# 第12章 排序

## 12.1 排序的基础知识

排序是非常基础、重要的算法，它将若干数据依照特定的顺序进行排列。如果排序的对象是数值，那么按数值递增或递减的顺序进行排序；如果排序的对象是字符串，那么按照字典顺序进行排序。由于数据排序之后能够利用二分查找算法提高查找的效率，因此很多数据都是排序之后再存储的。

目前已经有多种不同的排序算法。在面试的时候面试官经常要求应聘者比较插入排序、冒泡排序、堆排序、计数排序、归并排序和快速排序等不同算法的优劣。因此，应聘者在准备面试的时候一定要对各种排序算法的特点非常熟悉，能够从额外空间消耗、平均时间复杂度和最差时间复杂度等方面比较它们的优点与缺点。

一般面试中最可能遇到的是计数排序、快速排序和归并排序，并且经常有面试官要求现场写出这3种排序的代码，因此，应聘者在准备面试的时候一定要做到对这3种排序能够信手拈来才可以。另外，还需要着重理解快速排序和归并排序这两种算法的思想，因为有很多典型的算法面试题都可以应用它们的思想来解决。

如果面试题中的输入数据不是排序的，但数据排序之后便于解决问题，那么如果时间复杂度允许就可以先将输入的数据排序。下面是一个经典的例子。

### 面试题74：合并区间

题目：输入一个区间的集合，请将重叠的区间合并。每个区间用两个数字比较，分别表示区间的起始位置和结束位置。例如，输入区间[[1，3]，[4，5]，[8，10]，[2，6]，[9，12]，[15，18]]，合并重叠的区间之后得到[[1，6]，[8，12]，[15，18]]。

分析：首先需要考虑两个区间在什么情况下才能被合并。如图12.1（a）所示，如果区间1的起始位置小于区间2的起始位置，并且区间1的结束位置大于或等于区间2的起始位置，那么两个区间中间有重叠部分，它们能够被合并，合并之后的区间的起始位置是区间1的起始位置，合并之后的区间的结束位置是两个区间的结束位置的较大者。反之，如图12.1（b）所示，如果区间3的起始位置小于区间4的起始位置，并且区间3的结束位置也小于区间4的起始位置，那么两个区间没有重叠部分，它们不能被合并。

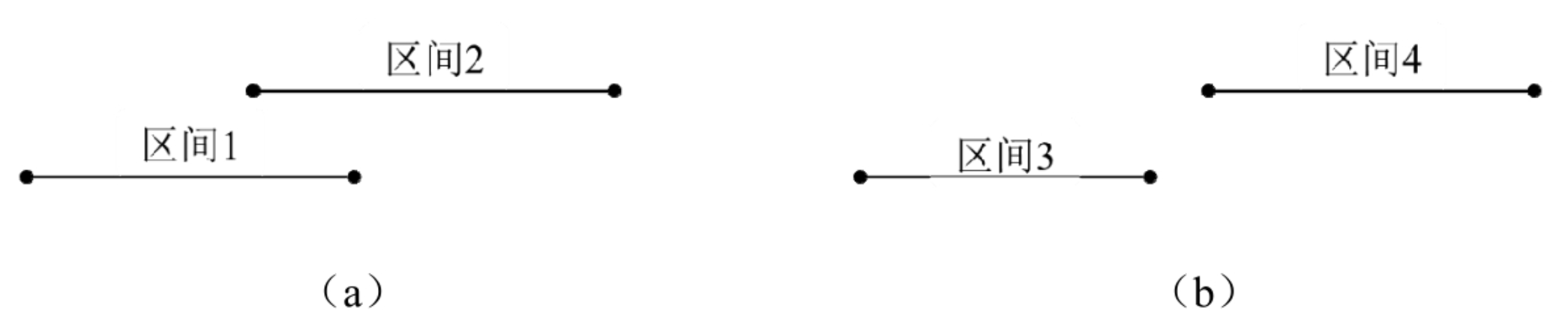
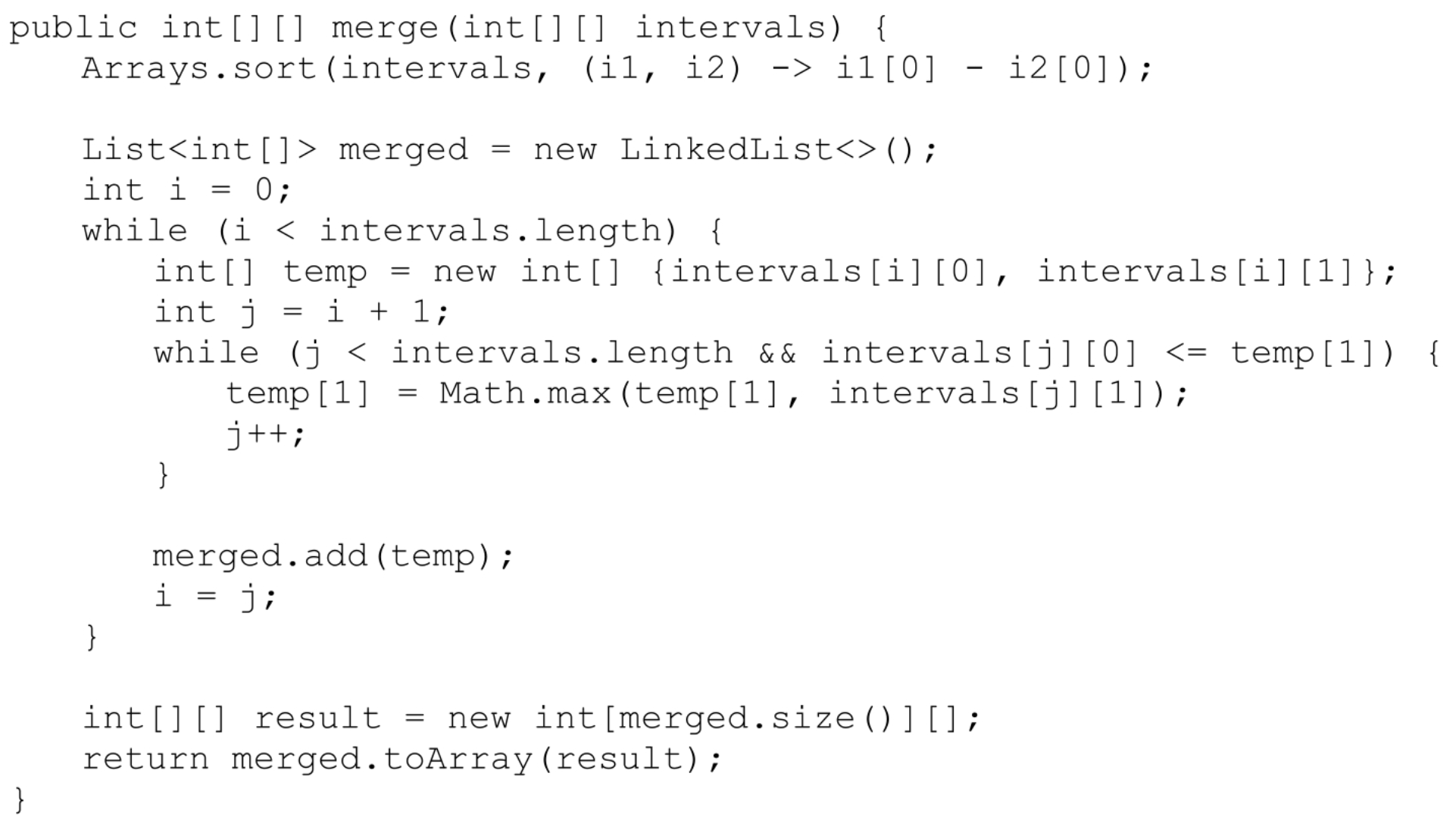


图12.1 判断两个区间能否被合并

说明：（a）区间1和区间2能够被合并；（b）区间3和区间4不能被合并如果先将所有区间按照起始位置排序，那么只需要比较相邻两个区间的结束位置就能知道它们是否重叠。如果它们重叠就将它们合并，然后判断合并的区间是否和下一个区间重叠。重复这个过程，直到所有重叠的区间都合并为止。

例如，如果将区间列表[[1，3]，[4，5]，[8，10]，[2，6]，[9，12]，[15，18]]按照起始位置排序就可以得到[[1，3]，[2，6]，[4，5]，[8，10]，[9，12]，[15，18]]。接下来扫描排序之后的区间。区间[1，3]和[2，6]重叠，可以合并，它们合并之后得到区间[1，6]。接下来的区间[1，6]和[4，5]重叠，也可以合并，它们合并之后得到区间[1，6]。由于区间[1，6]与下一个区间[8，10]不重叠，因此将区间[1，6]保存到合并之后的区间列表中，然后从区间[8，10]开始与后面的区间合并。区间[8，10]与它的下一个区间[9，12]重叠，合并之后得到区间[8，12]。由于区间[8，12]与下一个区间[15，18]不重叠，因此将区间[8，12]添加到合并之后的区间列表中，再从下一个区间[15，18]开始合并之后的区间。最终合并之后的区间列表是[[1，6]，[8，12]，[15，18]]。

这个合并的过程可以用如下所示的参考代码实现：



上述代码先调用函数Arrays.sort对输入的区间数组进行排序。排序的标准由lambda表达式“（i1，i2）->i1[0]-i2[0]”确定，该lambda表达式表明将两个区间i1和i2的起始位置（i1[0]和i2[0]）进行比较，并将起始位置小的排在前面。

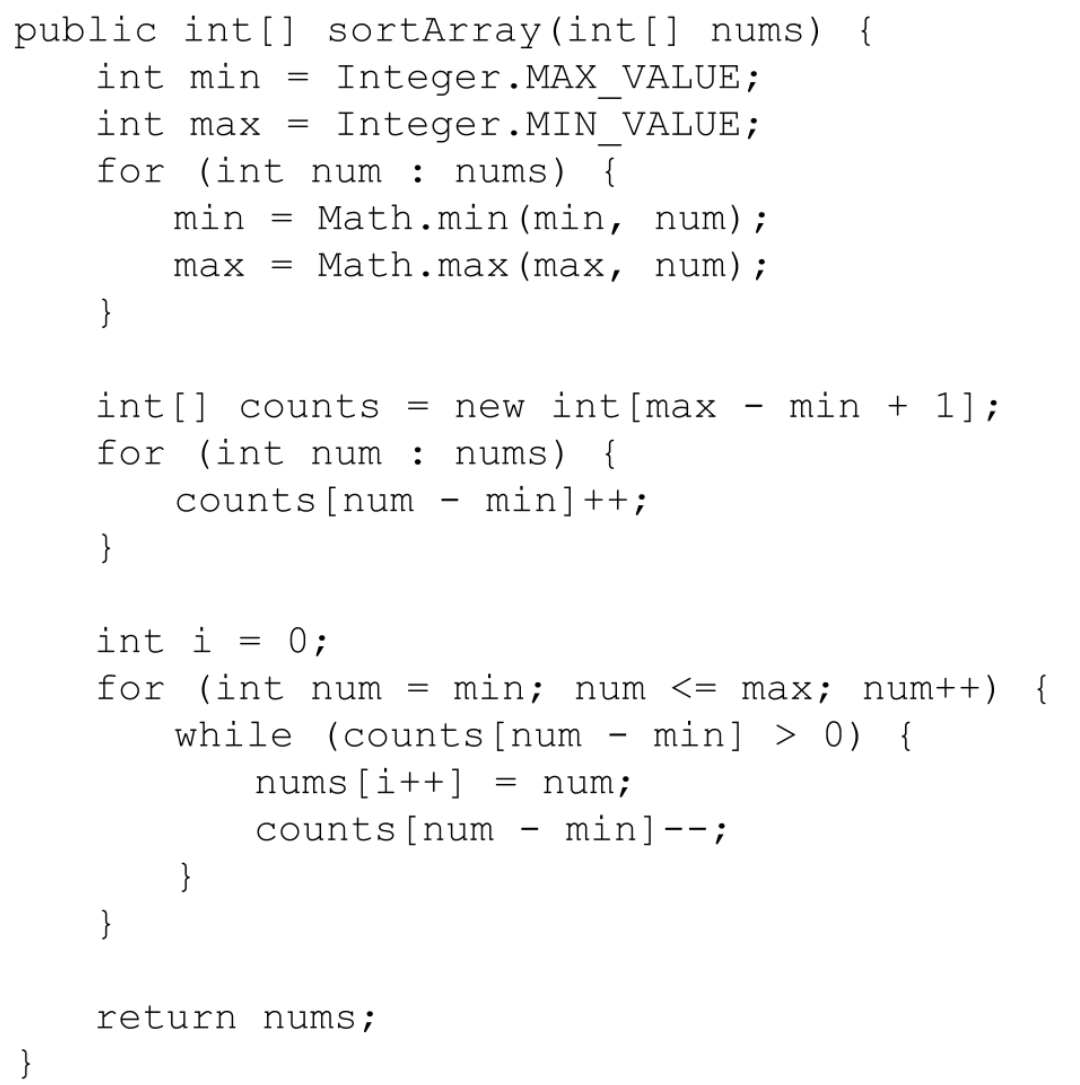
如果输入数组中有n个区间，那么将它排序的时间复杂度是O（nlogn），接着逐一扫描排序的区间数组并将相邻的区间合并。虽然代码中有嵌套的二重循环，但每个区间只会扫描一次，因此时间复杂度是O（n）。上述算法的总体时间复杂度是O（nlogn）。

## 12.2 计数排序

计数排序是一种线性时间的整数排序算法。如果数组的长度为n，整数范围（数组中最大整数与最小整数的差值）为k，对于k远小于n的场景（如对某公司所有员工的年龄排序），那么计数排序的时间复杂度优于其他基于比较的排序算法（如归并排序、快速排序等）。

计数排序的基本思想是先统计数组中每个整数在数组中出现的次数，然后按照从小到大的顺序将每个整数按照它出现的次数填到数组中。例如，如果输入整数数组[2，3，4，2，3，2，1]，扫描一次整个数组就能知道数组中1出现了1次，2出现了3次，3出现了2次，4出现了1次，于是先后在数组中填入1个1、3个2、2个3及1个4，就可以得到排序后的数组[1，2，2，2，3，3，4]。

计数排序的参考代码如下所示：



上述代码先扫描数组nums得到整数的最大值和最小值，并根据整数范围创建辅助数组counts。数组counts用来统计每个整数出现的次数。

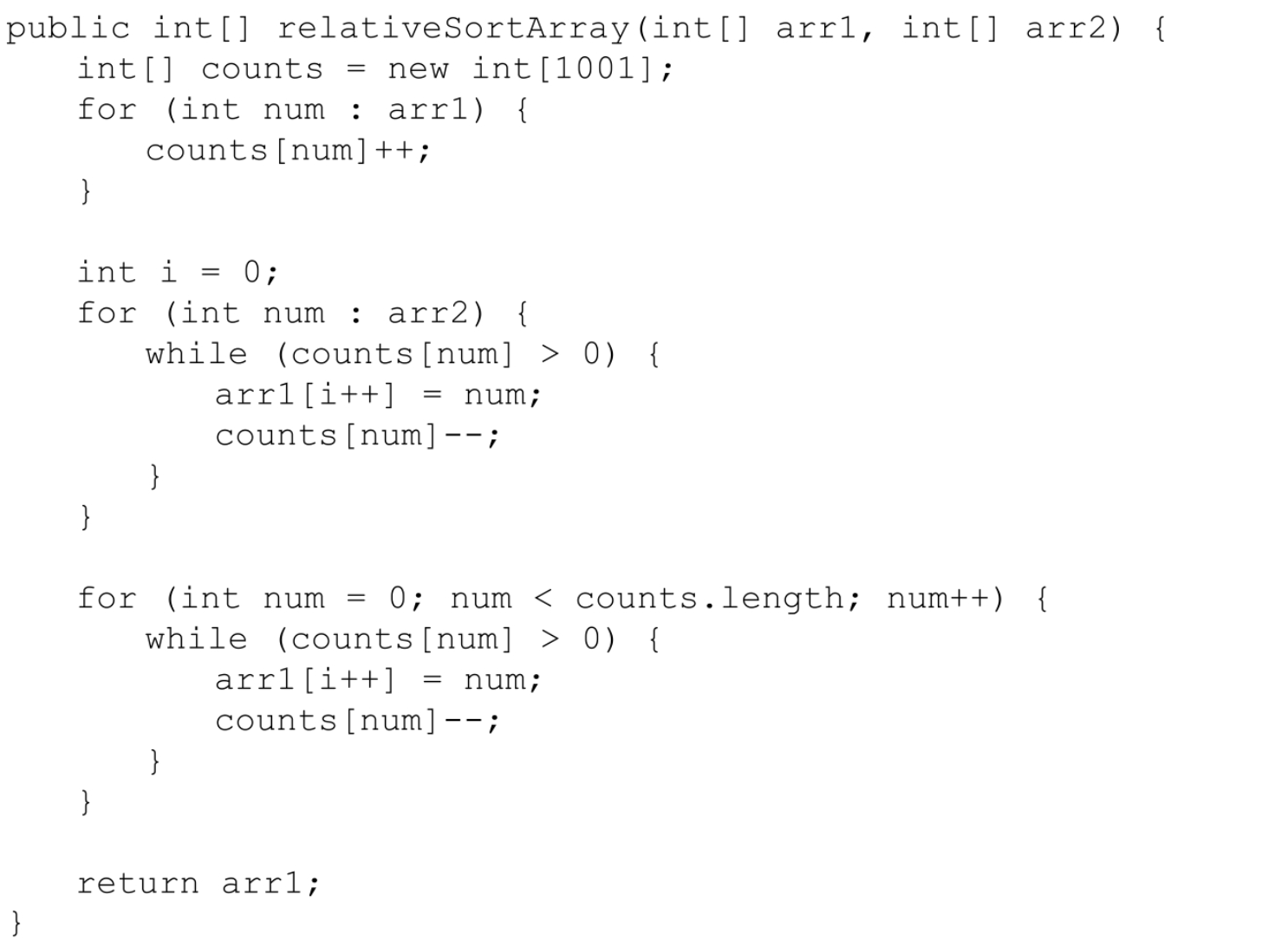
如果数组的长度为n，整数的范围为k，那么计数排序的时间复杂度就是O（n+k）。由于需要创建一个长度为O（k）的辅助数组counts，因此空间复杂度为O（k）。当k较小时，无论从时间复杂度还是空间复杂度来看计数排序都是非常高效的算法。当k很大时，计数排序可能就不如其他排序算法（如快速排序、归并排序）高效。

### 面试题75：数组相对排序

题目：输入两个数组arr1和arr2，其中数组arr2中的每个数字都唯一，并且都是数组arr1中的数字。请将数组arr1中的数字按照数组arr2中的数字的相对顺序排序。如果数组arr1中的数字在数组arr2中没有出现，那么将这些数字按递增的顺序排在后面。假设数组中的所有数字都在0到1000的范围内。例如，输入的数组arr1和arr2分别是[2，3，3，7，3，9，2，1，7，2]和[3，2，1]，则数组arr1排序之后为[3，3，3，2，2，2，1，7，7，9]。

分析：题目明确提出数组中的数字都在0到1000的范围内。这是一个很明显的提示，据此可以考虑采用计数排序。先统计数组[2，3，3，7，3，9，2，1，7，2]中每个数字出现的次数，发现数字1出现了1次，2出现了3次，3出现了3次，7出现了2次，以及9出现了1次。接下来根据数组[3，2，1]确定的数字顺序，先后输出3个3、3个2、1个1。由于还剩下数字7和9，因此再按照大小输出2个7和1个9。

这个排序过程可以用如下所示的参考代码实现：

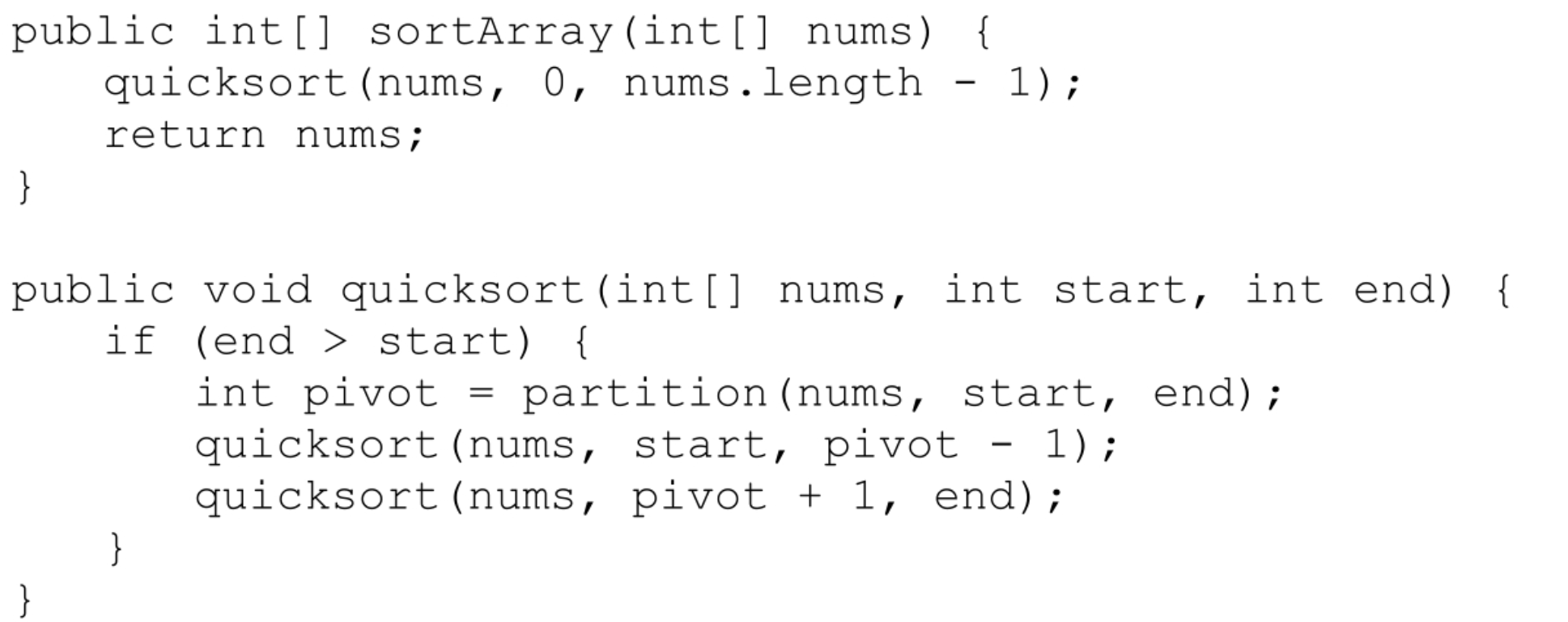


由于这个题目中的数字在0到1000的范围内，上述代码用来统计每个数字出现次数的辅助数组counts的长度为1001，是一个常数，因此空间复杂度可以认为是O（1）。如果数组arr1的长度为m，数组arr2的长度为n，那么时间复杂度是O（m+n）。

## 12.3 快速排序

快速排序是一种非常高效的算法，从其名字可以看出这种排序算法最大的特点就是快。当表现良好时，快速排序的速度比其他主要对手（如归并排序）快2～3倍。

快速排序的基本思想是分治法，排序过程如下：在输入数组中随机选取一个元素作为中间值（pivot），然后对数组进行分区（partition），使所有比中间值小的数据移到数组的左边，所有比中间值大的数据移到数组的右边。接下来对中间值左右两侧的子数组用相同的步骤排序，直到子数组中只有一个数字为止。这个过程可以用如下所示的递归代码实现：



理解快速排序的关键在于理解它分区的过程。下面以数组[4，1，5，3，6，2，7，8]为例分析分区的过程。假设数字3被随机选中称为中间值，该数字被交换到数组的尾部。接下来初始化两个指针，指针P1初始化至下标为-1的位置，指针P2初始化至下标为0的位置，如图12.2（a）所示。始终将指针P1指向已经发现的最后一个小于3的数字。此时尚未发现任何一个小于3的数字，因此将指针P1指向一个无效的位置。将指针P2从下标为0的位置开始向右扫描数组中的每个数字。当指针P2指向第1个小于3的数字1时，指针P1向右移动一格，然后交换两个指针指向的数字，此时数组（即两个指针）的状态如图12.2（b）所示。继续右移指针P2直到遇到下一个小于3的数字2，指针P1再次向右移动一格，然后交换两个指针指向的数字，此时数组（即两个指针）的状态如图12.2（c）所示。继续右移指针P2直到指向数字3也没有遇到新的小于3的数字，此时整个数组都已经扫描完毕。再次将指针P1向右移动一格，然后交换指针P1和P2指向的数字，于是所有小于3的数字都位于3的左边，所有大于3的数字都位于数组的右边，如图12.2（d）所示。

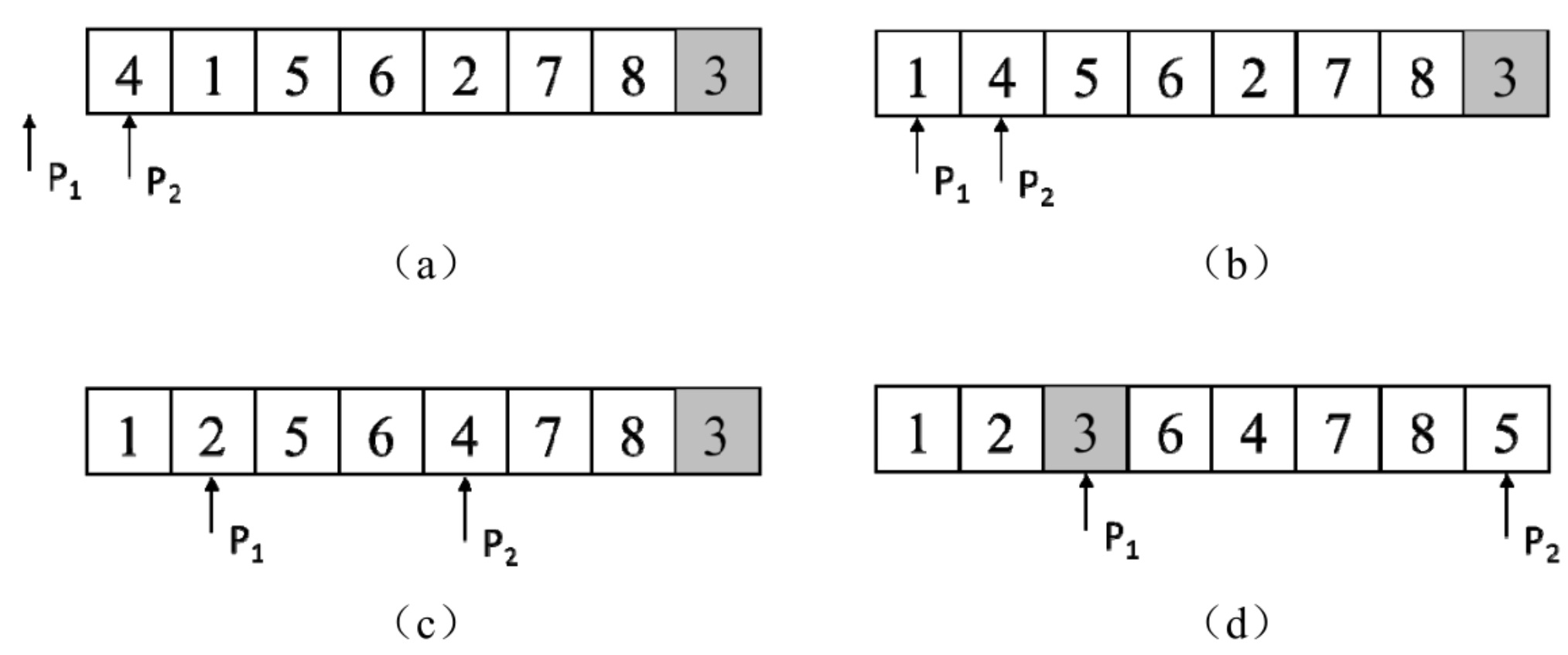
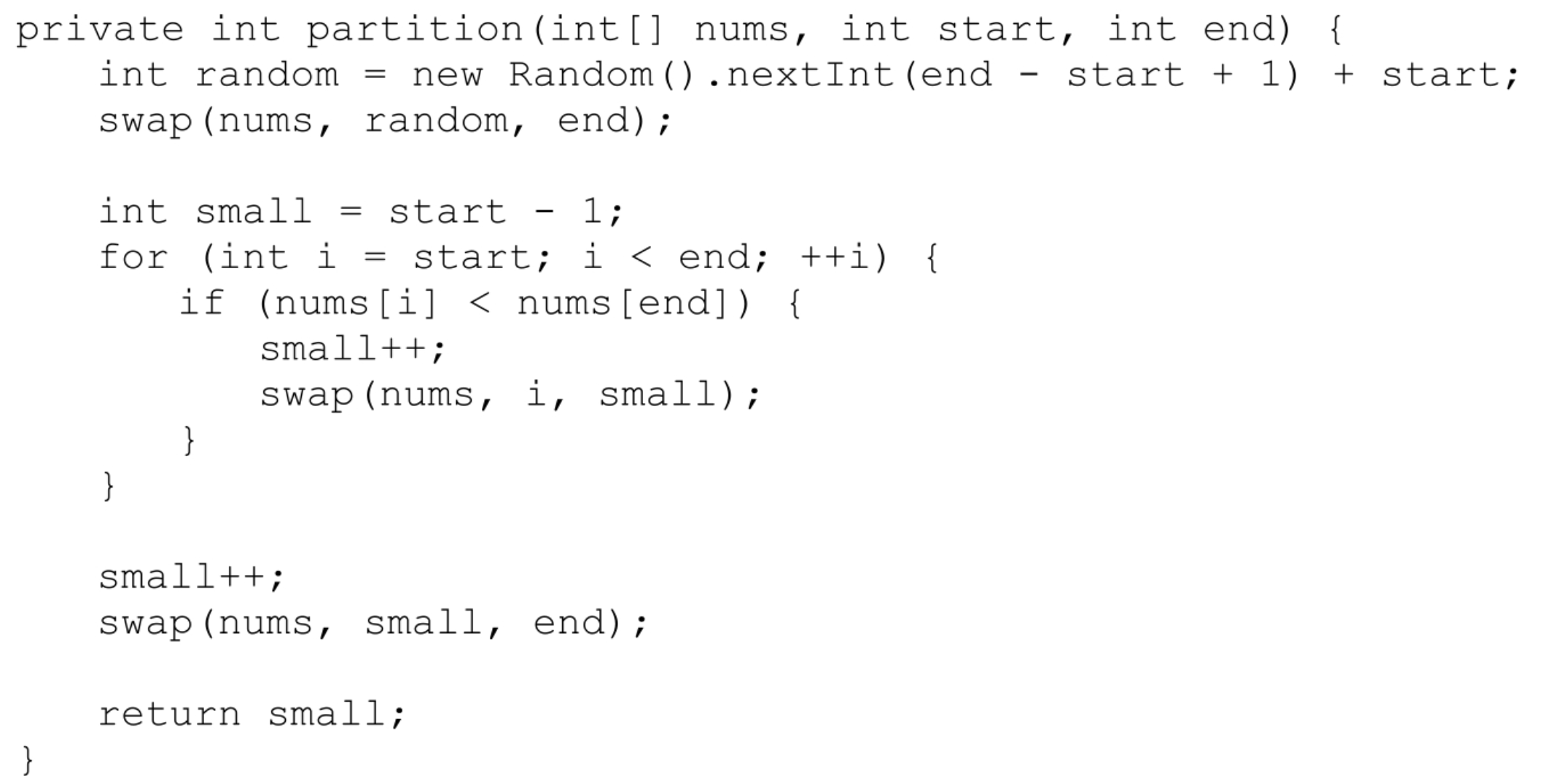


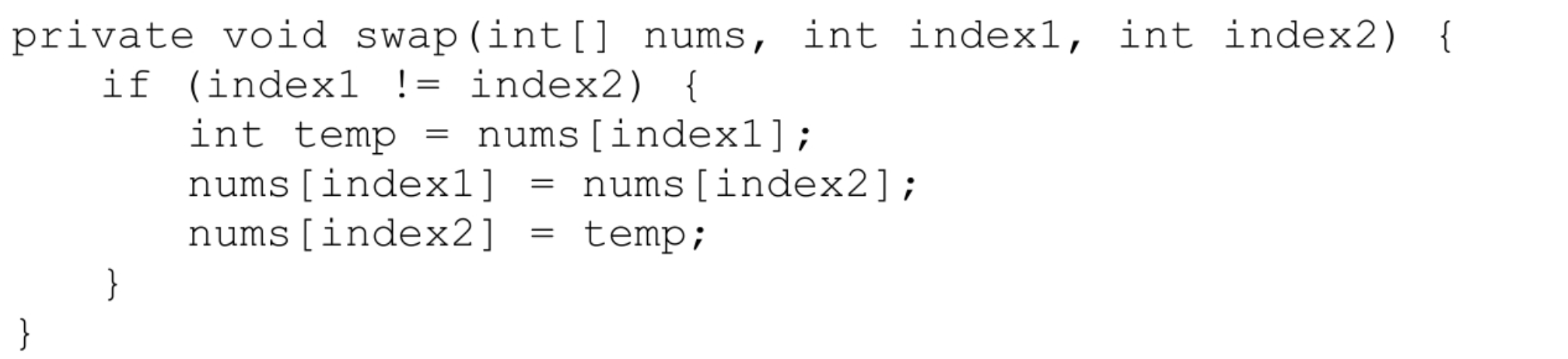
图12.2 对数组[4，1，5，3，6，2，7，8]分组的过程

说明：（a）选取3作为中间值，将其交换至数组的尾部。初始化指针P1至下标为-1的位置，指针P2至下标为0的位置。（b）右移指针P2直到遇到第1个比3小的数字1，指针P1右移一格然后交换指针P1和P2指向的数字。（c）右移指针P2直到遇到下一个比3小的数字2，指针P1右移一格然后交换指针P1和P2指向的数字。（d）右移指针P2直到指向数字3，指针P1右移一格然后交换指针P1和P2指向的数字上述分区过程可以用如下所示的代码实现：



函数partition中的变量small相当于指针P1，它始终指向已经发现的最后一个小于中间值的数字。而for循环中的变量i相当于指针P2，它从左到右扫描整个数组。函数partition先将随机选取的中间值交换到数组的尾部，最后又将它交换到合适的位置，使比它小的数字都在它的左边，比它大的数字都在它的右边。函数partition的返回值是中间值的最终下标。

函数swap用来交换数组中两个下标的值，它的代码如下所示：



快速排序的时间复杂度取决于所选取的中间值在数组中的位置。如果每次选取的中间值在排序数组中都接近数组中间的位置，那么快速排序的时间复杂度是O（nlogn）。如果每次选取的中间值都位于排序数组的头部或尾部，那么快速排序的时间复杂度是O（n2）。这也是随机选取中间值的原因，避免在某些情况下快速排序退化成时间复杂度为O（n2）的算法。由此可知，在随机选取中间值的前提下，快速排序的平均时间复杂度是O（nlogn），是非常高效的排序算法。

很多面试官喜欢要求应聘者手写快速排序算法的代码，因此应聘者需要深刻理解快速排序的思想及分区的过程，这样在遇到要求手写快速排序的代码时也能心中有底。另外，快速排序中的partition函数还经常被用来选择数组中第k大的数字，而这也是一道非常经典的算法面试题。

### 面试题76：数组中第k大的数字

题目：请从一个乱序数组中找出第k大的数字。例如，数组[3，1，2，4，5，5，6]中第3大的数字是5。

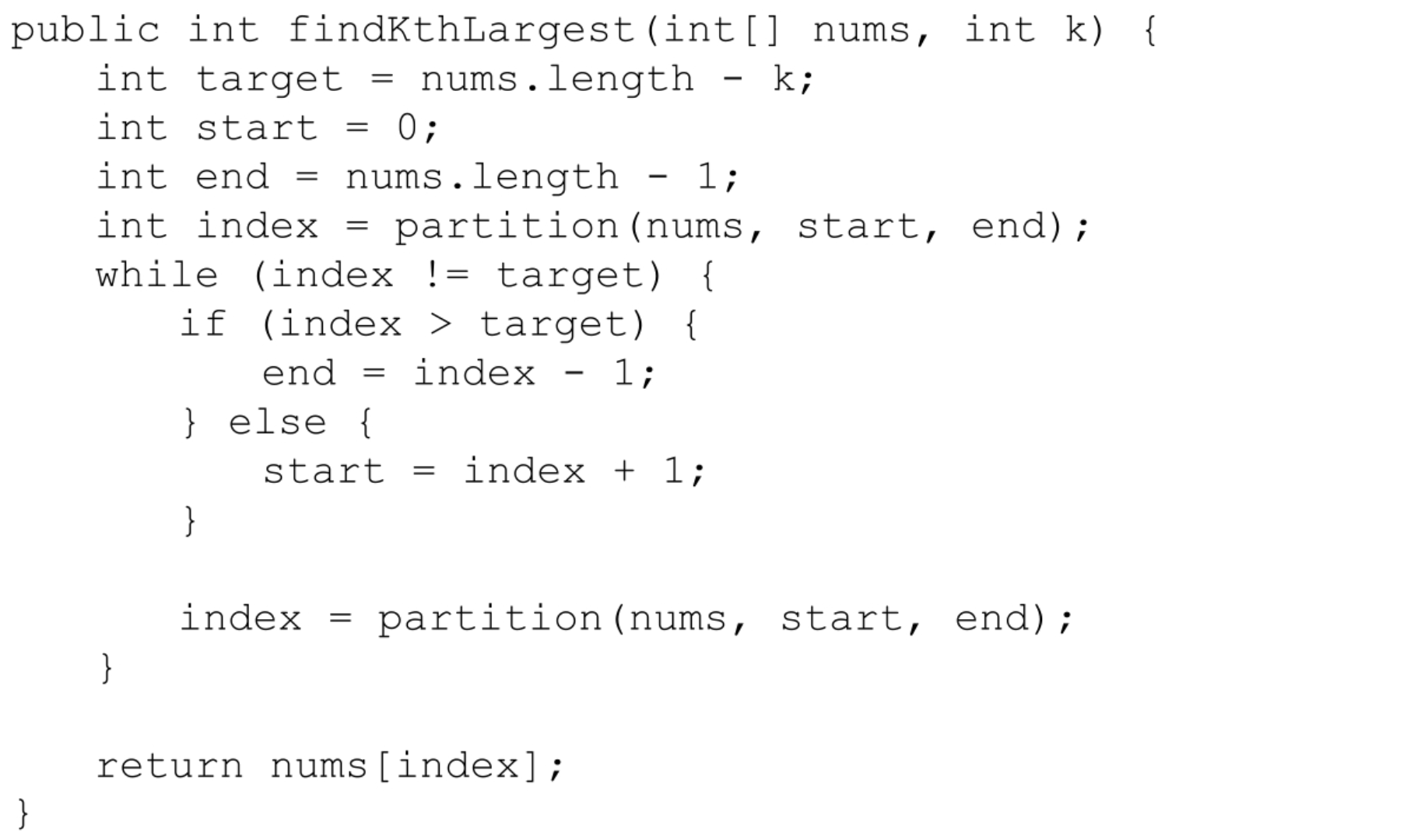
分析：面试题59中介绍过一种基于最小堆的解法。这种解法每次从数据流中读取一个数字并将其与位于最小堆堆顶的数字进行比较，当新读取的数字大于堆顶的数字时，删除堆顶的数字并将新数字添加到堆中。只要确保最小堆的大小为k，那么位于堆顶的数字就是第k大的数字。从数据流中读取n个数字并找出第k大的数字的时间复杂度是O（nlogk），空间复杂度是O（k）。

面试题59中的数据位于一个数据流中，不能一次性地将所有数据全部读入内存。而本题不一样，数据都保存在一个数组中，所有操作都在内存中完成。我们有更快找出第k大的数字的算法。

在长度为n的排序数组中，第k大的数字的下标是n-k。下面用快速排序的函数partition对数组分区，如果函数partition选取的中间值在分区之后的下标正好是n-k，分区后左边的值都比中间值小，右边的值都比中间值大，即使整个数组不是排序的，中间值也肯定是第k大的数字。

如果函数partition选取的中间值在分区之后的下标大于n-k，那么第k大的数字一定位于中间值的左侧，于是再对中间值左侧的子数组分区。类似地，如果函数partition选取的中间值在分区之后的下标小于n-k，那么第k大的数字一定位于中间值的右侧，于是再对中间值右侧的子数组分区。重复这个过程，直到函数partition的返回值正好是下标为n-k的位置。

上述过程可以用如下所示的参考代码实现：



上述代码中的函数partition就是快速排序的partition函数，此处不再重复它的代码。基于函数partition找出数组中第k大的数字的时间复杂度是O（n），空间复杂度是O（1）。

由于函数partition随机选择中间值，因此它的返回值也具有随机性，计算这种算法的时间复杂度需要运用概率相关的知识。此处仅计算一种特定场合下的时间复杂度。假设函数partition每次选择的中间值都位于分区后的数组的中间位置，那么第1次函数partition需要扫描长度为n的数组，第2次需要扫描长度为n/2的子数组，第3次需要扫描长度为n/4的数组，重复这个过程，直到子数组的长度为1。由于n+n/2+n/4+…+1=2n，因此总的时间复杂度是O（n）。

## 12.4 归并排序

归并排序也是一种基于分治法的排序算法。为了排序长度为n的数组，需要先排序两个长度为n/2的子数组，然后合并这两个排序的子数组，于是整个数组也就排序完毕。

归并排序可以用迭代代码实现。例如，输入一个长度为8的数组[4，1，5，6，2，7，8，3]，可以先合并相邻的长度为1的子数组得到4个排序的长度为2的子数组，如图12.3（a）所示。图12.3中的箭头表示源数据位于上面的数组中，合并时将数字写入下面的数组中。然后合并相邻的长度为2的子数组得到2个排序的长度为4的子数组，如图12.3（b）所示。此时源数据位于下面的数组中，合并时将数字写入上面的数组中。最后合并相邻的长度为4的子数组，此时整个数组排序完毕，如图12.3（c）所示。

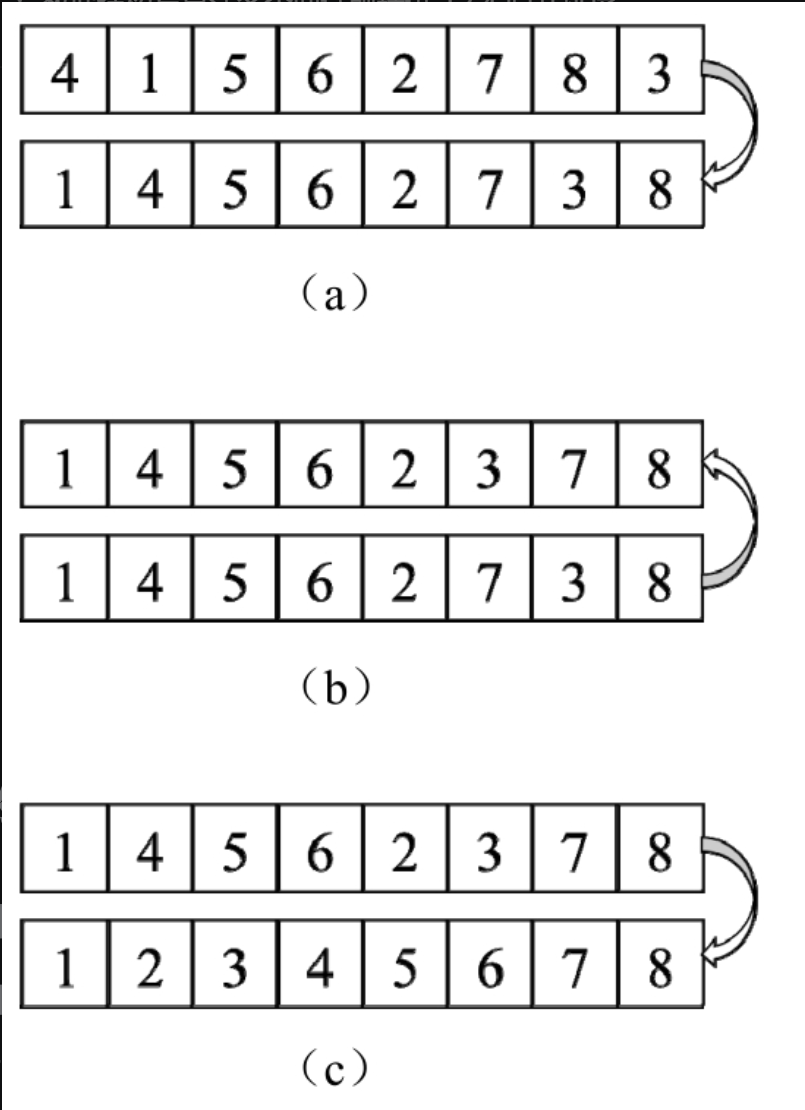
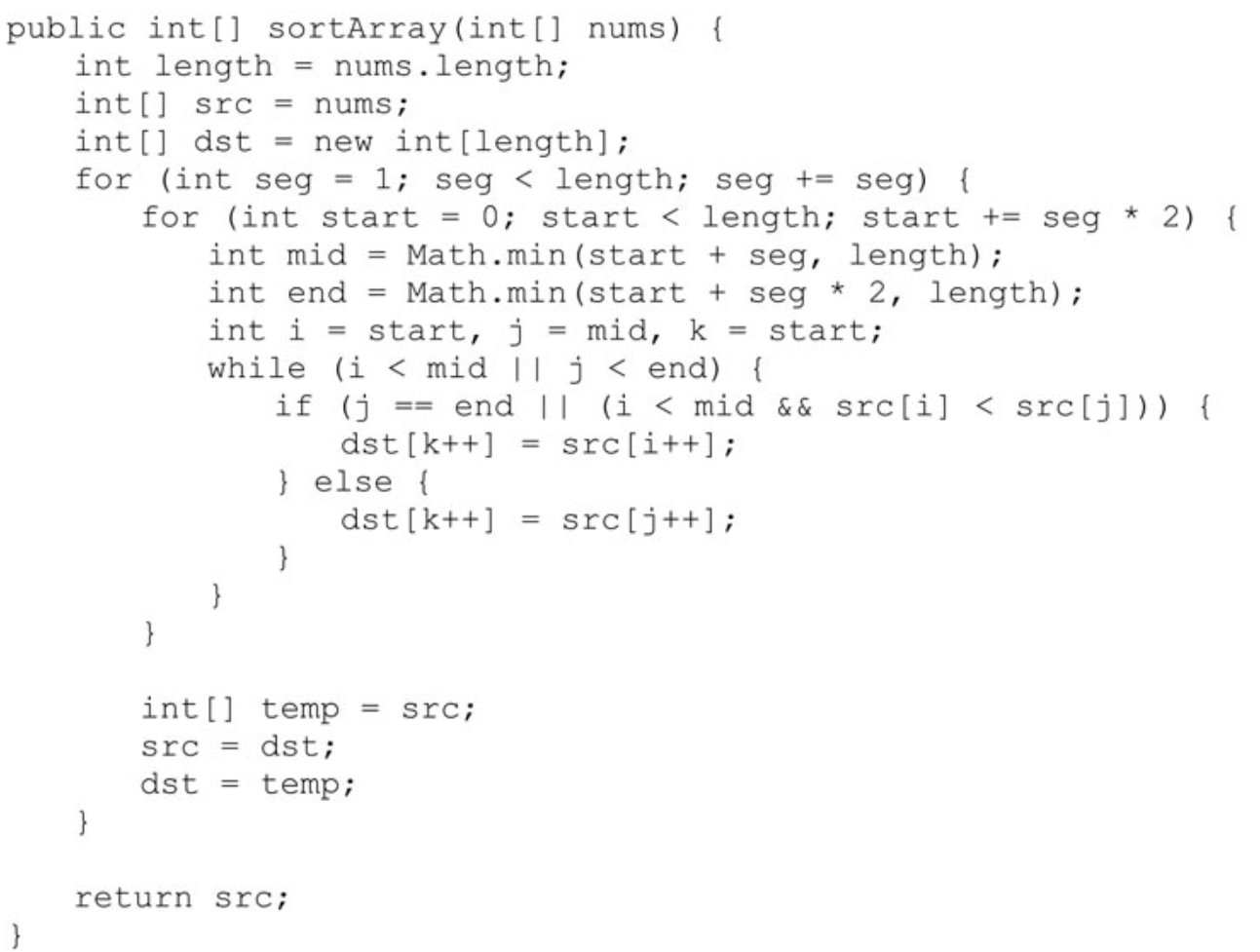


图12.3 归并排序的过程

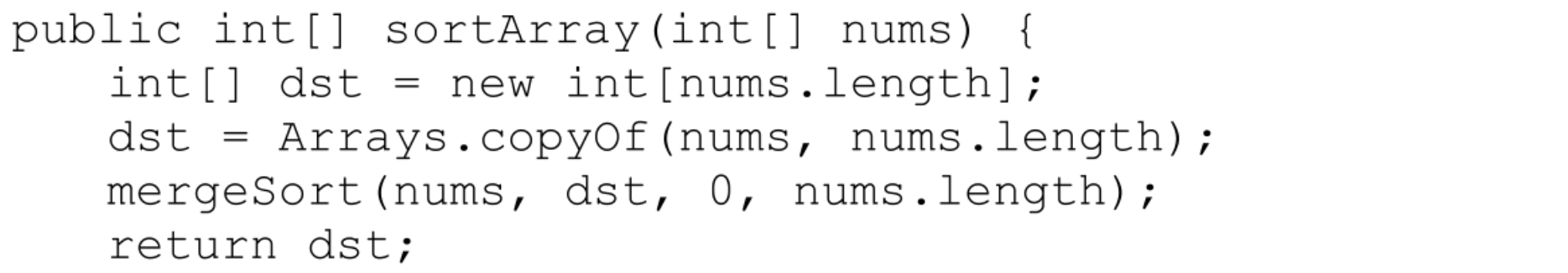
说明：（a）合并相邻的长度为1的子数组得到排序的长度为2的子数组；（b）合并相邻的长度为2的子数组得到排序的长度为4的子数组；（c）合并相邻的长度为4的子数组得到排序的长度为8的数组归并排序需要创建一个和输入数组大小相同的数组，用来保存合并两个排序子数组的结果。数组src用来存放合并之前的数字，数组dst用来保存合并之后的数字。每次在完成合并所有长度为n的子数组之后开始新一轮合并长度为2n的子数组之前，交换两个数组。

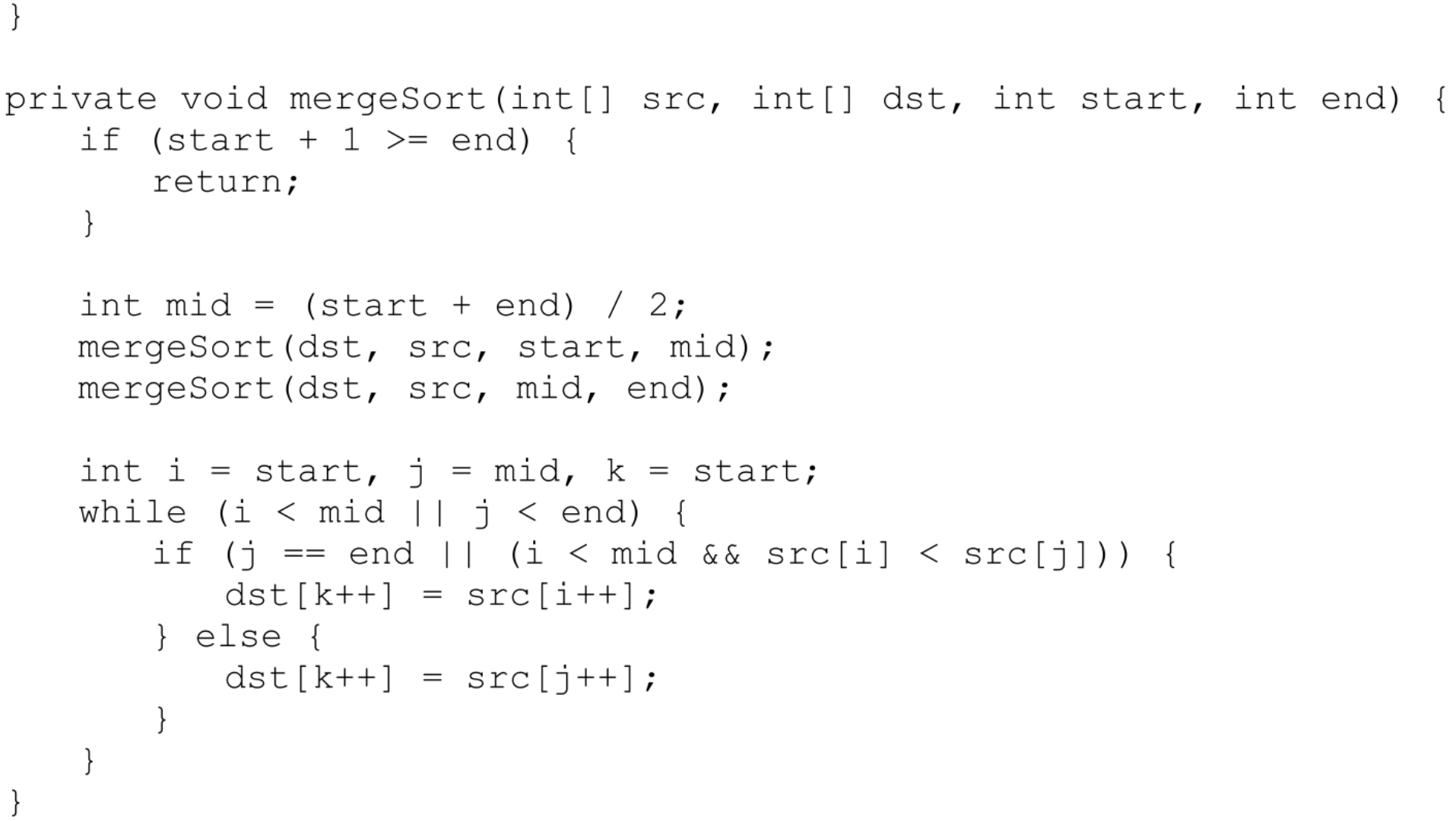
上述过程可以用如下所示的参考代码实现：



假设某一时刻准备合并数组src中从下标start开始的两个长度为seg的子数组，第1个子数组的起始下标是start，结束下标是start+seg-1；第2个子数组的起始下标是start+seg，结束下标是start+seg\*2-1。变量i、j是分别指向数组src中两个子数组的下标，它们从左到右扫描两个子数组，变量k是指向数组dst的下标。每次从数组src的两个子数组中选择将较小的数字写入数组dst中，最终数组dst中下标从start到start+seg\*2-1的子数组就是排序的。

归并排序也可以用递归的代码实现。为了排序长度为n的数组，只需要排序两个长度为n/2的子数组，然后合并两个排序的子数组即可。排序长度为n/2的子数组和排序长度为n的数组是同一个问题，可以递归调用同一个函数解决。归并排序的递归代码如下所示：

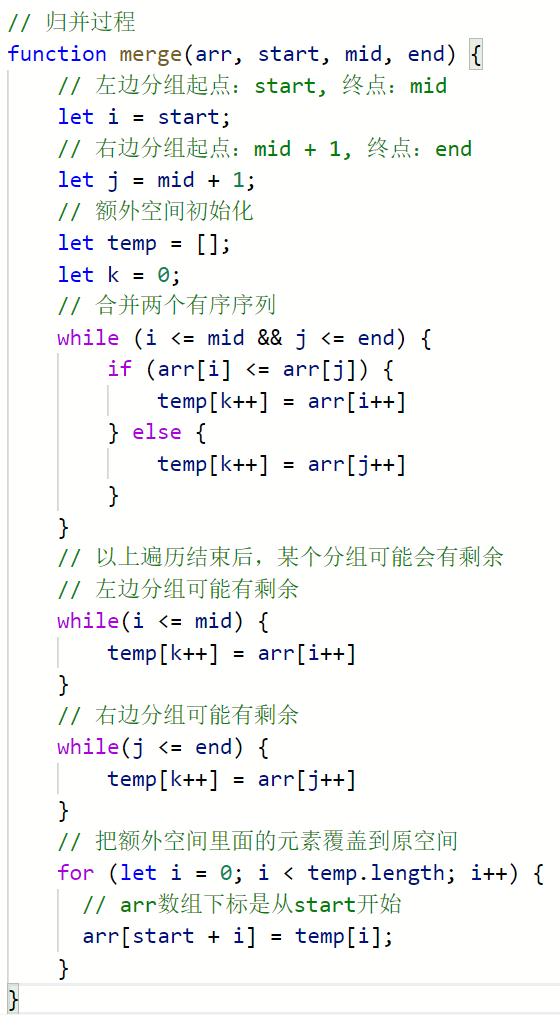
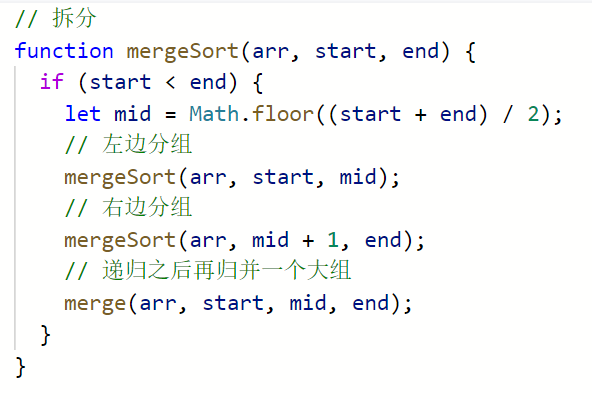




由于长度为n的数组每次都被分为两个长度为n/2的数组，因此不管输入什么样的数组，归并排序的时间复杂度都是O（nlogn）。归并排序需要创建一个长度为n的辅助数组。如果用递归实现归并排序，那么递归的调用栈需要O（logn）的空间。因此，归并排序的空间复杂度是O（n）。

手写归并排序的代码本身就是很常见的面试题。因此，应聘者应深刻理解归并排序的过程，熟悉归并排序的迭代和递归的代码实现。同时，归并排序是应用分治法来解决问题的，类似的思路可以用来解决很多其他的问题。

JavaScript版本：



### 面试题77：链表排序

题目：输入一个链表的头节点，请将该链表排序。例如，输入图12.4（a）中的链表，该链表排序后如图12.4（b）所示。

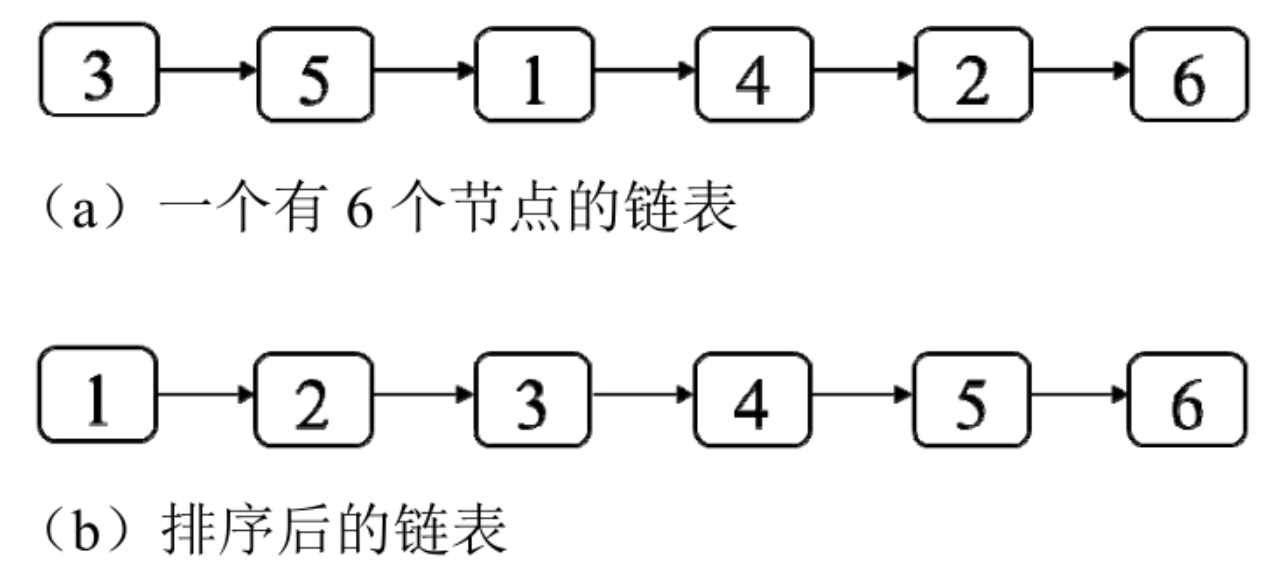


图12.4 链表排序

分析：前面讨论的排序算法的输入都是数组。这个问题的输入是一个链表，所以需要找到一个最适合链表的排序算法。

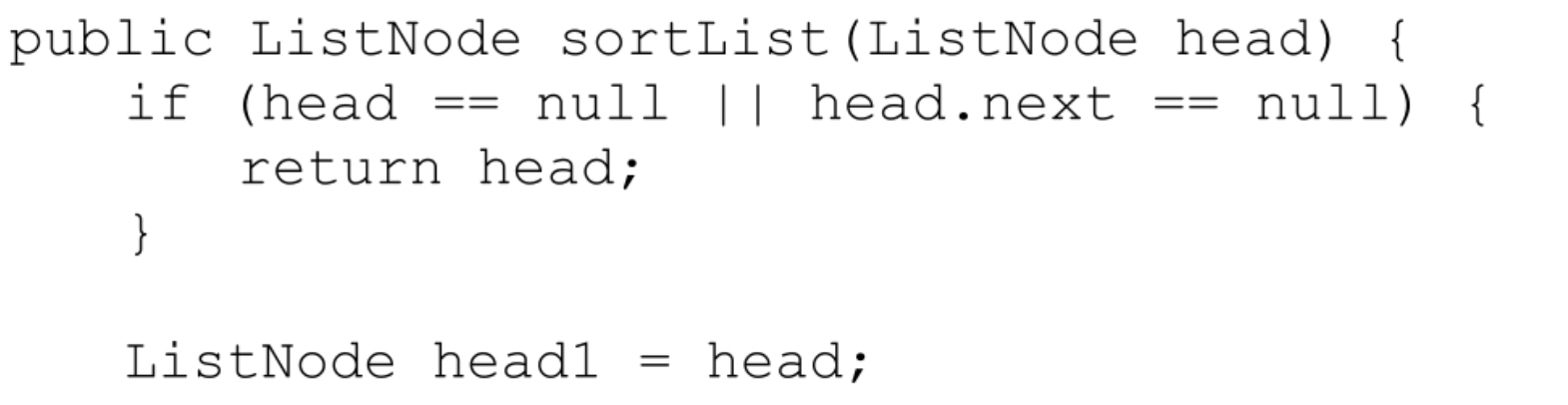
由于题目没有限定数字的范围，因此计数排序就不太适合。但可以考虑使用插入排序、冒泡排序等算法对链表进行排序，这些算法比较直观，实现起来也比较简单。只是这些算法的时间复杂度是O（n2），未必是最高效的算法。

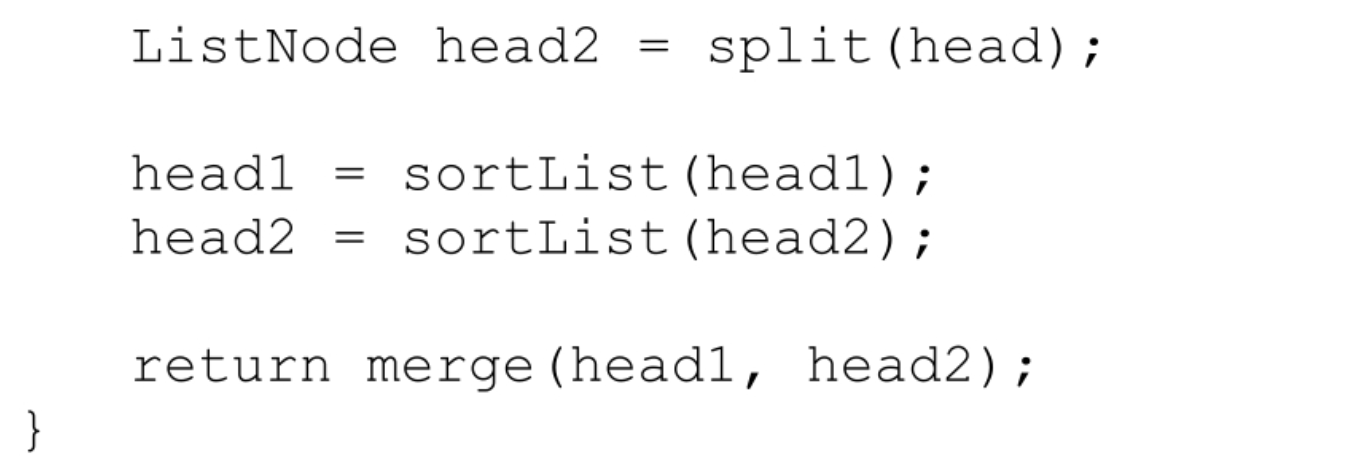
接下来考虑对数组进行排序的时间复杂度为O（nlogn）的排序算法，常用的有堆排序、快速排序和归并排序。

如果输入的是一个数组，那么堆排序用数组实现最大堆，该排序算法每次取出其中的最大值，再调整剩余的最大堆，直到所有数字都被取出。第9章已经介绍了如何用数组实现堆，其本质是将堆中的节点进行编号，数组的下标与节点的编号对应。可以根据某个数字的下标计算其父节点或子节点在数组中的下标。在数组中只需要O（1）的时间就能根据下标找到一个数字，但在链表中需要O（n）的时间才能根据节点的编号找到对应的节点。因此，不可能直接利用链表实现堆排序，但是如果链表的长度为n就可以创建一个长度为n的数组来实现堆，也就是说，通过O（n）的空间代价来实现堆排序。

接下来考虑快速排序。通常，快速排序算法首先随机生成一个下标，并以该下标对应的值作为中间值进行分区。如果输入的是数组，那么只需要O（1）的时间就能根据下标找到一个数字。但如果输入的是链表，那么需要O（n）的时间才能根据节点的编号找到对应的节点。快速排序也可以考虑不用随机的中间值，而是始终以某个固定位置的值作为中间值（如链表的头节点或尾节点），这样可能会出现每次分区时两个子链表的大小都不均衡，从而使时间复杂度退化为O（n2）。因此，虽然可以用快速排序算法对链表进行排序，但不如对数组排序高效。

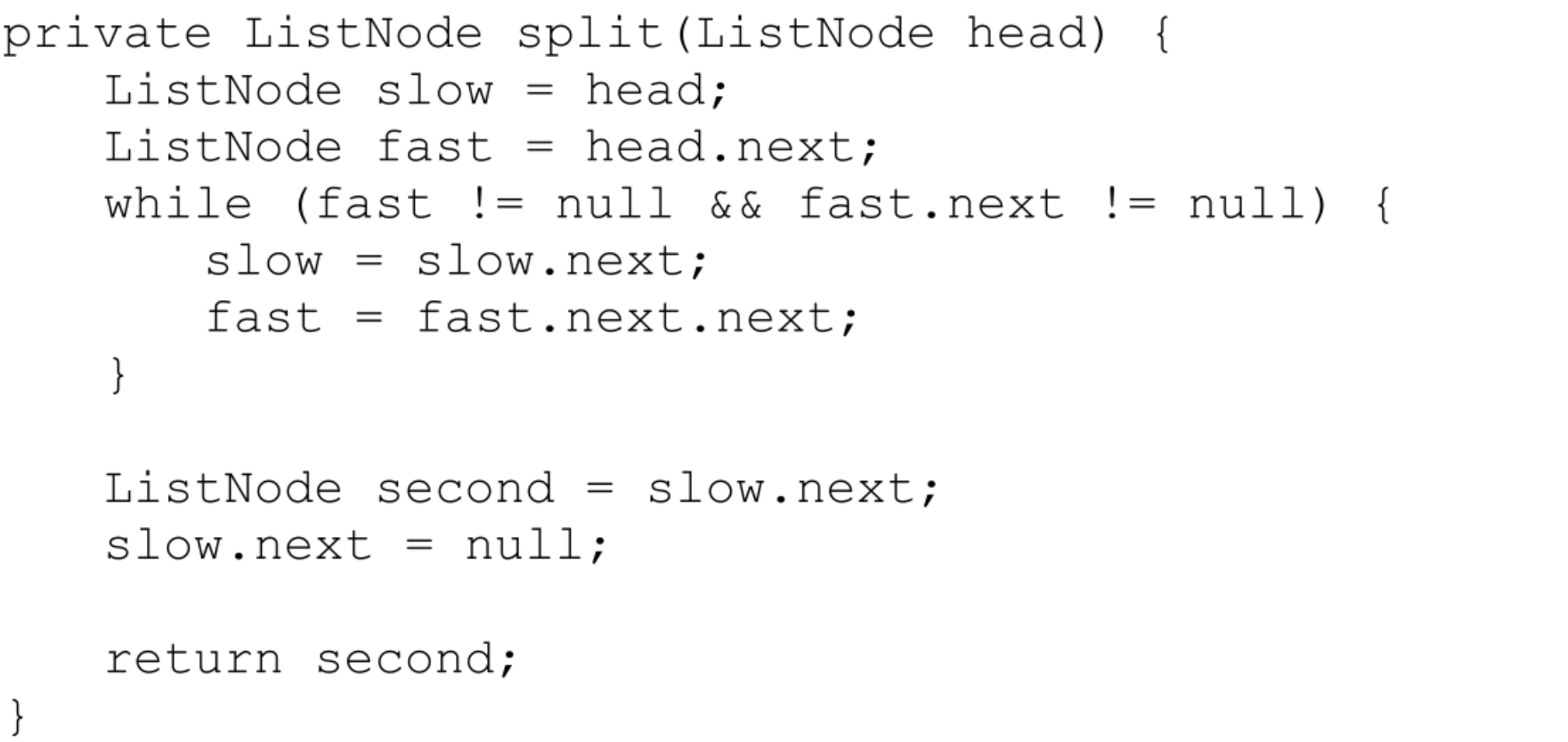
那么归并排序是否适合链表？归并排序的主要思想是将链表分成两个子链表，在对两个子链表排序后再将它们合并成一个排序的链表。这看起来没有什么问题，所以可以尝试基于归并排序算法对链表进行排序。用递归实现链表归并排序的参考代码如下所示：



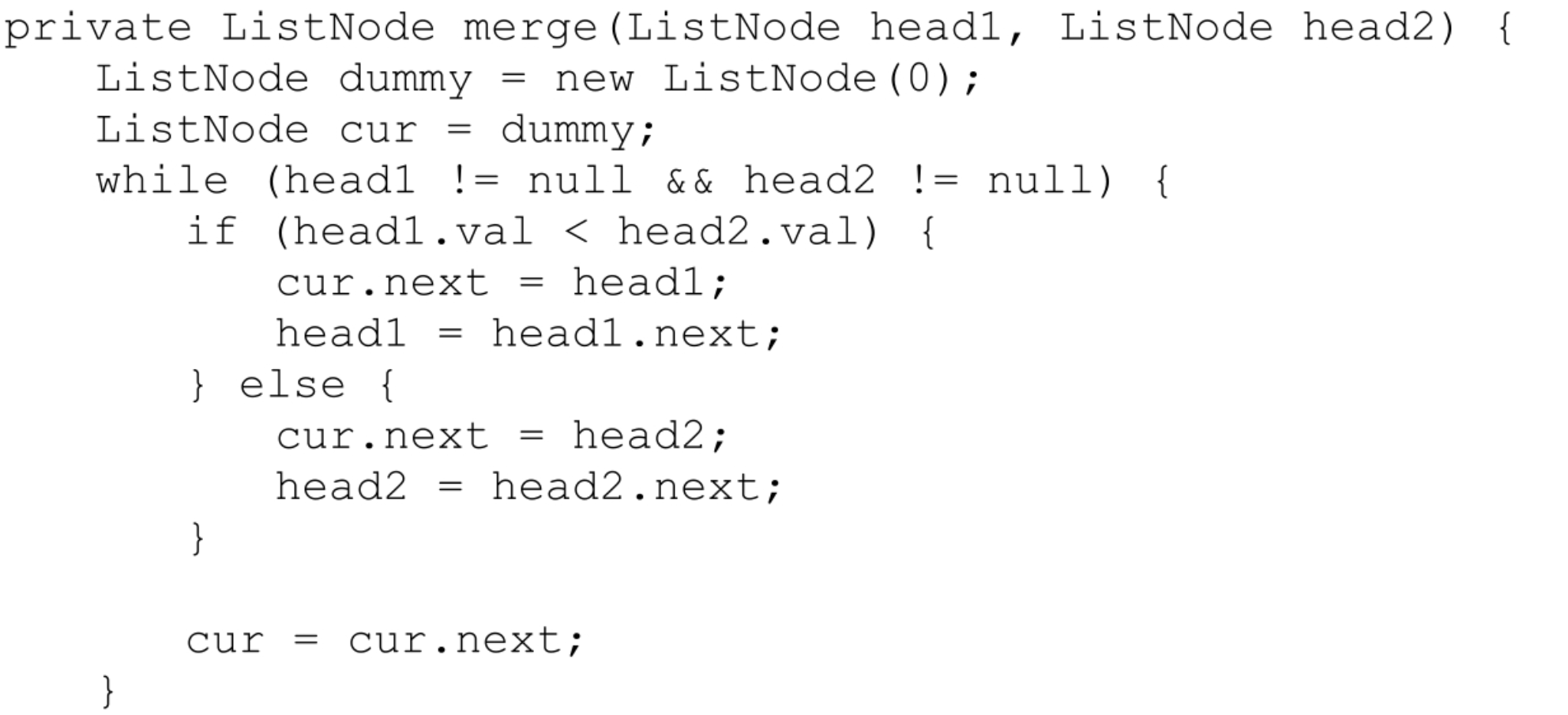


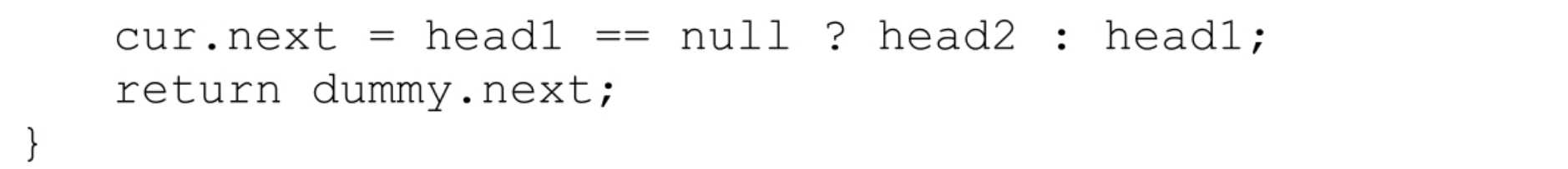
上述代码中的函数split将链表分成两半并返回后半部分链表的头节点。再将链表分成两半后分别递归地将它们排序，然后调用函数merge将它们合并起来。接下来讨论函数split和merge的实现细节。

第3章详细讨论了双指针。这里可以用快慢双指针的思路将链表分成两半。如果慢指针一次走一步，快指针一次走两步，当快指针走到链表尾部时，慢指针只走到链表的中央，这样也就找到了链表后半部分的头节点。函数split的参考代码如下所示：



和合并两个排序的子数组类似，也可以用两个指针分别指向两个排序子链表的节点，然后每次选择其中值较小的节点。与合并数组不同的是，不需要另外一个链表来保存合并之后的节点，而只需要调整指针的指向。函数merge的参考代码如下所示：

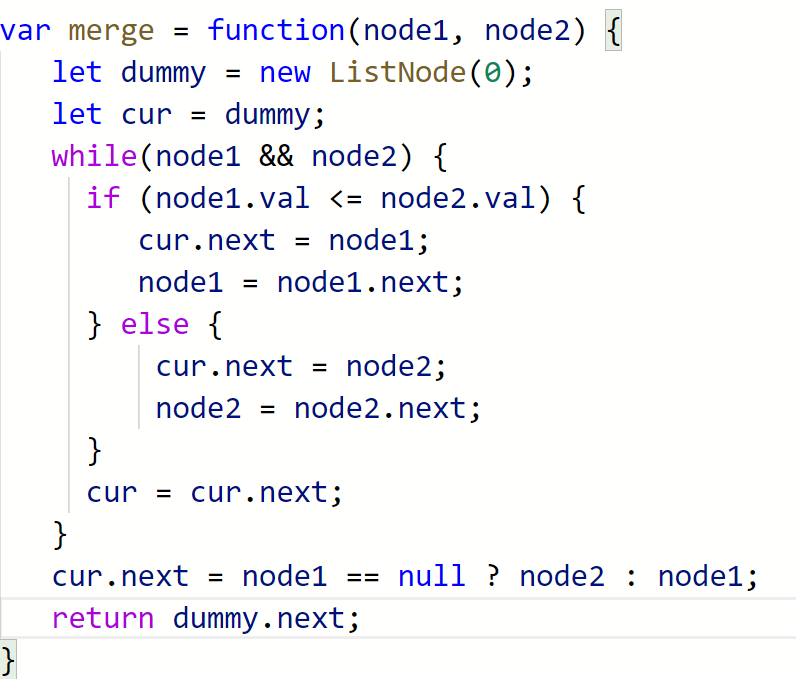
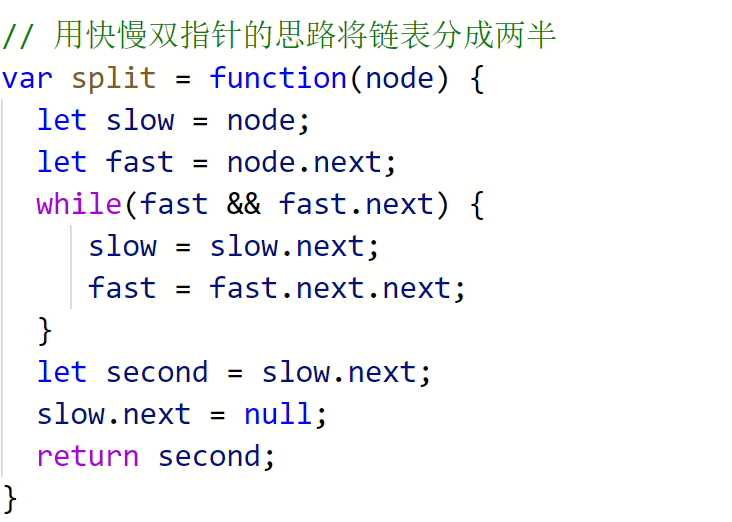




上述代码可以很好地实现归并排序的思想，它的时间复杂度是O（nlogn）。由于对链表进行归并排序不需要创建另外一个相同大小的链表来保存合并之后的节点，因此对链表进行归并排序的空间效率更高。由于代码存在递归调用，递归调用栈的深度为O（logn），因此空间复杂度为O（logn）。如果改用迭代的代码实现上述归并排序的过程，那么可以将空间复杂度优化到O（1），感兴趣的读者不妨一试。

JavaScript版本：





### 面试题78：合并排序链表

题目：输入k个排序的链表，请将它们合并成一个排序的链表。例如，输入3个排序的链表，如图12.5（a）所示，将它们合并之后得到的排序的链表如图12.5（b）所示。

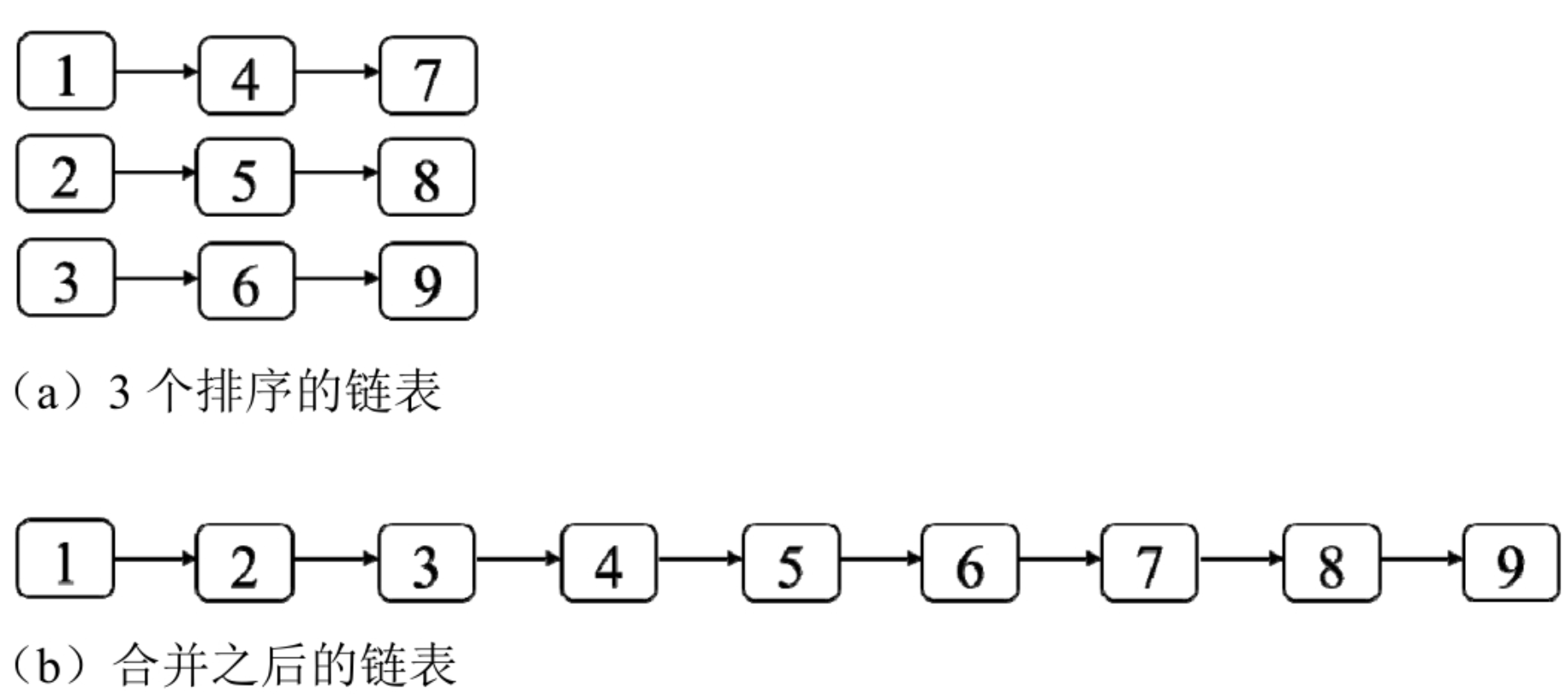


图12.5 合并排序链表

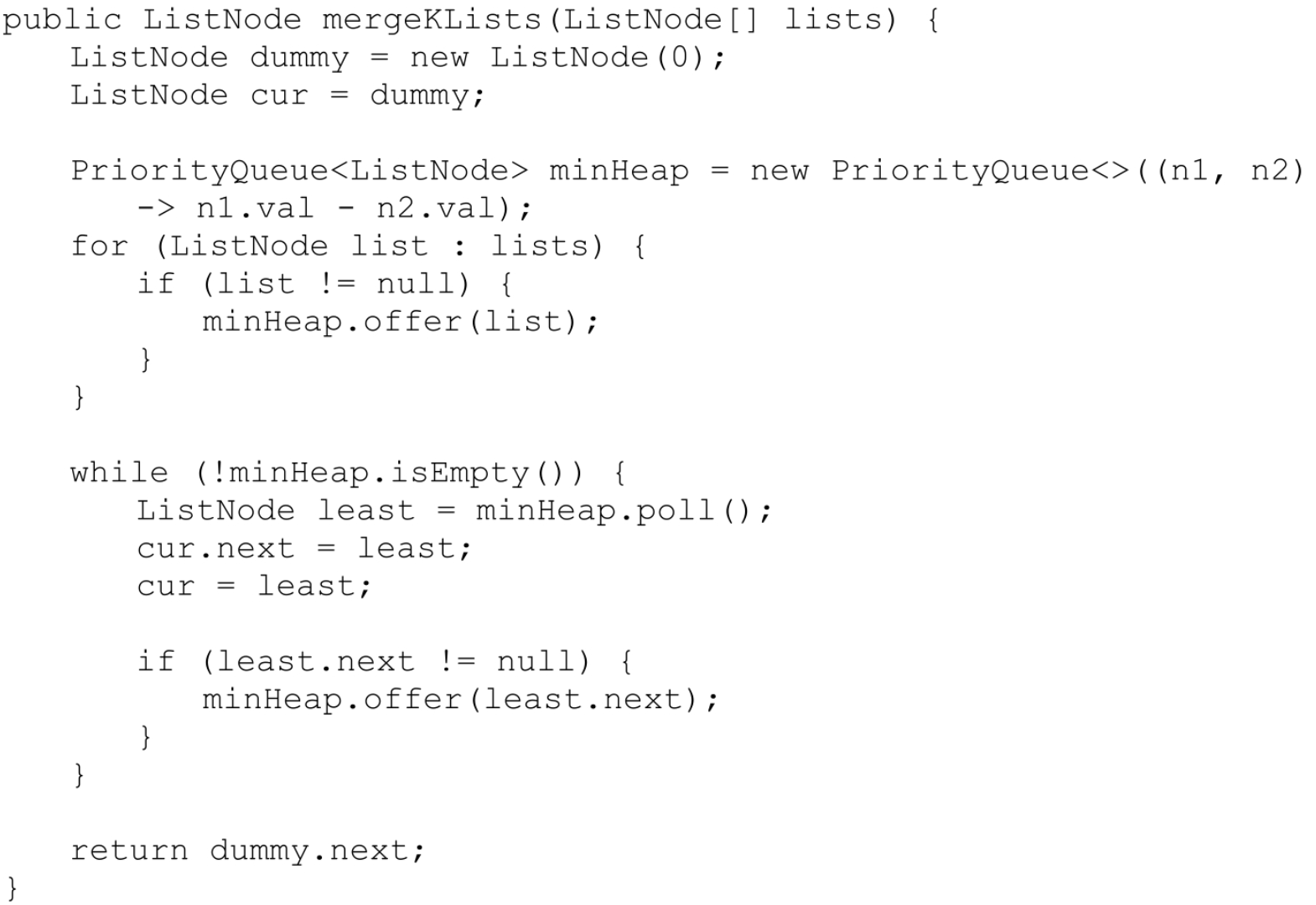
分析：在解决面试题77时需要合并两个排序的链表。这个题目将其扩展到合并任意k个排序的链表，所以仍然可以用类似的思路解决这个问题。

**利用最小堆选取值最小的节点**

**用k个指针分别指向这k个链表的头节点，每次从这k个节点中选取值最小的节点。然后将指向值最小的节点的指针向后移动一步，再比较k个指针指向的节点并选取值最小的节点。重复这个过程，直到所有节点都被选取出来。**

**这种思路需要反复比较k个节点并选取值最小的节点。既可以每次都用一个for循环用O（k）的时间复杂度比较k个节点的值，也可以将k个节点放入一个最小堆中，位于堆顶的节点就是值最小的节点。每当选取某个值最小的节点之后，将它从堆中删除并将它的下一个节点添加到堆中。从最小堆中得到位于堆顶的节点的时间复杂度是O（1），堆的删除和插入操作的时间复杂度是O（logk），因此使用最小堆比直观地用for循环的时间效率更高。**

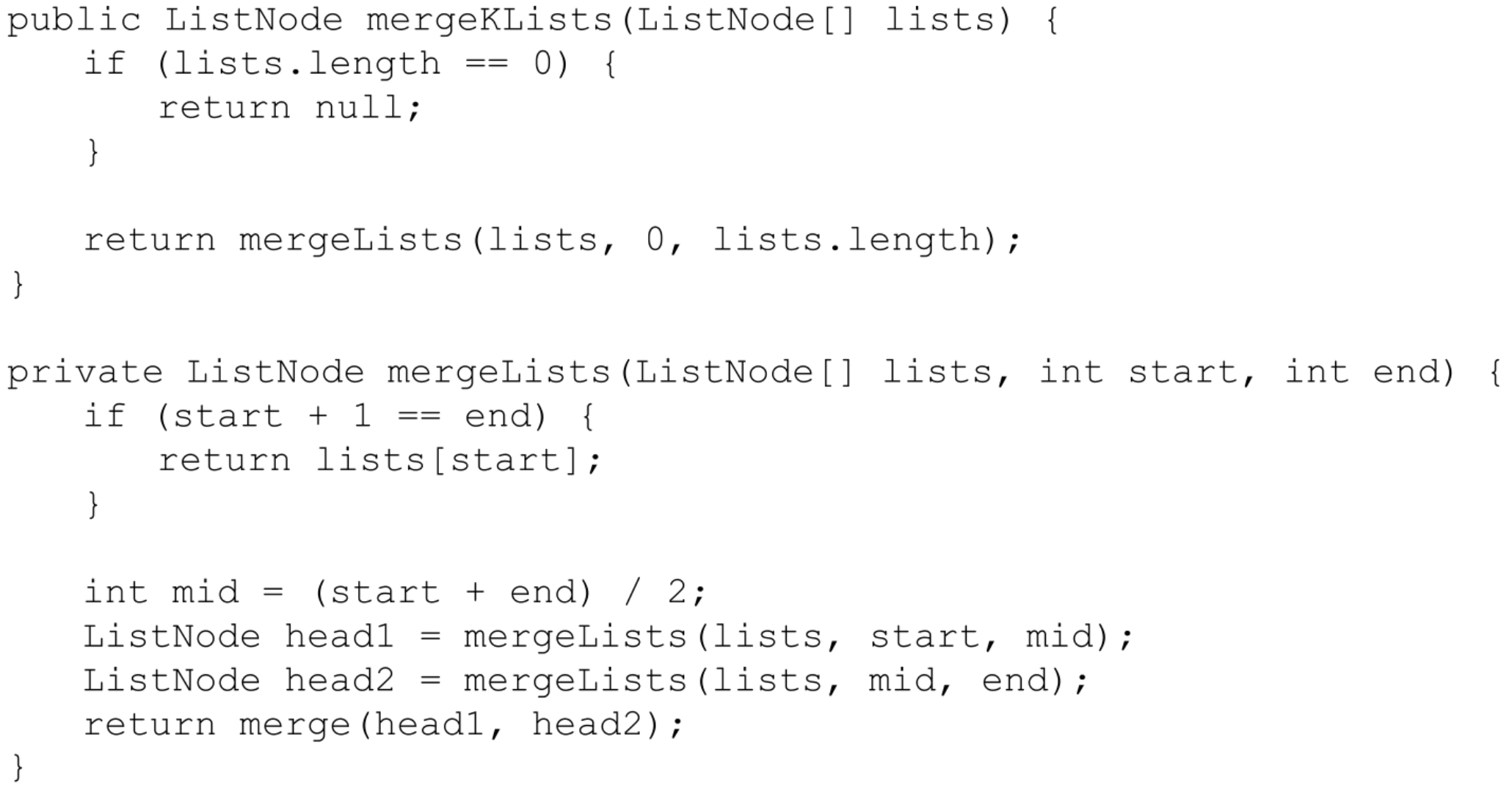
**使用最小堆的参考代码如下所示：**



**假设k个排序链表总共有n个节点。如果堆的大小为k，那么空间复杂度就是O（k）。每次用最小堆处理一个节点需要O（logk）的时间，因此这种解法的时间复杂度是O（nlogk）。**

**按照归并排序的思路合并链表**

**下面换一种思路来解决这个问题。输入的k个排序链表可以分成两部分，前k/2个链表和后k/2个链表。如果将前k/2个链表和后k/2个链表分别合并成两个排序的链表，再将两个排序的链表合并，那么所有链表都合并了。合并k/2个链表与合并k个链表是同一个问题，可以调用递归函数解决。这正是归并排序的思路，可以用如下所示的参考代码实现：**



**上述代码中的函数merge用来合并两个排序的链表。它的代码和面试题77中的函数merge的代码一样，此处不再重复介绍。**

**上述代码递归调用的深度是O（logk），每次需要合并n个节点，因此时间复杂度是O（nlogk）。它的空间复杂度是O（logk），用来维护递归调用栈。**

## **12.5 本章小结**

**本章介绍了几种不同的排序算法，包括计数排序、快速排序和归并排序。**

**如果整数在一个有限的范围内，那么可以先统计每个整数出现的次数，然后按照从小到大的顺序根据每个整数出现的次数写入输出数组中。如果n个整数的范围是k，那么计数排序的时间复杂度是O（n+k）。当k较小时，计数排序是非常高效的排序算法。**

**快速排序随机地从数组中选取一个中间值，然后对数组分区，使比中间值小的数值都位于左边，比中间值大的数值都位于右边，接下来将左右两边的子数组分别排序即可。快速排序的平均时间复杂度是O（nlogn）。**

**快速排序的函数partition可以用来选取第k大的数值。**

**归并排序将输入数组分成两半，在分别将左右两个子数组排序之后再将它们合并成一个排序的数组。归并排序的时间复杂度是O（nlogn），空间复杂度是O（n）。**

**对数组进行归并排序的过程可以用来解决类似的问题，如对链表进行排序。**