版本号：v1.8 阶段：初稿

# 排序（核心版）

MRL Liu

2022年08月01日

排序算法是最基本的算法之一。排序即对一系列对象根据某个关键字进行排序。首先介绍一下排序术语：

|  |
| --- |
| **稳定**：a=b，排序前a在b之前，排序后a仍然在b之前  **不稳定**：a=b，排序前a在b之前，排序后a可能在b之后  **内排序**：所有排序操作都可以在内存中完成  **外排序**：当数据太大时需要将数据放在磁盘中  **时间复杂度**：一个算法执行所耗费时间的估计值  **空间复杂度**：一个算法执行所耗费空间的估计值 |

根据排序方式，排序算法整体上可以划分为内部排序和外部排序。内部排序是指数据记录在内存中进行排序，不需要申请额外的存储空间。外部排序是指在排序过程中需要申请额外的内存空间。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **排序方式** | **排序算法** | **最好情况** | **最坏情况** | **平均时间复杂度** | **空间复杂度** | **稳定性** |
| In-place | 选择排序 |  |  |  |  | 不稳定 |
| 冒泡排序 |  |  |  |  | 稳定 |
| 插入排序 |  |  |  |  | 稳定 |
| 快速排序 |  |  |  |  | 不稳定 |
| 希尔排序 |  |  |  |  | 不稳定 |
| 堆排序 |  |  |  |  | 不稳定 |
| Out-place | 归并排序 |  |  |  |  | 稳定 |
| 计数排序 |  |  |  |  | 稳定 |
| 桶排序 |  |  |  |  | 稳定 |
| 基数排序 |  |  |  |  | 稳定 |

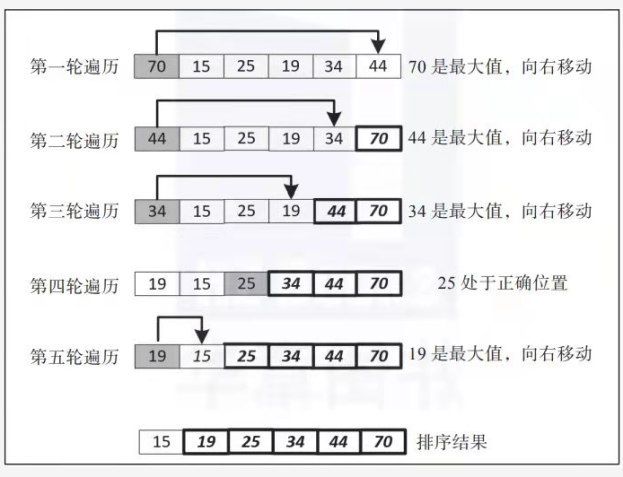
（注意：n指的是数据规模；k指的是“桶”的个数；In-place指的是占用常数内存，不占用额外内存；Out-place指的是占用额外内存）

### 1、选择排序

选择排序（selection-sort）是性能鲁棒性最好的排序算法之一，任何情况下耗费时间都为，而且不占用额外内存，适合于数据规模n较小的情景。

**【算法思路】**：从未排序序列中找到最小（大）元素，存放到排序序列的末尾位置，直至未排序序列为空。

**【编码思路】：**双层循环，外层循环负责从左至右确定已排序序列，内层循环负责从左至右在未排序序列中查找最值，找到后交换两数即可。

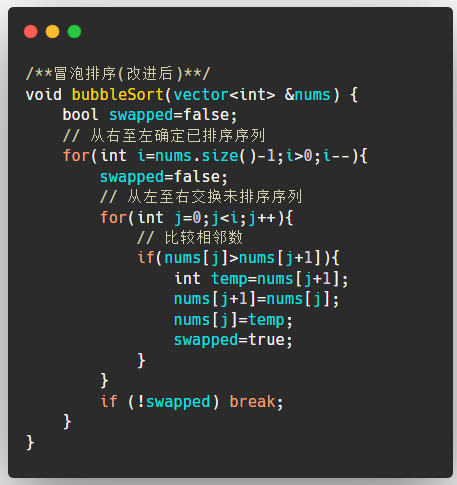
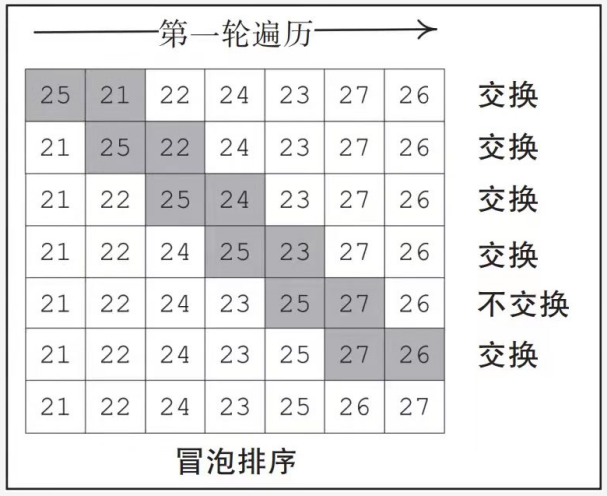


### 2、冒泡排序

冒泡排序（bubble-sort）比选择排序性能稍好一定，而且稳定，排序不影响相同数的位置，平均时间复杂度为，同样不占用额外内存，适合于数据规模n较小的情景。

**【算法思路】**：连续进行n轮遍历，每一轮遍历都让相邻数两两交换，确定一个已排序的数，n轮遍历后就会确定所有的数。

**【编码思路】：**双层循环，外层循环负责从右至左确定已排序序列，内存循环负责从左至右交换未排序序列中的相邻数。此处有个改进小技巧，内层循环维护一个bool值，如果当前一轮遍历没有交换过说明已完成排序，直接退出外层循环。

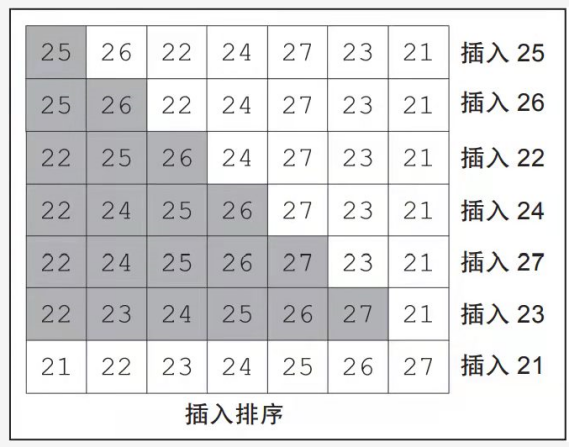


### 3、插入排序

插入排序（insert-sort）的性能和冒泡排序差不多，平均时间复杂度为，同样不占用额外内存，适合于数据规模n较小的情景。

**【算法思路】**：序列左侧视为已排序序列，序列右侧视为未排序序列，每次从未排序序列中移除一个元素到已排序序列，已排序序列每次加入新的元素就要重新交换一遍来确定顺序。

**【编码思路】：**双层循环，外层循环负责从左至右扩展已排序序列，内层循环负责从右至左在已排序序列中交换完成排序。



### 4、快速排序

快速排序（quick-sort）是性能鲁棒性最好的排序算法之一，任何情况下耗费时间都为，而且不占用额外内存，适合于数据规模n较小的情景。

**【算法思路】**：

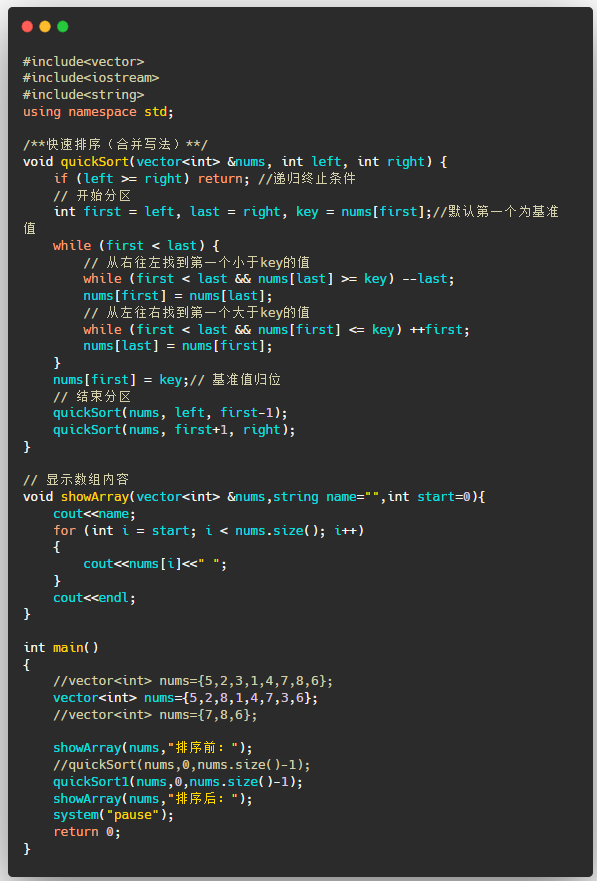
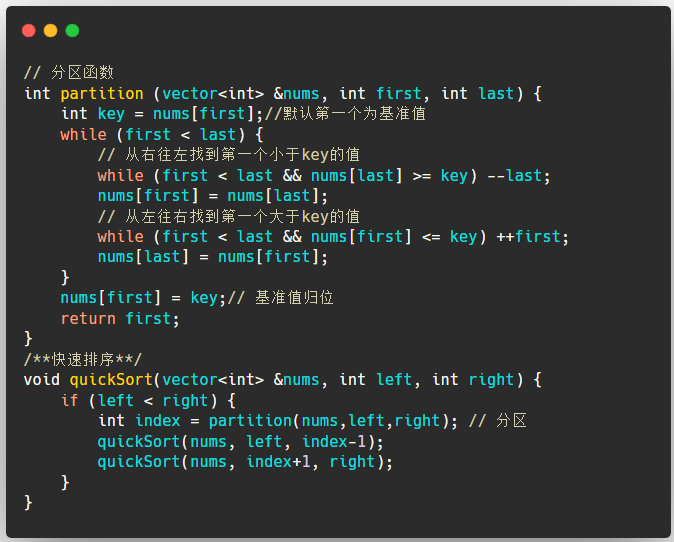
(1)首先设定一个分界值，通过该分界值将数组分成左右两部分。

(2)将大于或等于分界值的数据集中到数组右边，小于分界值的数据集中到数组的左边。此时，左边部分中各元素都小于或等于分界值，而右边部分中各元素都大于或等于分界值。

(3)然后，左边和右边的数据可以独立排序。对于左侧的数组数据，又可以取一个分界值，将该部分数据分成左右两部分，同样在左边放置较小值，右边放置较大值。右侧的数组数据也可以做类似处理。

(4)重复上述过程，可以看出，这是一个递归定义。通过递归将左侧部分排好序后，再递归排好右侧部分的顺序。当左、右两个部分各数据排序完成后，整个数组的排序也就完成了。

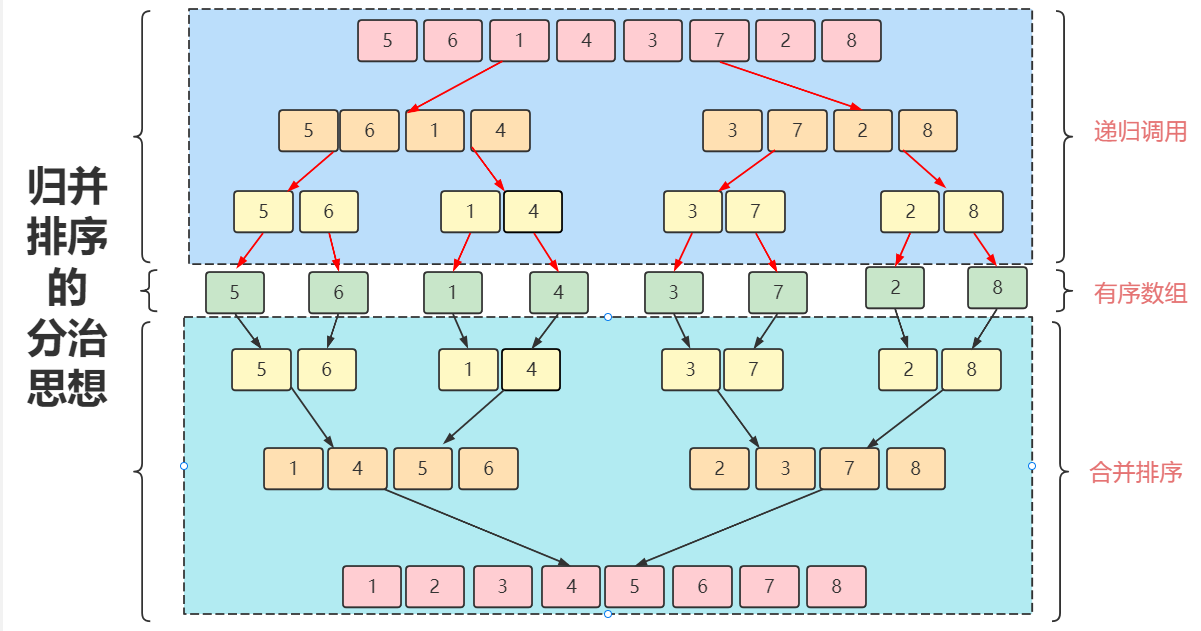
**【编码思路】：**双层循环，外层循环负责划分已排序序列和未排序序列，内存循环负责从未排序序列中查找最值，找到后交换两数即可。



### 5、归并排序

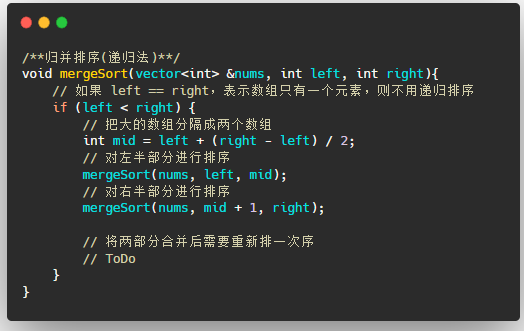
归并排序（merge-sort）采用了分治思想（即将一个大问题分解成若干个小问题来解决），分治思想在编码上一般容易使用递归法。

【算法思路】：归并排序认为如果要给一个数组排序，可以把整个大数组对等分成2个小数组，2个小数组继续分割，直到分出的每个小数组大小都是1，只有1个元素的数组一定是有序数组了，然后再把2个有序数组合并成新的更大数组，合并的时候需要重新排下序，合并后的数组就又有序了。

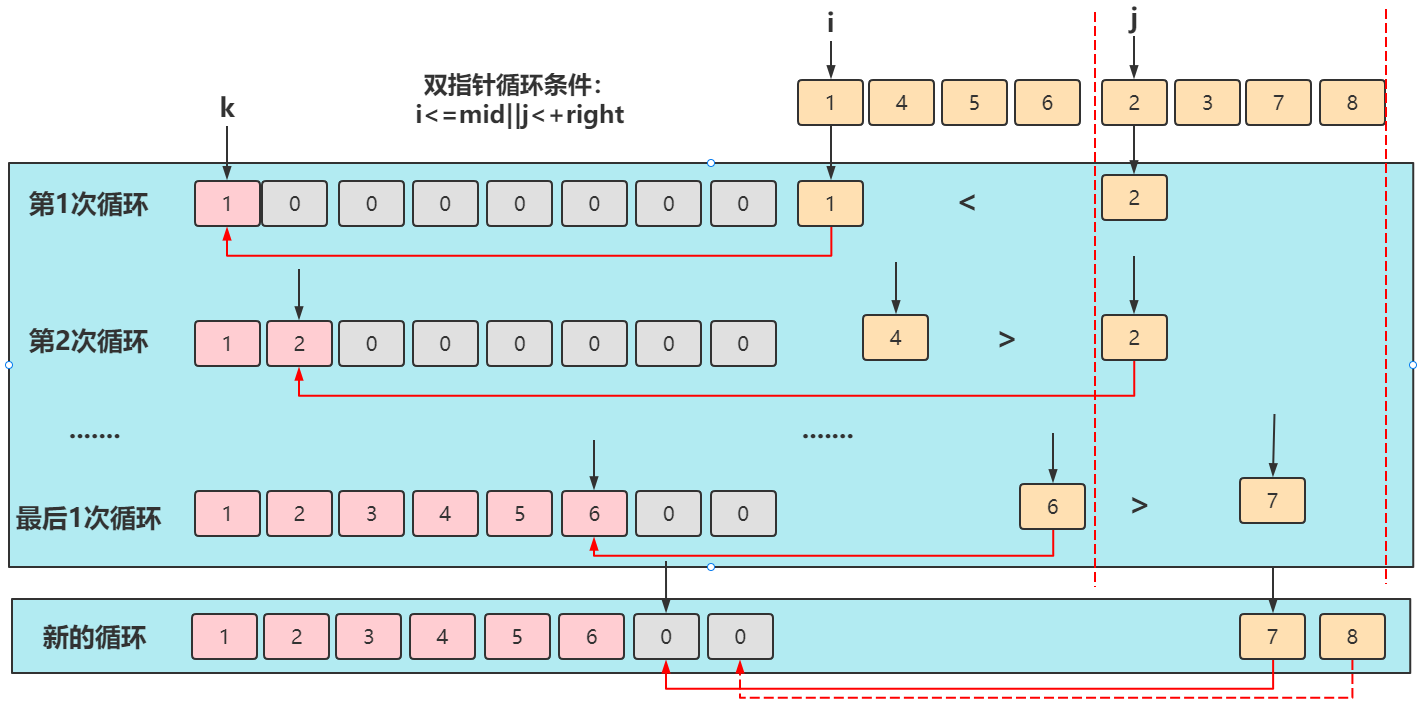


给2个小的有序数组（长度=N/2）重新排序通常比直接排序一个大的无序数组（长度=N）要快，这也是归并排序要先划分后合并的原因，先划分直至每个数组都是有序数组，然后合并排序成一个新的有序数组。

【编码思路】：在上述的思路分析中，首先递归调用来不断分割数组的编码很简单了，如下：



接下来就是考虑如何合并2个有序数组了，归并排序使用了申请新数组的双指针法来合并排序。首先直接申请一个足够容纳2个有序数组的临时数组，然后i指针指向数组的左半部分nums[left,mid]，j指针指向数组的右半部分nums[mid+1,right]。接下来依次比较2个数组中的每个元素，哪个小数组的数更小就放入临时数组中，如下：

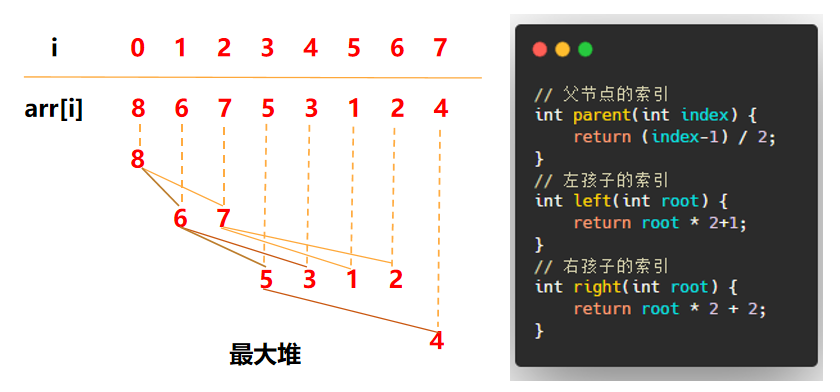


最后双指针循环结束后，原数组中还可能会有剩余的没有复制，因为双指针的循环条件是有一个指针达到其终点即可，所以循环结束时至少有一个数组的元素全部被复制完（图中是i指针指向的左数组），另一个数组则需要一个新的循环来复制，最后直接上代码。

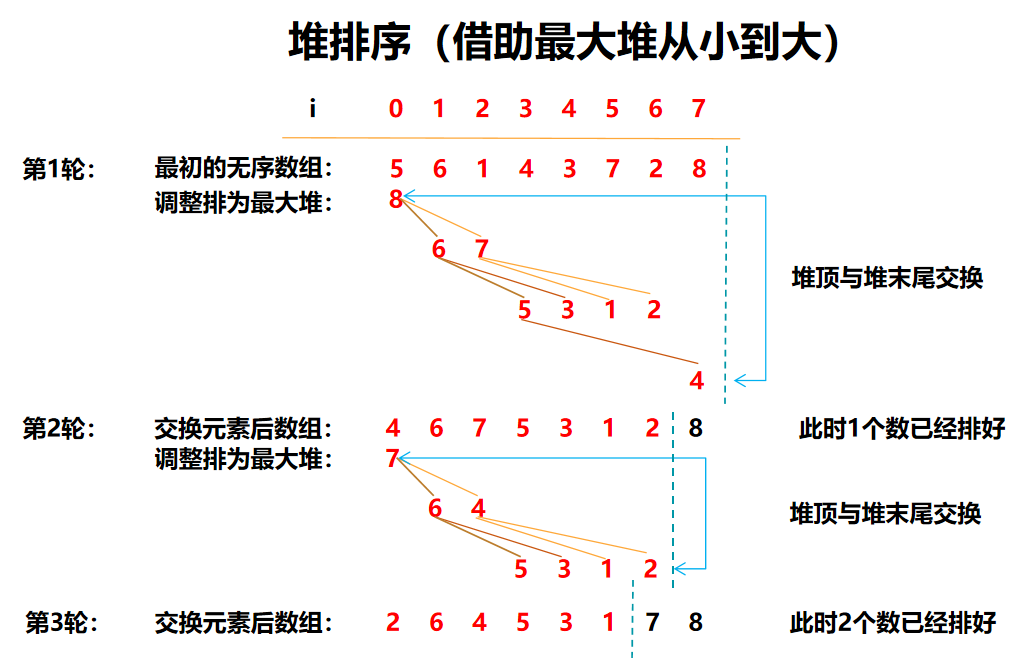


### 6、堆排序

堆排序（heap-sort）是利用一种叫做堆（heap）的数据结构所设计的排序算法。堆本质上就是用数组实现的二叉树，所以也叫做二叉堆。堆有一个性质，叫做堆有序。根据这种堆有序的性质，堆分为2种，最大堆（大顶堆）和最小堆（小顶堆）。在最大堆中，父节点的值都比每一个子节点的值要大；在最小堆中，父节点的值都比每一个子节点的值要小。（注意和二叉搜索树的区别，二叉搜索树是左子树的值比父节点的值小，右子树的值比父节点的值大）。下图是一个最大堆的内部数据的逻辑结构，根据这种特点，给定任意一个数所在的索引index，我们就可以知道它的父节点索引或者左右子节点的索引。

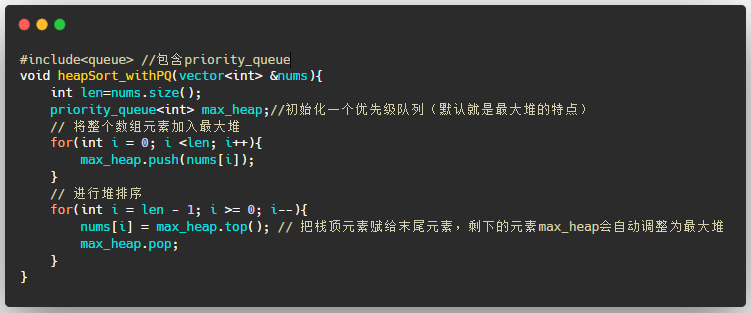


**【算法思路】**：假设从小到大排序，堆排序的基本思路是先将整个无序数组构造成一个最大堆，那么最大堆的根节点一定就是全局最大的数，将其取出来放入已排序序列；删除掉根节点的最大堆在重新构建一个最大堆，从而又找到了一个局部最大值，接着将其放入已排序序列，当无序序列中构建最大堆的数只剩下一个时，排序也就完成。过程如下：



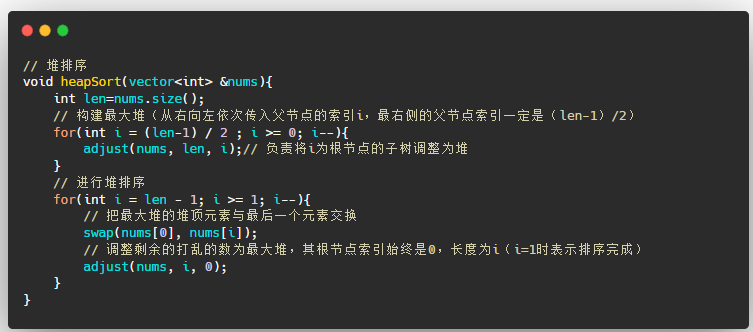
**【编码思路】：**在上述思路分析中，可知将一个无序序列转换为堆是最关键的步骤。最大堆是一个父节点都大于子节点的二叉树，在编码上可以采取递归的方式来实现。我们首先借助C++的优先级队列来实现排序。

在大多数编程语言中，优先级队列（Priority Queue）就可以看做是一个堆结构（其底层可能就是借助堆实现的），优先级队列的特点就是其队首元素始终是整个队列中最大或者最小的数，这和最大堆或最小堆的特点一样。所以，我们给出如下代码即可完成排序：

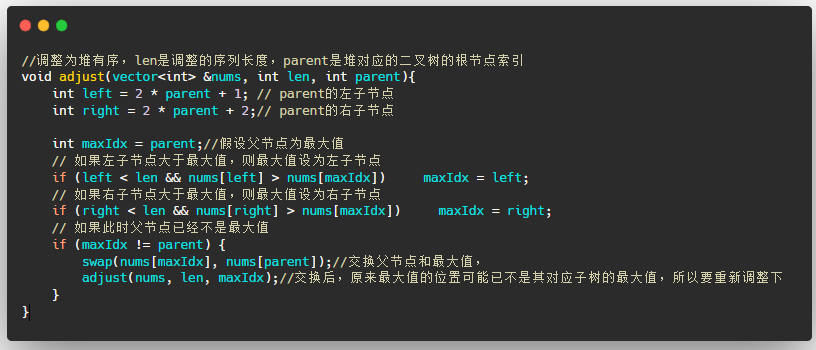


上述编码其实就是堆排序的思路，借助一个最大堆或最小堆，把无序序列始终调整为堆有序，其堆顶元素一定是全局最大或最小。现在假设面试官不让我们继续使用Priority Queue，让我们手写堆的调整结构，不要怕，难度也不会增加很多。

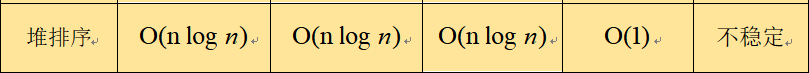
假设我们有一个函数adjust()，其功能就是负责将i为根节点的子树调整为堆有序，即满足最大堆或最小堆，那么我们借助这个adjust就可以写出如下堆排序的代码：



现在我们的任务就是写出adjust函数，其也利用了二叉树递归的思路，只要保证每个小子树中父节点比孩子节点都大或者都小即可，如下：



**【**性能分析**】**：



此时我们已经完成了整个堆排序的任务，最重要的是adjust的实现，读者要多默写几遍。学习堆排序后，我们实际也同时掌握了堆、优先级队列的核心知识，一石三鸟。