Data Structures and Algorithms in Java, 6th Edition-Wiley(2014)

# 12 排序和选择

## 12.1 合并排序

迄今为止我们已经介绍了几种排序算法,包括插入排序(见章节3.1.2、7.6和9.4.1);选择排序(见章节9.4.1);冒泡排序(见练习C-7.51);和堆排序(见章节9.4.2)。在这一章,我们将介绍四个排序算法,称为合并排序,快速排序,桶排序,基数排序,然后讨论的各种算法的优点和缺点在12.4节。

### 12.1.1 分治法

我们在本章中所描述的中,前两个算法为合并排序和快速排序，使用递归算法设计模式为“分治法”。我们已经看到递归算法的力量（见第5章）。分治法包括以下三个步骤。

1. 分：如果输入的大小小于某个阀值（或者说是一个或者两个元素），直接使用一个简单的方法来解决问题并返回所选的解决方法。否则，将输入的数据分成两个或多个不相交的子集。
2. 治理：递归解决相关的子问题的子集。
3. 显示：结合子问题的解决方案将他们合并成全局最原始的问题。、

**使用分治法进行排序**

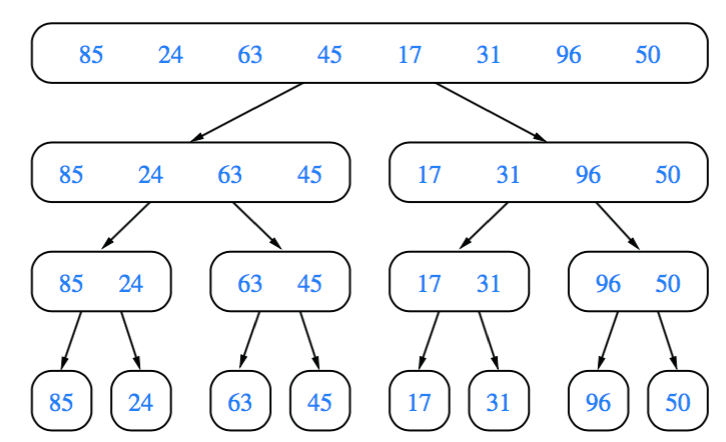
我们首先在搞级别中描述合并排序算法，而不是关注数据是否是一个数字或者列表（我们很快会创建一个实例），对序列S中有n个元素进行排序分为三个步骤——分别如下：

1. 分：如果拥有0个元素或者一个元素，立即返回S序列，它已经排序。否则则说明它已经排序。反之（S至少有两个元素）。从S中移除所有的元素并把它们放入两个序列中，S1和S2，每个序列大约包含S一般的元素，也就是说S1包含S前n/2的元素，S2包含S后n/2的元素。
2. 治理：递归排序序列S1和S2.
3. 结合： 把S1和S2序列中的元素合并排序到序列S中。

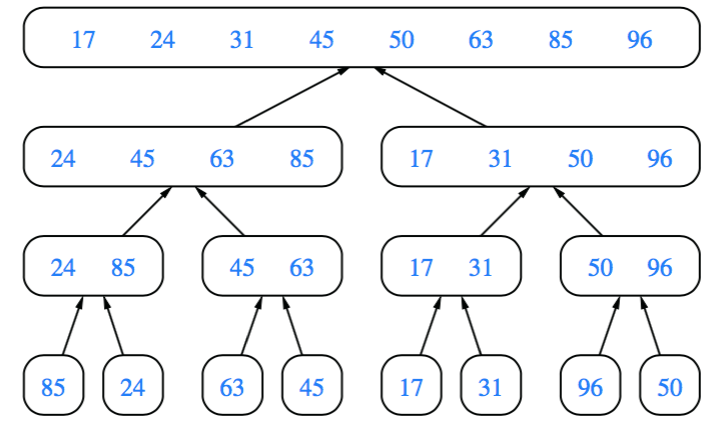
在分的步骤中，我们通过符号⌊x⌋表示数据的底线，最大的整数k，k ≤ x，同样的通过符号⌈x⌉ 表示x的上线，即最小的整数m，x ≤ m。

我们可以想象一个执行合并排序算法的二叉树T，称为合并排序树。T的每个节点代表一个递归调用（或回调）的合并排序算法。每个T的节点v代表S的处理和调用的过程，子节点v是调用子序列S1和S2的过程。T的外部节点相关S中的元素，对应的实例没有递归调用。

图12.1总结了排序算法中通过对每个节点的输入和输出的序列的处理过程。合并排序的下一步的树如图12.2至12.4所示。



（a）



（b）

图12.1：合并排序树T的一个算法包含8个序列元素：（a）输入序列在，每个节点处理（T）；（b）在每个节点的输出序列生成T。

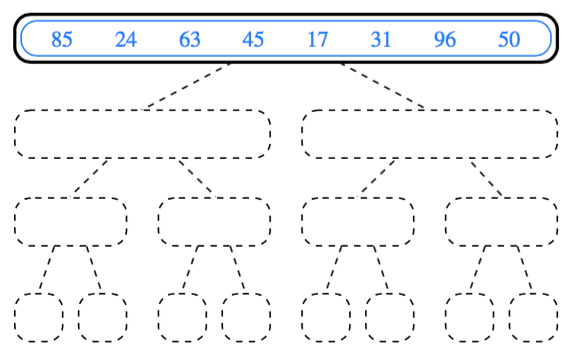
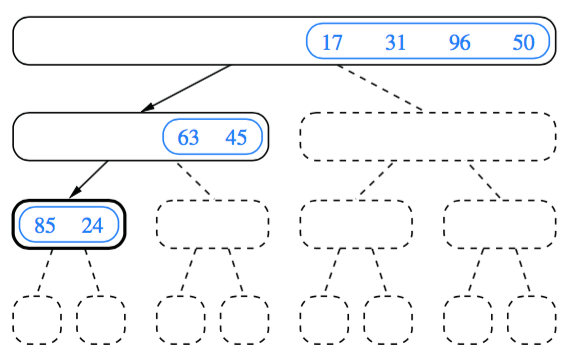
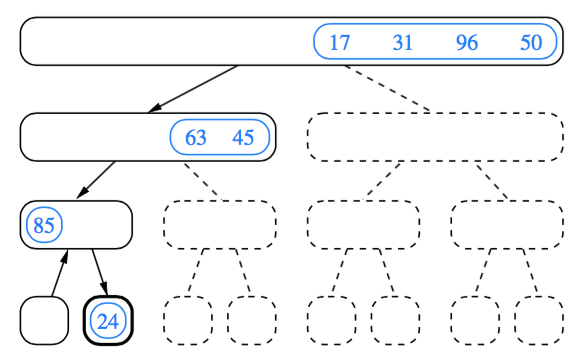
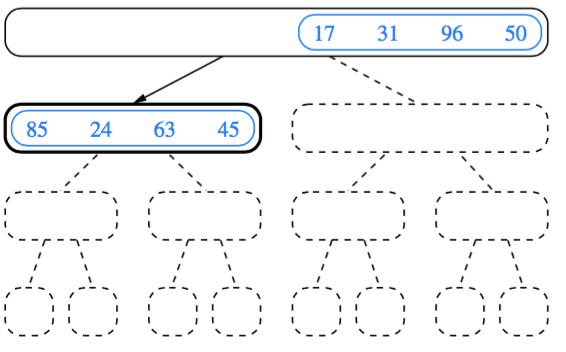
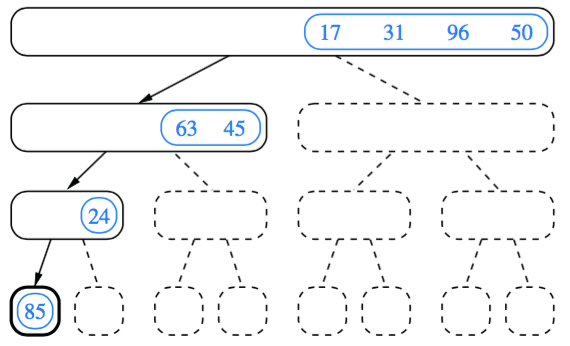
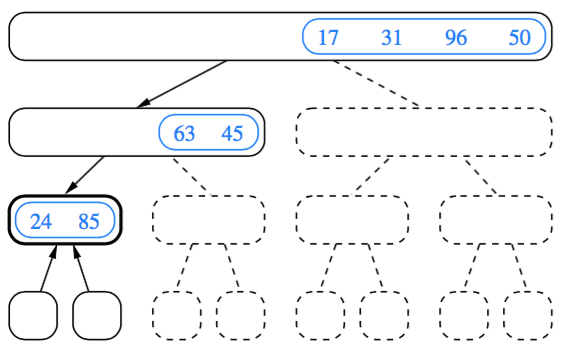
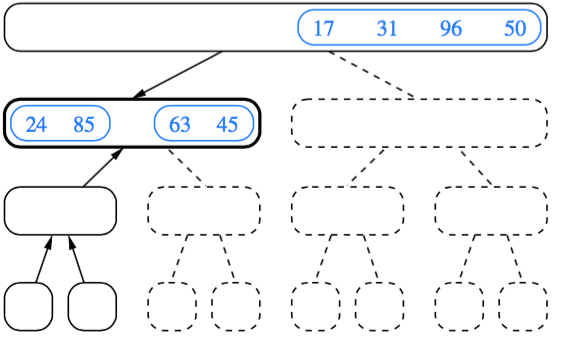
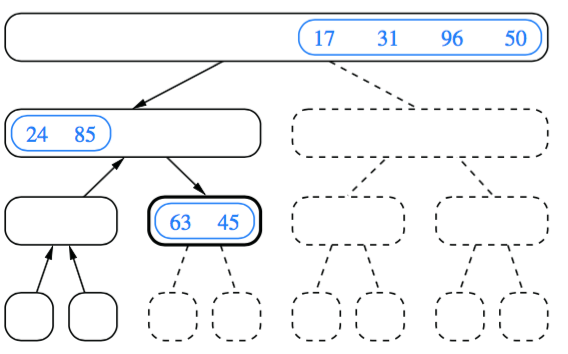
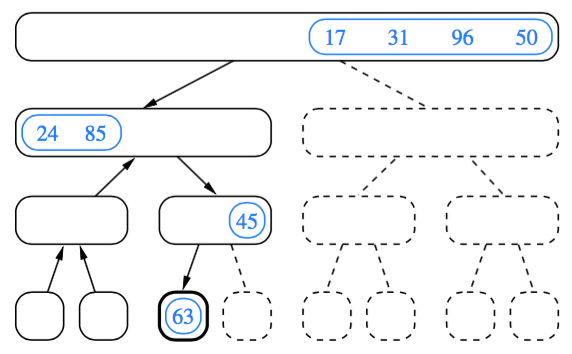
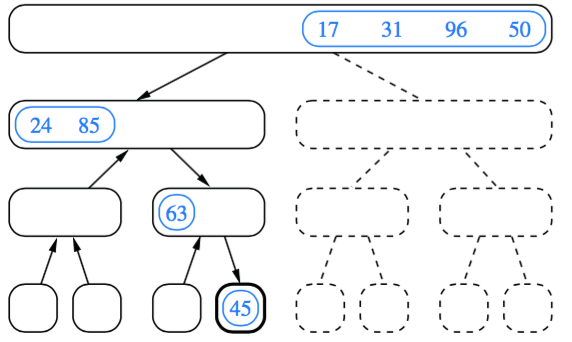
（a）（c）（e）（b）（d）（f）

图12.2：可视化的一个执行合并排序。树的每个节点表示一个合并排序的递归调用。节点用虚线发送调用。节点用粗线代表当前的回调。空节点用细线代表完成调用。剩余的节点(画细线和非空)代表调用正在等待一个子元素回调。(仍在图12.3)。

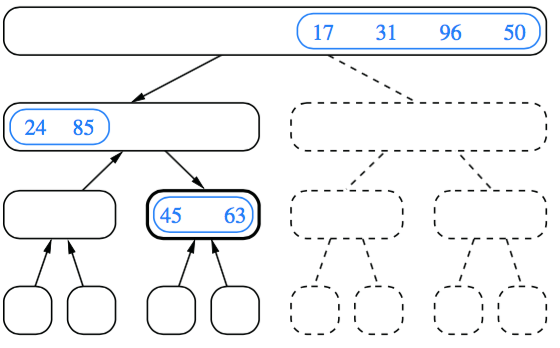
（g）

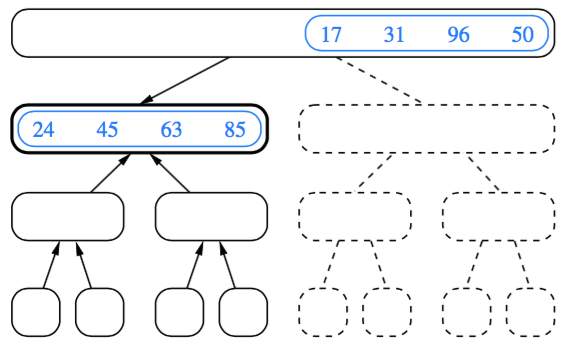
（h）



（i）

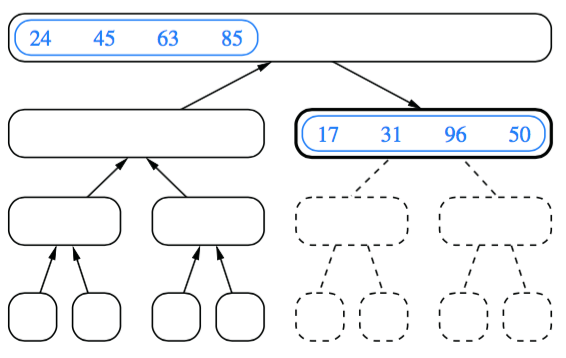
（j）

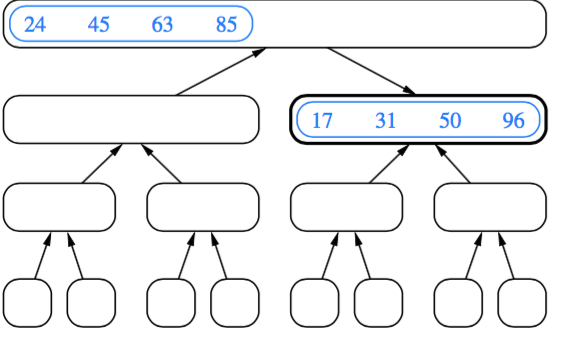


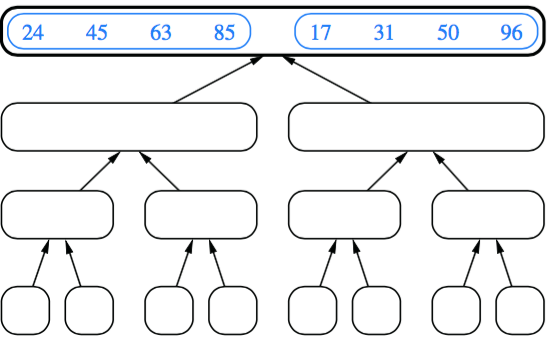
（k）

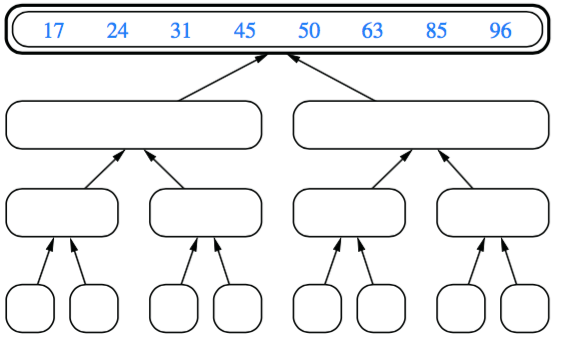
（l）

图12.3：可视化执行一个合并排序（结合图12.2和12.4）



（m）（n）



（o）

（p）

图12.4：可视化执行一个合并排序（从图12.4开始），几个回调之间省略（m）和（n），注意执行的两部分在步骤（p）中合并。

结论12.1： 合并的合并树与执行，排序书的高度为⌈log n⌉

我们用12.1做一个简单的练习（R-12.1）。我们将用这个练习分析合并排序算法的运行时间。

鉴于合并排序的概述,说明它是如何工作的,让我们考虑的每一个步骤的分治算法精简。一系列大小为n划分涉及分离的元素索引⌈n / 2⌉和递归调用可以开始通过这些小序列pa -参数。艰难的是结合两个排序序列为一个排序序列。因此,我们提出我们的合并排序的分析之前,我们需要说更多关于这是如何实现的。

### 12.1.2 基于数组的合并排序实现

我们首先关注的情况是下一个序列是否为数组。合并方法(12.1代码片段)负责合并两个排序序列,S1和S2,与输出复制到S中，我们在每一次循环复制一个元素的同时,应有条件地决定下一个元素从S1还是S2中。代码片段的分治合并排序算法12.2。

我们说明的合并过程步骤如图12.5所示。在过程中,索引i指向S1从S中复制的元素，索引j指向S2从S中复制的元素，假设S1和S2都有至少一个为标记的元素,我们复制较小的两个元素。从i+j指向的对象之前复制,下一个元素放置在S(i + j)中。(例如,当i + j = 0时,下一个元素复制到S[0])。如果我们达到的一个序列,我们必须从其他复制下一个元素。

/∗∗ Merge contents of arrays S1 and S2 into properly sized array S. ∗/

public static <K> void merge(K[ ] S1, K[ ] S2, K[ ] S, Comparator<K> comp) {

int i = 0, j = 0;

while (i + j < S.length) {

if (j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))

S[i+j] = S1[i++]; // copy ith element of S1 and increment i

else

S[i+j] = S2[j++]; // copy jth element of S2 and increment j

}

代码片段12.1： 实现一个进行合并操作的Java数组。

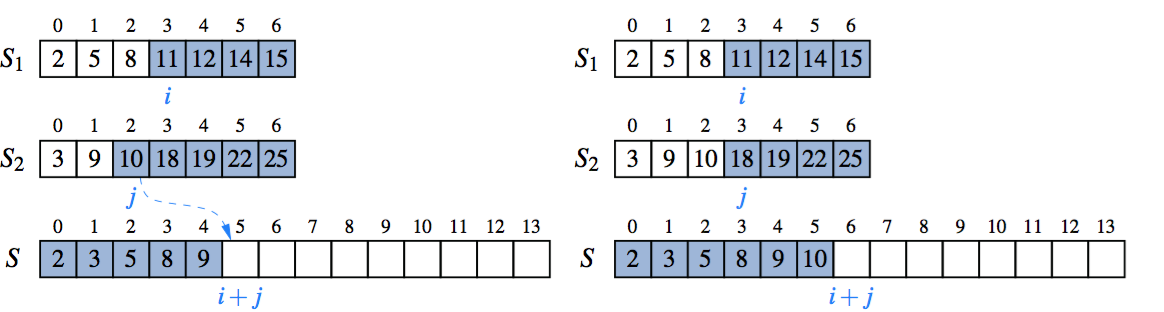


图12.5：合并两个数组排序的步骤。

/∗∗ Merge-sort contents of array S. ∗/

public static <K> void mergeSort(K[ ] S, Comparator<K> comp) {

int n = S.length; if (n < 2) return; // divide

int mid = n/2;

K[ ] S1 = Arrays.copyOfRange(S, 0, mid);

K[ ] S2 = Arrays.copyOfRange(S, mid, n); // conquer (with recursion) mergeSort(S1, comp);

mergeSort(S2, comp);

// merge results

merge(S1, S2, S, comp);

}

代码块12.2：通过递归合并的排序算法实现一个Java数组（合并方法定义在代码块12.1中）

我们注意,合并和归并排序方法依赖实例来比较两个的属于一个序列的对象。这样的方法我们在定义优先级队列9.2.2节中介绍过,当研究实现排序图10和11章。

### 12.1.3 合并排序的运行时间

我们首先分析合并算法的运行时间。让n1和n2分别为S1和S2的元素的数量。很明显,while循环的每一次操作都需要O(1)时间。在主要的过程中,在每个迭代的循环下，一个元素从S1和S2复制到S(元素不再被认为需要排序)。因此,循环的迭代次数是n1 + n2。因此,算法的运行时间是O(n1 + n2)。

在分析了用于组合子序列合并算法的运行时间后,让我们分析整个合并排序算法的运行时间,假设它是给定的是一个输入n个元素的序列。为简单起见,我们限制n是2的幂。我们不再是在练习中( R-12.3)表明我们的分析的结果中还有当n不是2的幂的情况

在评估合并排序递归时候,我们依赖5.2节中介绍的分析技术。我们占用的时间分布在每个递归调用,但不包括任何等待逐次递归调用终止的时间。归并排序的方法中,我们花费的时间划分为两个子序列时间,并调用了合并两个需要排序的序列,但我们并没有将两种需要排序的序列使用递归调用。

在二叉树T，图12.2到12.4中通过我们的分析。考虑一个需要递归调用与合并排序树T的中的一个节点v 。节点v的递归操作步骤很简单，通过创建两个子序列的方法，这一步和时间序列的大小成正比。但我们已经注意到这还需要时间是和线性合并后的序列的大小有关的。如果我们让我们表示节点v的深度,v的时间节点是O(n / 2^i),因为序列的大小由递归调用与v = n / 2^i决定。

在树T的全局中看来,如图12.6所示,我们看到,因为我们的定义了“一个时间节点”，合并排序的运行时间等于其之和乘以在T的节点。观察到T中正好有2^i的节点深度。这个简单的观察有一个重要的发现,它意味着整个时间T的节点深度是O(2^i·n / 2^i)。在命题12.1下,T的高度⌈logn⌉。因此，其需要花费的时间为⌈log n⌉ + 1 levels of T is O(n) 。我们有如下结果：

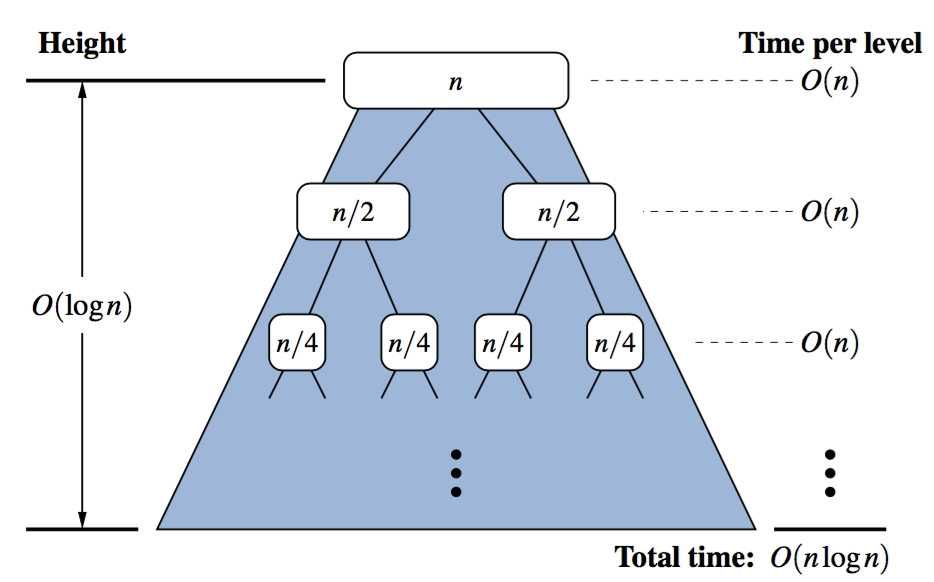
结论12.2：当S中的两个元素在O（1）中可以比较 ，则序列S的合并排序算法需要的时间复杂度为O(n log n)。

图12.6： 通过分析合并排序的运行时间，每个节点代表了了一个标注了子序列大小的递归调用。

### 12.1.4 合并排序的递归方程

还有另一种方法来证明合并排序算法的时间复杂度为O(n log n)（结论12.2）