互换，总共3次

当序列中的元素数量减为7个的时候，构建大根堆可能发生互换的次数变为2次；

当序列中的元素数量减为6个的时候，构建大根堆可能发生互换的次数同样为2次；

当序列中的元素数量减为5个的时候，构建大根堆可能发生互换的次数还是2次；

当序列中的元素数量减为4个的时候，构建大根堆可能发生互换的次数还是2次；

当序列中的元素数量减为3个的时候，构建大根堆可能发生互换的次数变为1次；

……

通过这个规律我们不难发现，当一个堆结构重新编程大根堆的时候，堆中的一个元素有序，而这个过程，需要进行$\log_2 n$次互换，即logn次互换

如果一个序列中存在n个元素，那么我们需要构建n-1次大根堆才能够堆序列中所有的元素进行排序，

所以堆序列整体进行排序的时间复杂度是logn*(n-1) = nlogn - logn

在省略表达式中幂次较低的项logn之后，最终得到堆排序的时间复杂度是O(nlogn)

注意：堆排序的时间复杂度在理论上来讲是O(nlogn)，但是在实际编程过程中，因为我们需要在堆排序过程中实现更多的比较的代码

所以堆排序的实际表现并不是很好，根据实际操作推断，通过Java代码实现的堆排序算法的实际时间复杂度接近$O(n^2)$

2.堆排序空间复杂度

在进行堆排序的过程中，只需要使用一个临时变量，在子节点和父级节点之间进行互换的时候使用即可

所以：堆排序算法的空间复杂度是O(1)

3.堆排序的稳定性

堆排序算法因为在子节点和父节点进行互换的时候，可能造成不相邻的两个等值节点之间的顺序会发生层级变化

最终导致这两个等值节点在序列中的相对位置发生变化

所以：堆排序是一种不稳定的排序算法

4.又快又麻烦的排序：桶排序

①排序原理

桶排序算法是一种具有一些特殊条件约束的排序算法。但是在这些特殊条件的约束之下，桶排序算法的时间复杂度非常低，甚至可以说，

在我们目前接触的8种排序算法中，桶排序算法的时间复杂度是最低的。

桶排序算法的排序流程可以做如下总结：

步骤1：遍历整个待排序序列，并找到这个序列中的最大值

步骤2：使用待排序序列的最大值，作为一个辅助序列下标的最大值，创建一个辅助序列，我们称这个序列为桶序列（或者说直白一些就是桶数组）

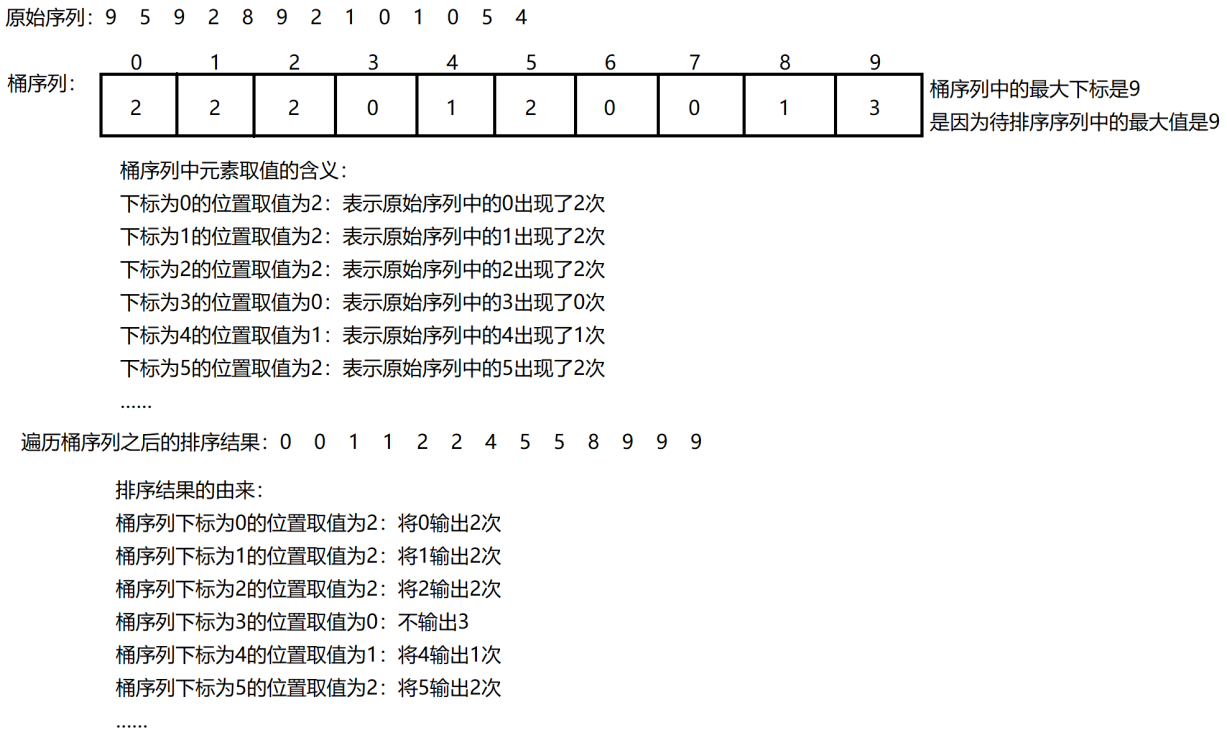
步骤3：再次遍历整个待排序序列，在遍历过程中，如果遇见取值为m的元素，那么就在桶序列下标为m的位置上+1

步骤4：遍历整个桶序列，在遍历过程中，如果下标为m的位置上的取值是k，那么就将这个下标m输出k次到原始数组中，排序完成

听了上面的步骤，可能各位同学会觉得一言难尽……总觉得有些云里雾里……

桶排序没有复杂的步骤，没有高深的结构，但是却能够通过简单的3次遍历完成对序列的排序，这到底是是怎么实现的呢？

下面我们使用一张图片来演示桶排序的具体步骤：



实际上通序列的排序思路非常简单：在通序列当中，桶序列的下标就是天然有序的，比如数组的下标，永远是从0开始，然后每隔一位，下标加1

我们就是利用了有序序列下标的这个特征，将待排序序列中的元素，作为桶排序的下标，然后使用桶排序中对应下标位上的取值，表示这个值在元素序列中出现了多少次

最后在遍历整个桶序列的时候，因为是按照下标遍历的，所以通序列中对应下标位的取值是几，我们就将这个下标输出多少次，然后进完成了排序

我们可以做如下想象：我们有n个桶，每一个桶都贴有一张数字标签，标签的取值范围是0到n-1，现在给你好多个乒乓球，每一个乒乓球上都有一个数字

数字的取值范围同样是0到n-1，并且取值有重复。但是这些小球是无序的，当你拿到一个乒乓球的时候，你要看一眼这个乒乓球上面的数字，并且将乒乓球放在对应数字标签的桶中。

当这些乒乓球被归类完成之后，我们从编号为0的桶开始，将桶中的乒乓球一个一个的取出，没有球的桶略过，那么在取出乒乓球的过程中，这些乒乓球就是有序的了

但是通过上面的描述和数组的一些特性进行对比，我们不难发现桶排序算法的一些限制：

1.在桶排序过程中，原始序列中绝对不能出现比0小的数字，因为有序序列的最小下标就是0，桶序列不能存储比0更小的值

2.原始序列中不能出现浮点值，因为有序序列的下标都是整数值，所以不能够使用桶序列保存浮点值

3.假设在待排序序列中存在的取值只有两种：0和999，并且都重复n次，那么我们就不得不创建一个长度为1000，下标取值范围为0-999的通序列作为辅助空间

但是在这个桶序列中，只有下标为0的位置和下标为999的位置上的取值是大于0的，其余位置上的取值全都是0，这样就造成了极大的空间浪费

所以我们认为桶排序非常适合原始序列中元素取值范围比较小，而且大量重复的情况

如果原始序列中的元素取值范围非常广泛而分散，重复率不高，在使用桶排序的时候，就会造成大量的空间浪费

通过上面的描述，我们可以写出如下的桶排序代码实现：

```
public void bucketSort(int[] array) {

    //[1]首先遍历整个数组，找到数组中元素的最大值
    int max = array[0];
    for(int i = 0; i < array.length; i++) {
        if(array[i] > max) {
            max = array[i];
        }
    }
```

```
10
11      //[2]根据数组的最大值，创建一个辅助数组，
12      //辅助数组的最大下标等于原始数组的最大值
13      //辅助数组的每一位可以看做一个桶，这个桶中最终的取值，
14      //就是这个桶对应下标的取值，在原始数组中出现的次数
15      int[] buckets = new int[max+1];
16
17      //[3]再次遍历整个原始数组，
18      //将原始数组中所有的元素按照取值，加入对应下标的桶中
19      for(int i = 0; i < array.length; i++) {
20          buckets[array[i]]++;
21      }
22
23      //[4]遍历桶数组，每一个桶中的取值是几，
24      //说明这个元素就在原始数组中出现了多少次
25      int index = 0;  //用来控制原始数组下标的变量
26      for(int i = 0; i < buckets.length; i++) {
27          for(int j = buckets[i]; j > 0; j--) {
28              array[index++] = i;
29          }
30      }
31
32  }
```

②时间复杂度、空间复杂度、稳定性分析

1.桶排序时间复杂度

实际上我们在实现桶排序的过程中，总共进行了3次序列遍历：

第1次：遍历长度为n的待排序序列，找到序列中的最大值m，并构建长度为m的桶序列

第2次：遍历长度为n的待排序序列，将待排序序列中的元素全部存放到桶序列的对应取值下标位中

第3次：遍历长度为m的桶序列，根据每一个桶当中的取值，输出对应次数的下标取值

不难看出，前两次遍历的长度都是n，所以加和是2n，第3次遍历的长度是m，那么3次遍历的结果相加，就得到2n+m

在省略系数之后，我们得到：桶排序的时间复杂度是O(n+m)，其中n表示待排序序列的长度，m表示待排序序列中最大值的取值

2.桶排序空间复杂度

桶排序中，我们需要创建一个长度为m的桶序列作为辅助空间

所以：桶排序的空间复杂度是O(m)

3.桶排序的稳定性

在桶排序过程中，我们主要考虑的是待排序元素出现的次数，所以我们并不能很好的讨论桶排序算法的稳定性

但是实际上桶排序还有一种实现方式，是稳定排序

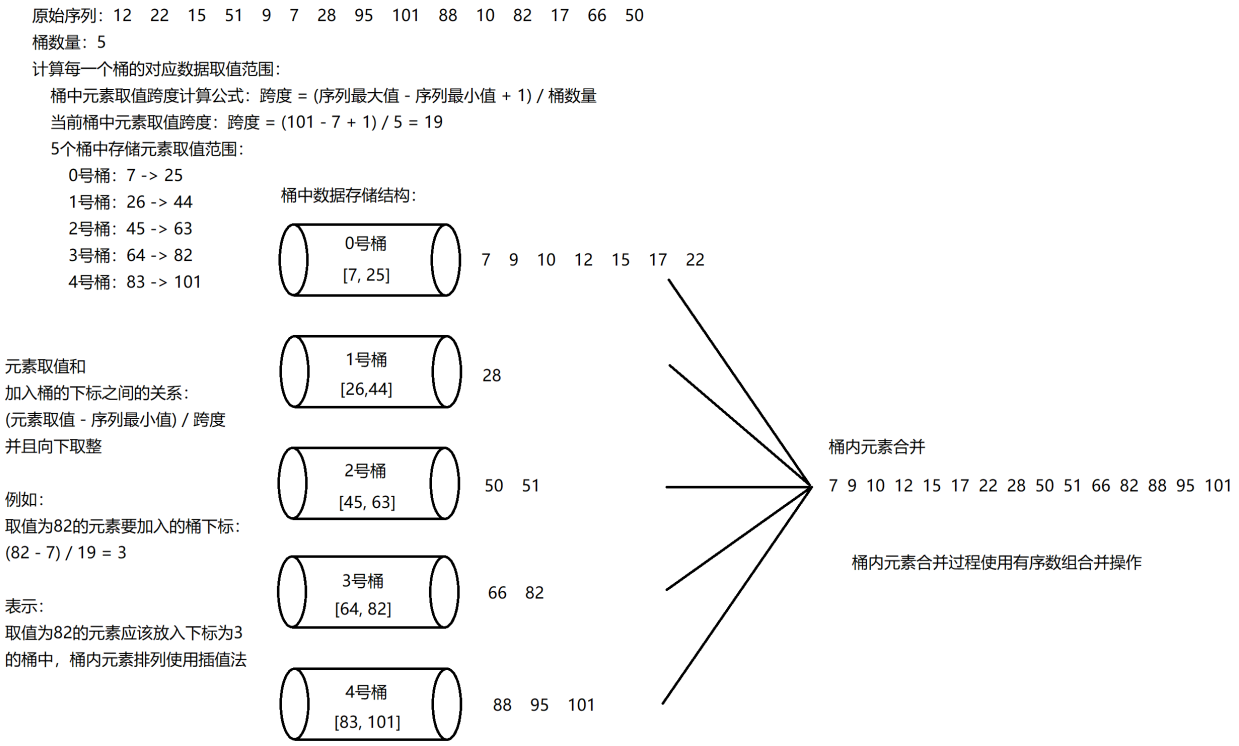
另一种实现方式我们稍后通过一个具体案例进行说明

③桶排序的另一种实现：分桶操作

上面所述的原生的桶排序算法，是一种使用“空间换时间”的操作，也就是使用大量的空间消耗，甚至不惜浪费存储空间，达到提升排序速度的目的

但是如果我们将将空间和时间进行一些均衡，就能够得到桶排序的另一种实现方式

下面我们通过一个具体的案例，来说明这种排序方式



通过上述操作我们不难看出：这种操作方式使用的桶的数量不在取决于序列内最大值的取值

而是人为规定，并且桶内存储元素的方式使用链表挂载，这大大节省了存储空间

但是同时也多出来一些诸如元素所在桶下标计算、有序数组合并等额外的操作，这会引起一定程度上的排序效率下降

但是影响并不明显。同样桶内元素因为使用插值插入的方式实现，所以桶排序的这种操作方式，也是一种稳定的排序方式。