第 3 章　基于比较的排序算法

一个数字序列，$a_1, a_2, a_3, \cdots, a_n$，如满足 $a_1 \leqslant a_2 \leqslant a_3 \leqslant \cdots \leqslant a_n$，则称为递增（或非递减）序列。类似地，如果该序列满足 $a_1 \geqslant a_2 \geqslant a_3 \geqslant \cdots \geqslant a_n$，则称为递减（或非递增）序列。把无序的 n 个数排成一个递增或递减的序列称为排序（sorting）。不言而喻，排序是一个非常重要的工作，许多算法问题都需要先对输入数据或中间结果作出排序。

如果我们只允许通过比较数字之间相对大小来作出排序决定，那么这种排序就称为基于比较的排序（comparison sorting）。基于比较的排序不允许算法知道和利用一个数字内在的构成，例如，这个数字是几位数，这个数字的第二位是多少等。我们常用的排序算法大都是基于比较的排序算法。人们在实践中提出了许多排序算法，但本章只讨论 4 种最常用、最重要的排序算法，即插入排序、合并排序、堆排序和快排序。假定要排序的 n 个数存放在数组 $A[1..n]$中。因为对称的关系，我们只讨论如何把它们排为一个递增序列，使得 $A[1] \leqslant A[2] \leqslant \cdots \leqslant A[n]$。

3.1　插入排序

插入排序（insertion sort）是最简单的一种排序方法。插入排序的做法是，先把第一个数 $A[1]$ 排好序，再把前两个数，$A[1]$ 和 $A[2]$，排好序，然后，把前三个数排好序，等等，直到 n 个数全排好。因为第一个数的排序不需任何比较，插入排序是从第二个数起进行比较。下面介绍详细做法。

3.1.1　插入排序的算法

假设我们已把前 $k-1$ 个数（$k \geqslant 2$），$A[1]$，$A[2]$，$\cdots$，$A[k-1]$，排好序，那么下一步要考虑如何把 $A[k]$插入到这 $k-1$ 个数中使前面 k 个数排好序。做法是，将 $A[k]$复制到一临时变量 x 中并将其与前面这 $k-1$ 个数从 $A[k-1]$开始向前依次比较。如果 $A[k-1] \leqslant x$，则比较停止，这时前 k 个数已排好。否则，继续把 x 与 $A[k-2]$比较，与 $A[k-3]$比较，$\cdots$，直至找到某个 $A[j]$（$j \geqslant 1$），使得 $A[j] \leqslant x$。显然，$A[k]$应该插在 $A[j]$ 和 $A[j+1]$之间。算法把原来在 $A[j+1]$，$A[j+2]$，$\cdots$，$A[k-1]$中数字顺序向右移动一个位置后将 x 中数置入 $A[j+1]$。如果这样的 j 不存在，那么 x 应置入 $A[1]$。这样的插入操作从 $k=2$ 做起直至 $k=n$，排序完成。以下是插入算法的伪码。

```
Insertion-Sort (A[1..n])
1    if n = 1
2        then exit
3    endif
4    for k ← 2 to n
5        x ←A[k]
6        j ← k – 1
7        while j > 0 and A[j] > x
8            A[j+1] ← A[j]
9            j ← j –1
10       endwhile
11       A[j+1] ← x
```

```
12    endfor
13    End
```

插入排序的正确性一目了然。下面我们分析其复杂度。

3.1.2 插入排序算法的复杂度分析

我们以序列中数字间的比较作为主要的基本运算来计算复杂度。最好的情况是输入数组 A 已经是一个递增序列。这时的复杂度为 $T(n) = n-1$，因为从 $k = 2$ 到 $k = n$，只需一次比较即可插入 $A[k]$。

最坏的情况是输入数组 A 是一个递减序列。这时 $A[k]$需要和 $A[1..k-1]$中每一个数比较后方可插入。因此最坏情况的复杂度为 $T(n) = \sum_{k=2}^{n}(k-1) = \frac{n(n-1)}{2} = O(n^2)$。

平均情况的复杂度可分析如下。我们先考虑将 $A[k]$ 插入所需要的平均比较次数。$A[k]$可能插在从 $A[1]$到 $A[k]$的任一位置。我们假定 $A[k]$插入这 k 个位置中任一个的概率都是 $1/k$。如图 3-1 所示，如果 $A[k]$是插在 $A[k]$位置，则一次比较就够了；如果 $A[k]$是插在 $A[k-1]$位置，则正好要二次比较；如果 $A[k]$是插在 $A[j]$位置则正好要（$k-j+1$）次比较。注意，$A[k]$插在 $A[1]$和 $A[2]$位置所需比较次数是相同的，均为 $k-1$，这是因为当 $A[k]$与 $A[1]$比较后即可确定它是在 $A[1]$或 $A[2]$位置。为方便起见，我们假定 $A[k]$插入 $A[1]$位置所需比较次数为 k。显然，这一简化不影响渐近复杂度。由此，我们得到插入 $A[k]$的平均复杂度为 $\frac{1}{k}\sum_{j=1}^{k} j = \frac{k+1}{2}$。这样整个算法的平均复杂度为：

$A(n) = \sum_{k=2}^{n}\frac{k+1}{2} = \Theta(n^2)$。

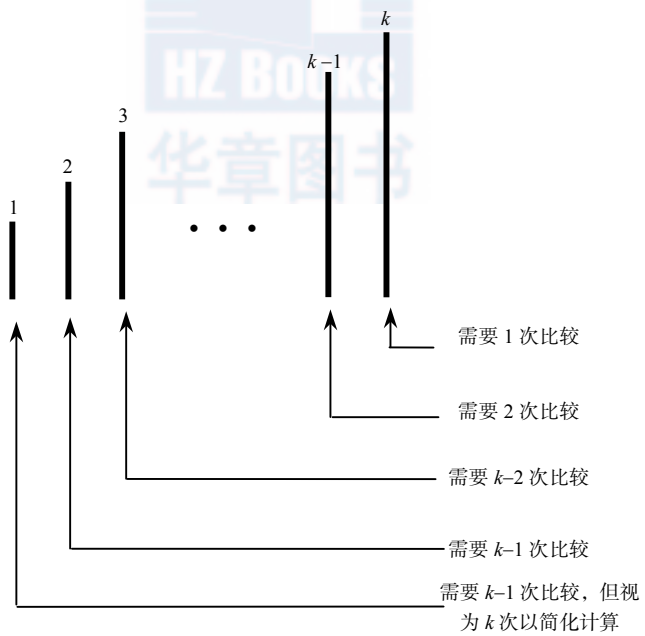


图 3-1 将 $A[k]$插入到各个位置所需比较次数

3.1.3 插入排序的优缺点

插入排序的复杂度较高，后面会介绍复杂度为$\Theta(n\lg n)$的算法，但是插入排序算法简单，实现容易，并且是一种**稳定**（stable）排序。一个排序算法称为是稳定的，如果序列中任意两个相等的

数在排序后不改变它们的相对位置。插入排序的另一个优点是，它是一个就地操作的算法。如果一个算法不需要使用或只需要使用常数个除输入数据所占有的空间以外的存储单元，我们称之为**就地操作**（in-place）的算法。除了数组 $A[1..n]$以外，插入排序只需要指针 k、指针 j 和临时工作单元 x，显然是就地操作的算法。

3.2 合并排序

因为插入排序最坏情况和平均情况的复杂度均为$\Theta(n^2)$，我们希望能找到更快的排序算法。合并排序（merge sort）有较小的复杂度。合并排序用的是分治法，它把一个要排序的序列一分为二，然后将这两个子序列分别递归地排序后再合并为一个完整的递增序列。这个算法的核心是将两个有序的子序列合并（merge）的算法。所以，我们先讲这个合并算法。

3.2.1 合并算法及其复杂度

假设 $A[1..n_1]$ 和 $B[1..n_2]$ 分别为两个递增序列，即

$$A[1] \leqslant A[2] \leqslant A[3] \leqslant \cdots \leqslant A[n_1]$$

$$B[1] \leqslant B[2] \leqslant B[3] \leqslant \cdots \leqslant B[n_2]$$

合并算法就是把这两个序列合并为一个递增序列 $C[1..n_1+n_2]$使得

$$C[1] \leqslant C[2] \leqslant C[3] \leqslant \cdots \leqslant C[n_1+n_2]$$

合并算法的做法是，每一步选出当前序列 A 和 B 中最小的数，并把它顺序放入序列 C 中，而找出这个最小的数只需比较两个序列的首项即可。假定我们已经找到（k–1）个最小的数并放入 $C[1]$至 $C[k–1]$中，而此时 $A[i]$ 和 $B[j]$分别为两个序列首项（算法开始时 $i=j=1$）。如果 $A[i] \leqslant B[j]$，那么 $A[i]$必定是所有还未被放入序列 C 的数中最小的数，所以把它放入 $C[k]$中，而序列 A 中下一个数 $A[i+1]$成为新的首项。同理，如果 $A[i] > B[j]$，那么 $B[j]$被放入 $C[k]$而序列 B 的下一个数 $B[j+1]$成为新的首项。当序列 A（或 B）中的数已全部放入序列 C 时，则只需把另一序列中所剩下的数依次放入 C 中即可。下面是该算法的伪码。

```
Merge (A[1.. n1], B[1.. n2], C[1.. n1+ n2])
1   i ← 1
2   j ← 1
3   k ← 1
4   while i ⩽ n1 and j ⩽ n2
5       if A[i] ⩽ B[j]
6           then C[k] ← A[i]
7                i ← i + 1
8           else C[k] ← B[j]
9                j ← j + 1
10      endif
11      k ← k + 1
12  endwhile
13  if i > n1
14      then C[k.. n1+ n2] ← B[j.. n2]
15      else C[k.. n1+ n2] ← A[i.. n1]
16  endif
17  End
```

如果我们把序列中数字的比较作为主要运算，那么任何情况下，合并算法需要最多 n_1+n_2-1 次比较。这是因为每一次比较后有一个数被放入序列 C 中，当序列 A（或 B）中的数已全部放入序列 C 时，另一序列中至少还有一个数未被选入 C。因为最多需要 n_1+n_2-1 次比较，合并算法的最坏情况复杂度为 $T(n)=O(n_1+n_2)$，这里 $n=n_1+n_2$ 是被合并的两个序列中所有数字的总和。

值得一提的是，上面合并算法的最好情况的复杂度仍然是 $T(n)=\Theta(n_1+n_2)$。这是因为，不论算法做了多少次比较，它都需要把序列 A 和 B 中每一个数字都放入 C 中。这个赋值运算有 n_1+n_2 次。实际上，把这个赋值运算作为主要运算似乎更为合理。但是，有一点要注意的是，如果所有三个数组 A、B、C 都是用链表连接，则算法中 14 和 15 行可以在 $O(1)$时间内完成。这时，最好情况的复杂度是$\Theta(\mathrm{Min}(n_1, n_2))$。

3.2.2 合并排序的算法及其复杂度

合并排序是一个分治算法。它的底是被排序列只含一个数，这时不需要任何操作，这个数本身即为有序的序列。否则，算法把序列一分为二，再递归地将这两个子序列排序，最后用合并算法将它们合并为一个递增序列。我们假定要排序的数字放在数组 $A[p..r]$中，$p \leqslant r$，下面是合并排序的伪码。

```
Mergesort (A[p..r])
1    if p < r
2        then mid ← ⌊(p + r)/2⌋
3             Mergesort(A[p.. mid])
4             Mergesort(A[mid +1 .. r])
5             Merge(A[p..mid], A[mid+1..r], C[p..r])
6             A[p..r] ← C[p..r])
7    endif
8    End
```

显然，前面所讲的合并算法 Merge ($A[1..n_1]$, $B[1..n_2]$, $C[1..n_1+n_2]$)需要稍加修改后才能用在排序算法之中。主要修改的地方是各数组两端的序号必须是变量，不能总是从 1 开始。这种修改极为容易，不在此赘述了。当我们需要将数组 $A[1..n]$中的 n 个数排序时，只需调用 Mergesort ($A[1..n]$)即可。

有些读者也许对递归算法不熟，这里举一个例子来帮助理解递归。假设我们对数组 $A[1..8]$ 进行合并排序，$A[1..8]=\{9, 6, 2, 4, 1, 5, 3, 8\}$。合并排序把它一分为二后做三件事：

1　合并排序 $A[1..4]$
2　合并排序 $A[5..8]$
3　$A[1..9]$ ← 合并 $A[1..4]$和 $A[5..8]$

当算法执行第 1 步时，它又递归地做三件事：

1.1　合并排序 $A[1..2]$
1.2　合并排序 $A[3..4]$
1.3　$A[1..4]$ ← 合并 $A[1..2]$和 $A[3..4]$

当算法执行第 1.1 步时，它又递归地做三件事：

1.1.1　合并排序 $A[1]$
1.1.2　合并排序 $A[2]$
1.1.3　$A[1..2]$ ← 合并 $A[1]$和 $A[2]$

当算法执行第 1.1.1 步和 1.1.2 步时，它遇到了底。合并 $A[1]$和 $A[2]$后得到 $A[1..2]=\{6, 9\}$，这也是第 1.1 步的结果。下面算法执行第 1.2 步。经过类似 1.1 步中三件事后，得到 $A[3..4]=\{2, 4\}$。这时算法执行第 1.3 步，得到 $A[1..4]=\{2, 4, 6, 9\}$。这也是第 1 步的结果。

完成第 1 步后，算法开始执行第 2 步。它的过程与第 1 步类似，结果得到 $A[5..8]=\{1, 3, 5, 8\}$。这时 $A[1..4]$ 和 $A[5..8]$ 都已完成排序，第 3 步将它们合并为 $A[1..8]=\{1, 2, 3, 4, 5, 6, 8, 9\}$。

从这个例子可以看出，合并排序是把一个序列不断地一分为二直到只含一个数为止，然后再不断地把较短的（排好的）序列合并为较长的序列直到全部排好。这个过程可以用一棵二叉树来表示。图 3-2 显示对应于上面例子的二叉树，其中树叶对应着底的情况。每个内结点 x 代表一个子序列。它的两个儿子结点代表这个子序列被一分为二后的两个更小的子序列。当这两个更小的子序列被排好序之后，它们在结点 x（父结点）处被合并为一个排好的序列。因此，树中每一结点既代表着一个划分操作又代表着一个合并操作。其划分的顺序是从根开始，自上而下，与树的前向遍历一致，而合并的顺序是从叶子（底）开始，自下而上，与树的后序遍历顺序一致。

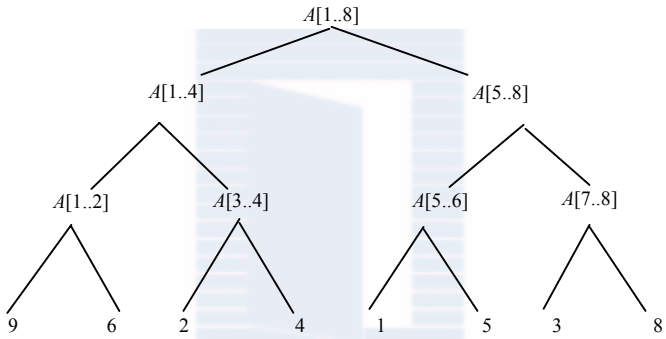


图 3-2　表示合并排序的一棵树

如果把被排序数字之间的比较作为主要的基本运算，合并排序在最坏情况下需要的比较次数 $T(n)$可以用下面的递推关系表示：

$$T(1)=0$$
$$T(n)=\mathrm{T}(\lceil n/2\rceil)+\mathrm{T}(\lfloor n/2\rfloor)+(n-1)$$

置 $n=2^k$ 后，有以下推导

$$\begin{aligned}T(n)&=T(2^k)=2T(2^{k-1})+2^k-1\\&=2[2T(2^{k-2})+2^{k-1}-1]+(2^k-1)\\&=2^2T(2^{k-2})+(2^k-2)+(2^k-1)\\&=\cdots\\&=2^kT(2^0)+(2^k-2^{k-1})+\cdots+(2^k-2)+(2^k-1)\\&=k2^k-(2^{k-1}+2^{k-2}+\cdots+2+1)\\&=k2^k-(2^k-1)\\&=n\lg n-(n-1)\end{aligned}\qquad(3.1)$$

更简便的方法是从主方法得到 $T(n)=2T(n/2)+(n-1)=\Theta(n\lg n)$。另外，因为排序中合并部分的最好情况需要至少 $\mathrm{Min}(n_1, n_2)=\lfloor n/2\rfloor$ 次比较，所以合并排序的递推关系满足不等式：$T(n)\geqslant 2T(n/2)+\lfloor n/2\rfloor$。因此，合并排序的最好情况的复杂度仍为$\Theta(n\lg n)$，从而其平均复杂度必定也是$\Theta(n\lg n)$。

3.2.3 合并排序的优缺点

不论是最坏情况还是平均情况，合并排序的复杂度是最好的。在第 4 章中，我们会证明这一点。作为另外一个优点，合并排序是个稳定排序。但是，它的一个重大缺点是，它不是一个就地操作的算法，需要使用$\Omega(n)$个存储单元（除数组 $A[1..n]$外的存储单元）。当 n 很大时，这是一个很大的开销。另外，当 n 很大时，递归所需要的堆栈也会增加开销。

3.3 堆排序

堆排序（heap sort）克服了合并排序不是就地操作的缺点，并且有相同的渐近复杂度$\Theta(n\lg n)$。堆排序的工作原理如下。它先把要排序的 n 个数字组织成一个称为堆的二叉树。这棵二叉树帮助我们很快找到最大（或最小）的一个数。当我们从堆里取走这个最大（或最小）数以后，我们可以方便地把剩下的二叉树再修复为一个堆。这样我们可以再取走下一个最大（或最小）数。堆排序就是通过这样不断从堆里取出最大（或最小）数和不断修复的过程完成排序的。下面我们先介绍堆的结构。

3.3.1 堆的数据结构

含有 n 个数字的堆（heap）是一棵有 n 个结点（包括叶结点）的二叉树，并满足以下 3 个条件：

1）所有叶结点必须出现在树的最底下一层或两层。

2）如果倒数第二层有叶结点，那么这层中的所有内结点必须出现在所有叶结点的左边。并且，除最后一个内结点可能只有一个子结点外，堆中每一个内结点必须要有两个子结点。

3）每个结点上存有一个数字并满足堆顺序（heap order），即任一内结点中的数要大于或等于其子树中每个结点中的数字。遵循这样顺序的堆称为最大堆。显然，我们也可以定义最小堆，其堆顺序则是父结点中的数字要小于或等于其子树中每个结点中的数字。为了方便和一致，本书用最大堆顺序。显然，在这样的堆中，根结点中的数最大。

图 3-3 给出了一个含有 10 个数字的堆的例子。可以证明（见习题），一个有 n 个结点的堆的高度为 $h=\lfloor \lg n \rfloor$。

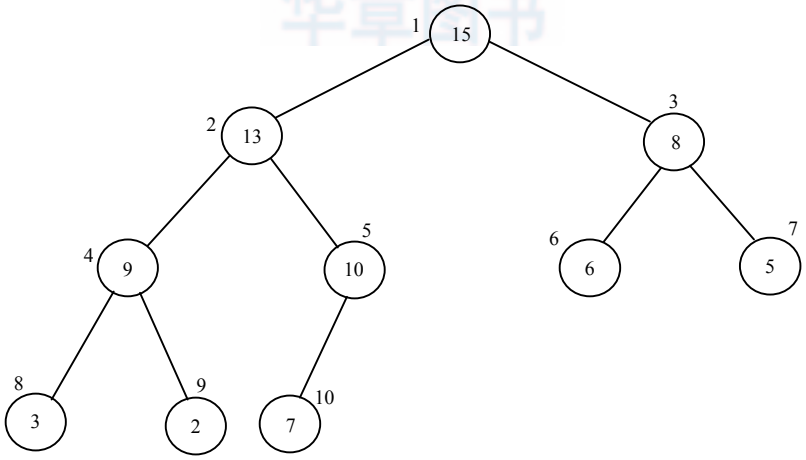


图 3-3 一个有 10 个数字的堆

有 n 个数字的堆通常可以用一个有 n 个单元的数组，比如 $A[1..n]$，来实现。具体办法是把堆中的数字从根开始，从上到下逐层顺序存入数组。对每一层中的数，采用从左到右的次序存入数组。以图 3-3 的堆为例，每个结点边上的序号标明了它在 $A[1..10]$中的位置。图 3-4 直接显示了

图 3-3 中 10 个数在 $A[1..10]$中存放的情况。这样存放后，这棵二叉树及其堆顺序隐含在这个数组中。这样的数据结构称为隐式数据结构。那么这个堆是怎样隐含在里面的呢？

A	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	15	13	8	9	10	6	5	3	2	7

图 3-4　用数组 $A[1..10]$实现图 3-3 中的堆

我们注意到，堆中任一个父结点的序号和它两个子结点的序号之间有一定的关系。假定 $A[i]$，$1 \leqslant i \leqslant n$，是堆中第 i 个结点。我们有以下关系：

1）$A[i]$的左儿子（left son）= $A[2i]$　　（如果 $2i \leqslant n$，否则 $A[i]$没有左儿子。）

2）$A[i]$的右儿子（right son）= $A[2i+1]$　　（如果 $2i+1 \leqslant n$，否则 $A[i]$没有右儿子。）

3）$A[i]$的父亲（father）= $A[\lfloor i/2 \rfloor]$　　（$i > 1$，否则 $A[i]$是根。）

以上这三个关系不难证明，留给读者。有了这三个关系，树中结点间的关系就完全确定了。

3.3.2　堆的修复算法及其复杂度

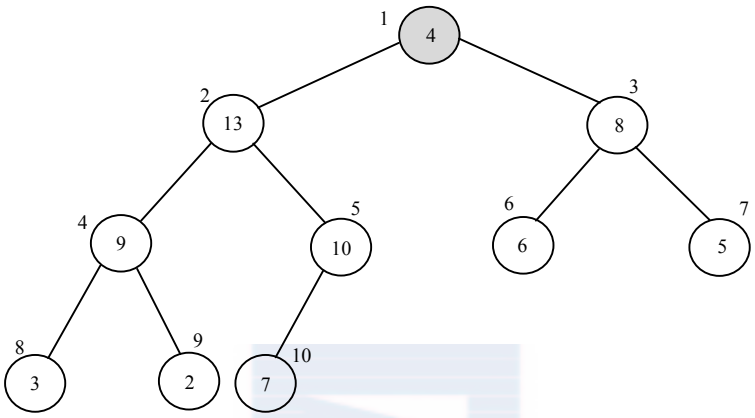
假设 $A[1..n]$中的 n 个数已形成一个堆，那么我们可以立即在 $A[1]$处找到最大的数。可是当我们把这个最大数从堆中取走后，如何去找第二个最大数呢？这时，$A[1]$中数不能再用，我们必须把第二个最大数移到 $A[1]$中，并使 $A[1..n-1]$成为余下的 $n-1$ 个数形成的堆。这就是把剩下的 $n-1$ 个数再修复成一个有 $n-1$ 个数的堆。这样，可以立即在树根处找到下一个最大的数，把它取走后再修复剩下的堆。重复这样的过程直至所有数都从堆中取走。这一节，我们解释如何修复一个堆。修复的第一步是把 $A[1]$和 $A[n]$中的数字对调，并把堆的规模减为 $n-1$。这样，余下的 $n-1$ 个数就在 $A[1..n-1]$中了。但是，$A[1..n-1]$还不是堆。唯一的问题是 $A[1]$中的数可能会小于它子结点中的数字。我们用下面的算法对 $A[1]$点进行修复。

下面的堆修复采用分治法。我们假设，在需要修复的堆中，某一个内结点 $A[i]$中的数小于其某个子结点中的数值，而其余的所有内结点都保持着最大堆顺序。算法把 $A[i]$中的数与其两子结点中的数进行比较。如果 $A[i]$的数字最大，则修复停止，否则 $A[i]$与持最大数的儿子进行交换，这样便修复了 $A[i]$与其儿子的关系。这样做没有影响到其他内结点的最大堆顺序，但与 $A[i]$交换数字的儿子除外，所以算法接着递归地对这个子结点进行修复。因为下一个需要修复的点都是当前修复点的儿子，递归迟早会停止。停止时，或者所有点都满足最大堆顺序，或者修复点是个叶结点而无需修复。下面是堆修复的伪码，其中变量 *heap-size* 是当前堆中数字的个数。

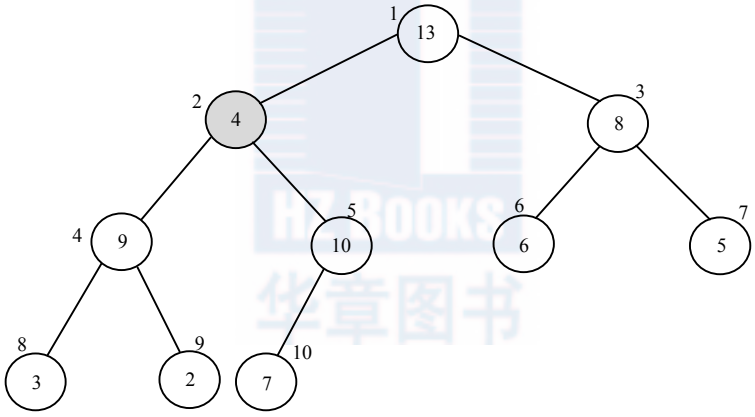
```
Max-Heapify(A[1.. heap-size], i)
1    l ← 2i          //左儿子的序号
2    r ← 2i+1        //右儿子的序号
3    if l ⩽ heap-size and A[l] > A[i]
4         then largest ← l
5         else largest ← i
6    endif
7    if r ⩽ heap-size and A[r] > A[largest]
8         then largest ← r
9    endif
10   if largest ≠ i
11        then      A[i] ↔A[largest]
```

```
12                  Max-Heapify([1.. heap-size], largest)
13    endif
14    End
```

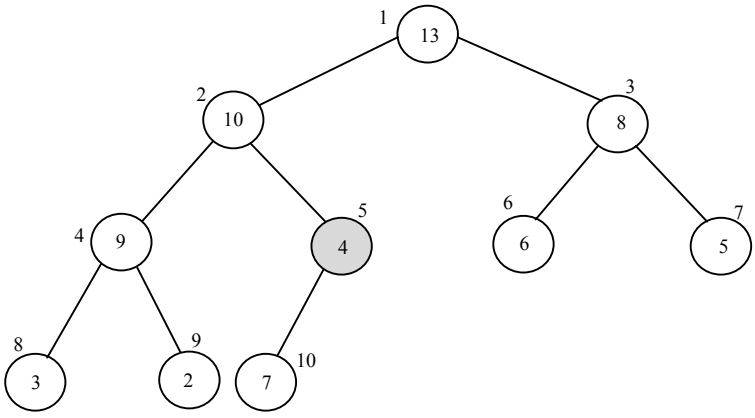
当我们需要修复根结点时，只需调用 Max-Heapify(*A*[1.. *heap-size*], 1)即可。图 3-5 给出一个堆被逐步修复的例子。



a）结点 *A*[1]需要修复

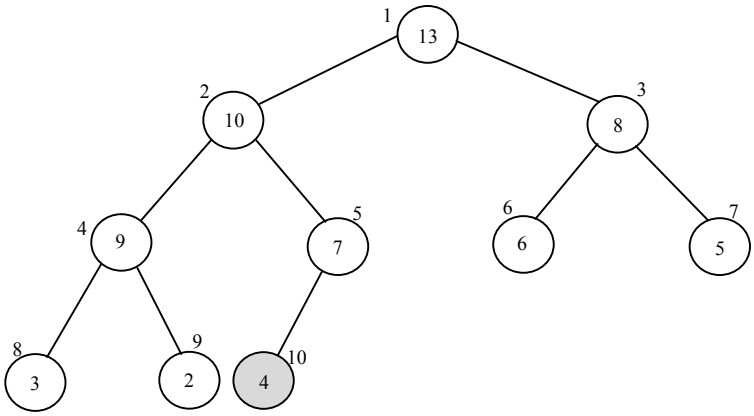


b）数字 4 和 13 交换后，结点 *A*[2]需要修复



c）数字 4 和 10 交换后，结点 *A*[5]需要修复

图 3-5　一个堆被逐步修复的例子



d）数字 4 和 7 交换后，没有结点需要修复

图 3-5 （续）

堆修复算法的复杂度可以这样分析：算法修复当前点时需要两次比较算出该点和它的两个子结点中，谁的数最大。然后，算法递归修复某子结点。因为堆的高度为 $h=\lfloor \lg n \rfloor$，所以递归的深度最多为$\lfloor \lg n \rfloor$。因此，最坏情况下，堆修复需要做 $2h=2\lfloor \lg n \rfloor = O(\lg n)$次比较。

值得注意的是，堆修复算法是个就地操作的算法，也就是说，它不需要或只需要常数个存储单元（除数组 A 本身以外）。

3.3.3 为输入数据建堆

假设我们要把存在数组 $A[1..n]$中的 n 个数用堆排序将其变为递增序列，那么我们首先要做一个准备工作，就是调动它们在数组中的位置使 $A[1..n]$成为一个堆。如图 3-4 所示，任何在 $A[1..n]$中的 n 个数已构成一棵二叉树，只是结点中的数字并不满足最大堆顺序。我们希望这个建堆的工作不要花太长时间，否则就会自找麻烦。下面是一个线性时间的分治法建堆算法。算法 Build-Max-Heap($A[1..n]$, i) 的作用是调整以 $A[i]$ 为根的子树里的数字使该子树满足最大堆顺序。

```
Build-Max-Heap (A[1..n], i)
1    l  ← 2i              //左儿子的序号
2    r  ← 2i + 1          //右儿子的序号
3    if  l ⩽ n
4        then Build-Max-Heap (A[1..n], l)
5    endif
6    if  r ⩽ n
7        then Build-Max-Heap (A[1..n], r)
8    endif
9    Max-Heapify (A[1..n], i)
10   End
```

这个分治法中的底没有明说，而是隐含其中。当 $A[i]$没有儿子时，算法不再继续。否则，算法先把左子树和右子树调整为堆以后调用 Max-Heapify($A[1..n]$, i) 将 $A[i]$为根的树变为堆。当我们调用 Build-Max-Heap ($A[1..n]$, 1) 后，即可将 $A[1..n]$变为一个堆。这个建堆算法显然也是就地操作的算法。它的复杂度 $T(n)$可分析如下。

置 $n=2^{h+1}-1$，这样 $A[1..n]$所对应的二叉树是一棵高度为 h 的满二叉树（complete binary tree），

这里 $h = \lg(n+1) - 1$。满二叉树的所有叶结点都在底层。图 3-6 给出了一个高为 3 的满二叉树的例子。一棵满二叉树的左右两棵子树是高度减 1 的满二叉树。

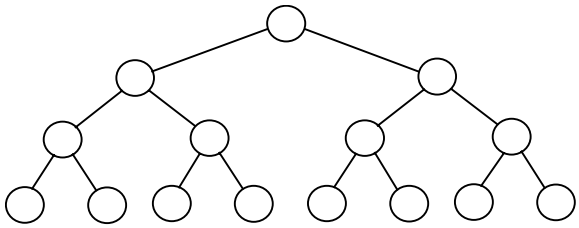


图 3-6 一棵满二叉树的例子

上面的建堆算法的复杂度满足递推关系：

$$T(n) = 2T\left(\frac{n-1}{2}\right) + O(\lg n)$$

这是由于左右两棵子树各有$(n-1)/2$ 个结点，而 Max-Heapify($A[1..n]$, i)需要最多 $O(\lg n)$次比较。由这个递推关系我们得到：

$$T(n) = 2T\left(\frac{n-1}{2}\right) + O(\lg n) \leqslant 2T(n/2) + O(\lg n) = O(n)$$

其中，最后一步可由主方法得到。

3.3.4 堆排序算法

将 $A[1..n]$建堆后，堆排序的过程就很简单了，其算法如下：

Heapsort ($A[1..n]$)
1 Build-Max-Heap ($A[1..n]$, 1)
2 *Heap-size* ← *n*
3 **while** *heap-size* > 1
4 $A[1] \leftrightarrow A[heap\text{-}size]$
5 *heap-size* ← *heap-size* − 1
6 Max-Heapify($A[1..heap\text{-}size]$, 1)
7 **endwhile**
8 **End**

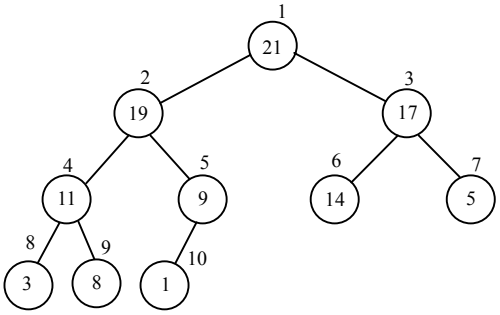
上面这个算法将 $A[1..n]$建堆以后进入一个循环。每次循环，算法做三件事：

1）把 $A[1]$和最后一个叶子 $A[heap\text{-}size]$中的数交换。这样当前堆中最大的数就放在正确位置上了。例如，循环第一步把最大数 $A[1]$放入 $A[n]$中。

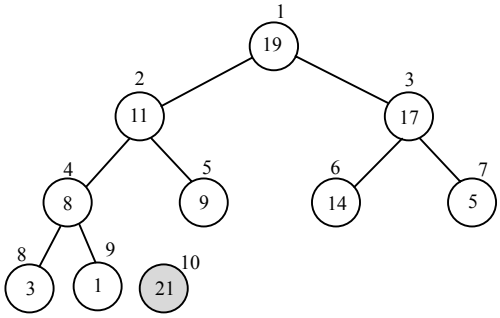
2）把堆的规模（*heap-size*）减 1，这等价于把最后一个叶子 $A[heap\text{-}size]$从树中摘去。原来在这个叶子中的数字已移到 $A[1]$中。

3）调用 Max-Heapify($A[1..heapsize]$, 1)进行堆修复。修复后，$A[1..heap\text{-}size]$变成一个堆，以便下一次循环中找到下一个最大数。

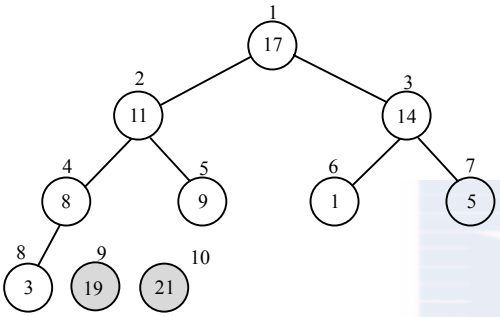
循环在 *heap-size* = 1 时，即堆中只有 $A[1]$一个数时结束。堆排序显然是就地操作的算法。图 3-7 给出了一个堆排序的例子。它显示了前四次循环后及循环结束时，数组 $A[1..10]$中哪些数字已排好序和哪些数字仍在堆中。为了清楚起见，用二叉树图示数组中数字在每次循环后的位置，但略去显示每一循环中对堆的修复过程。



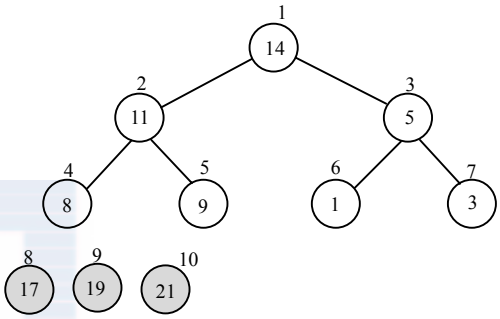
a）循环开始时的 $A[1..10]$是一个堆



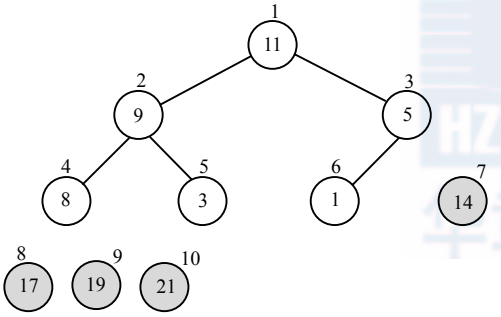
b）一次循环后，最大数 21 放在 $A[10]$，而 $A[1..9]$修复为一个堆



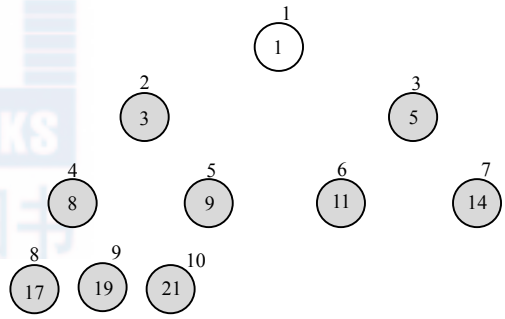
c）两次循环后，第二大数放入 $A[9]$，而 $A[1..8]$修复为一个堆



d）三次循环后，第三大数放入 $A[8]$，而 $A[1..7]$修复为一个堆



e）四次循环后，第四大数放入 $A[7]$，而 $A[1..6]$修复为一个堆



f）九次循环后，$A[1..n]$排好序，算法结束

图 3-7　一个堆排序的例子

3.3.5　堆排序算法的复杂度

堆排序需要先花 $O(n)$的时间把数组 $A[1..n]$变为一个堆。然后，算法循环（$n-1$）次，而每次循环的时间主要花在堆修复算法上。因为堆修复的时间是 $O(\lg n)$，所以堆排序算法的最坏情况复杂度是 $T(n) = O(n) + (n-1)O(\lg n) = O(n\lg n)$。那么它的最好情况和平均情况复杂度是多少呢？可以证明堆排序算法的最好情况的复杂度有下界$\Omega(n\lg n)$。从而，在任何情况下，堆排序复杂度都是 $T(n) = \Theta(n\lg n)$。这个证明将在下一章中讨论。

3.3.6　堆排序算法的优缺点

堆排序是就地操作的算法，它的复杂度 $T(n) = \Theta(n\lg n)$是渐近最优的，但是，堆排序的最大缺点是不稳定。读者可以很容易地找到不稳定的例子。另外，虽然它的复杂度与合并排序一样都是渐近最优的，但最坏情况下，它实际上需要比较的次数接近 $2n\lg n$，比合并排序的 $n\lg n-(n-1)$要多。这是因为在堆修复时，每层需要两次比较。

3.3.7 堆用作优先队列

堆的数据结构不仅可以用来排序，而且可以被许多其他算法用来动态地管理和修改数据，比如插入一个数据，删除一个数据，减少（或增加）一个数据的值，找到最大（或最小）的数等。能够有效提供这些操作的数据结构都称为优先队列（priority queue）。这里，有效指的是低复杂度。注意，这里的一个数据可能是一个含多个域的记录。比如，图书馆中的每本书由序号、书名、作者、出版社、出版时间等构成一个记录。其中每一项称为一个域，而用来查找的域称为关键字（key），比如序号常被用作一本图书的关键字找到了关键字也就找到了整个记录。所以管理和修改数据的算法往往也就是管理和修改关键字的算法。我们讲的数字就是指关键字。

数组是个最简单的优先队列。用数组可以在 $O(1)$时间内更改一个数，但要找最大数时却很慢，需要$\Theta(n)$时间。这里，我们看一下堆 $A[1..n]$是如何支持这些操作的，并讨论相应的复杂度。以下的算法一目了然而不需解释。

1．增加一个数的值

假设我们需要把 $A[i]$中的数增加到新的值 *key*，可用下面的算法。

```
Heap-Increase-Key(A, i, key)
1    if key < A[i]
2         then error      //要增加的值比原来值还小，不合理
3    endif
4    A[i] ← key
5    while i > 1 and A[father(i)] < A[i]
6         A[i] ↔ A[father (i)]
7         i ← father (i)
8    endwhile
9    End
```

这个算法复杂度为 $O(\lg n)$。

2．插入一个数

假设我们需要插入一个新的值 *key* 到堆里，可用下面的算法。

```
Max-Heap-Insert(A[1..n], key)
1    heap-size[A] ← n + 1
2    A[n+1] ← – ∞
3    Heap-Increase-Key(A, n+1, key)
4    End
```

这个算法复杂度为 $O(\lg n)$。

3．取走最大数

假设我们要取走最大的数并放入变量 *max* 中，可用下面的算法。

```
Heap-Extract-Max(A, max)
1    n ← heap-size[A]
2    if n < 1
3         then error "heap underflow"
4    endif
5    max ← A[1]
```

```
6    A[1] ← A[n]
7    heap-size[A] ← n – 1
8    Max-Heapify(A[1..heap-size], 1)
9    return max
10   End
```

这个算法复杂度为 $O(\lg n)$。

4．将最大的数复制到变量 *max* 中

假设我们把堆里最大的数放入变量 *max* 中，但不把它从堆中删除，而只是做一个复制，可用下面的算法。

```
Heap-Maximum(A[1..n], max)
1    max ← A[1]
2    return max
3    End
```

其他的基本操作，例如，减少一个数的值或删去一个数等，留为练习。

3.4 快排序

快排序（quick sort）是人们常用的又一个排序算法，它有很好的平均情况下的复杂度并且是一个就地操作的算法。下面进行详细介绍。

3.4.1 快排序算法

快排序是分治法的又一个例子。假定数组 $A[p..r]$，$p<r$，中数字需要排序，不同于合并排序，快排序不是把序列分为大小相等的两段，而是按照以下原则将序列分为两部分：第一部分放在 $A[p..q-1]$中$(p \leqslant q \leqslant r)$，而第二部分放在 $A[q+1..r]$中，使得第一部分中任何数字都小于或等于 $A[q]$，而 $A[q]$又小于第二部分的任何一个数。在 $A[q]$中的数称为中心点（pivot），不归入任一部分。这个中心点 $A[q]$的选取有多种方法，但结果都差不多，本书只介绍其中一种。有兴趣的读者可进一步查阅文献。

在把序列进行上述划分之后，快排序算法再递归地对第一部分和第二部分进行快排序。当第一、二部分排好序之后，整个序列就自然排好了。这个分治法的底是当序列为空或只有一个数字的情况。这时算法不需做任何事。所以，快排序实际上是个不断对序列进行划分的过程。

我们这里介绍的方法是用 $x=A[r]$做中心点来划分的。从 $A[p]$开始到 $A[r-1]$为止将每个数逐个与 $A[r]$比较。假设在某一步时，我们已比较了 $A[p]$到 $A[j-1]$中数字并且把它们分为两部分。其中小于或等于 $A[r]$的数字放在 $A[p..i]$中，而大于者放入 $A[i+1..j-1]$中。如 $i=p-1$，则表示第一部分是空集，而 $i=j-1$，则表示第二部分为空集。在 $A[j]$到 $A[r-1]$中的数是还未检查的数字。图 3-8 描述了这一情况。

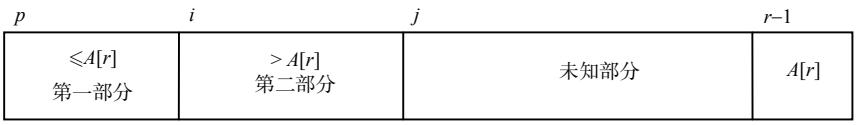


图 3-8　划分算法的图示

我们在划分算法中用了两个指针，i 和 j，其中 i 指向第一部分最右的位置，而 j 指向下一个需比较的数。开始时，$i=p-1$，$j=p$。每次我们比较 $A[j]$和 $A[r]$时，有两种结果，分述如下：

1）$A[j]>A[r]$。对这种情况，$A[j]$中数应属于第二部分。我们只需把指针 j 加 1 即可。

2）$A[j] \leqslant A[r]$。对这种情况，$A[j]$中数应属于第一部分。我们把 $A[j]$和 $A[i+1]$ 中数字交换后可把 $A[j]$中的数字并入第一部分。这时需把指针 i 和 j 分别加 1。

当 $A[1..r-1]$中每个数都和 $A[r]$比较后，交换 $A[i+1]$和 $A[r]$中数字。这样就把 $A[r]$中数放在了中心点位置上。显然，中心点位置是 $q = i+1$。

根据上面的讨论，划分算法只需要（$r-p$）次比较并且是就地操作，其伪码如下：

```
Partition(A[p .. r], q)
1    x ← A[r]
2    i ← p − 1
3    for j ← p to r −1
4        if A[j] ⩽ x            //如果 A[j] > x，for 循环会自动把 j 加 1
5            then    i ← i +1
6                    A[i] ↔ A[j]
7        endif
8    endfor
9    A[i + 1] ↔ A[r]
10   q ← i +1
11   return q
12   End
```

图 3-9 是一个划分的例子。它显示了每一次比较后数组中的变化，其中第二部分的数字用阴影表示。

初始状态								
i	*j* = *p*							*r*
	2	8	7	1	3	5	6	4

第一次比较后

	i	*j*						*r*
	2	8	7	1	3	5	6	4

第 2 次比较后

	i		*j*					*r*
	2	8	7	1	3	5	6	4

第 3 次比较后

	i			*j*				*r*
	2	8	7	1	3	5	6	4

第 4 次比较后

		i			*j*			*r*
	2	1	7	8	3	5	6	4

第 5 次比较后

			i			*j*		*r*
	2	1	3	8	7	5	6	4

第 6 次比较后

			i				*j*	*r*
	2	1	3	8	7	5	6	4

第 7 次比较后

			i				*j*	*r*
	2	1	3	8	7	5	6	4

把 $A[r]$放到中心点后

			i	*q*			*j*	*r*
	2	1	3	4	7	5	6	8

图 3-9　一个划分的例子

基于上面的划分算法，快排序算法可设计如下：

```
Quicksort(A[p..r])
1    if p < r
2        then Partition(A[p .. r], q)
3             Quicksort(A[p .. q−1])
4             Quicksort(A[q+1 .. r])
5    endif
6    End
```

如果要把 $A[1..n]$ 排序，只需调用 Quicksort($A[1..n]$)。

3.4.2 快排序算法最坏情况复杂度

快排序中主要操作仍然是数组中数字间的比较。直观上看，如果算法中每次划分都很不对称，则需要比较的次数最多。一个极端的情况是，$A[1..n]$是一个已排好序的序列，即 $A[1] \leqslant A[2] \leqslant A[3] \leqslant \cdots \leqslant A[n]$。容易看出，快排序所需要比较的次数为：$(n-1)+(n-2)+\cdots+1=n(n-1)/2$。所以，最坏情况的复杂度是 $T(n)=\Omega(n^2)$。

那么上面的情况是否是最坏呢？有没有更坏的情况呢？结论是没有更坏的情况了。

定理 3.1 给定序列 $A[p..r]$，快排序 Quicksort($A[p..r]$)需要最多 $n(n-1)/2$ 次比较，这里 n 为数组中元素的个数，即 $n=p-r+1$。

证明：我们用数学归纳法证明。当 $n \leqslant 2$ 时，定理显然正确。假设当 $0 \leqslant n \leqslant k$ 时（$k \geqslant 2$）定理正确。我们证明当 $n=k+1$ 时，定理也正确，即快排序需要最多 $k(k+1)/2$ 次比较。假定在快排序第一次划分后，第一部分有 a（$\geqslant 0$）个元素而第二部分有 b（$\geqslant 0$）个元素，$a+b=k$。这一划分需要 k 次比较。由归纳假设，用快排序对第一部分排序时最多需要 $a(a-1)/2$ 次比较，而对第二部分排序时最多需要 $b(b-1)/2$ 次比较。因此整个排序需要最多 $k+a(a-1)/2+b(b-1)/2$ 次比较。因为

$$
\begin{aligned}
& k+a(a-1)/2+b(b-1)/2 \\
=& \frac{1}{2}(a^2+b^2+2k-a-b) \\
=& \frac{1}{2}(a^2+b^2+k) \\
=& \frac{1}{2}[(a+b)^2-2ab+k] \\
=& \frac{1}{2}[k^2-2ab+k] \\
=& \frac{1}{2}(k^2+k)-ab \\
\leqslant& \frac{1}{2}k(k+1)
\end{aligned}
$$

所以定理成立。 ■

从定理 3.1 的证明看到快排序的最坏情况出现在 a 或 b 为零时，即 $A[1..n]$是一个已排好序的递增序列或递减序列，所以它的最坏情况复杂度是 $T(n)=\Theta(n^2)$。那么它的平均情况和最好情况的复杂度又是怎样的呢？下面两节证明这两个复杂度都是$\Theta(n\lg n)$。对证明不感兴趣的读者可跳过它们，或等有时间再看。

3.4.3 快排序算法平均情况复杂度

假设快排序对 n 个数进行划分后，第一部分含 k 个数，第二部分含（$n-k-1$）个数。整数 k 的值可以是 0 到（$n-1$）中任何一个数，一共有 n 种不同的情况。我们假定这 n 种情况的概率相等，均为 $1/n$。那么算法平均需要的比较次数 $T(n)$满足以下递推关系：

$$\begin{aligned} T(n) &= \frac{1}{n}\sum_{k=0}^{n-1}[T(k)+T(n-k-1)] + (n-1) \\ &= \frac{1}{n}\left[\sum_{k=0}^{n-1}T(k)+\sum_{k=0}^{n-1}T(n-k-1)\right] + (n-1) \\ &= \frac{2}{n}\sum_{k=0}^{n-1}T(k)+(n-1) \end{aligned} \tag{3.2}$$

由（3.2）式，我们得到

$$nT(n) = 2\sum_{k=0}^{n-1}T(k) + n(n-1) \tag{3.3}$$

把 n 换为（$n-1$）后可得

$$(n-1)T(n-1) = 2\sum_{k=0}^{n-2}T(k) + (n-1)(n-2) \tag{3.4}$$

由（3.3）减去（3.4）式得到

$$\begin{aligned} nT(n) - (n-1)T(n-1) &= 2T(n-1) + 2(n-1) \\ nT(n) &= (n+1)T(n-1) + 2n - 2 \end{aligned} \tag{3.5}$$

将（3.5）式两边同除 $n(n+1)$后得到

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1} - \frac{2}{n(n+1)} \tag{3.6}$$

将（3.6）式展开后得到

$$\begin{aligned} \frac{T(n)}{n+1} &< \frac{T(n-1)}{n} + \frac{2}{n+1} \\ &< \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &< \cdots \\ &< \frac{T(1)}{2} + \frac{2}{3} + \frac{2}{4} + \cdots + \frac{2}{n} + \frac{2}{n+1} \\ &< 2\left(\frac{1}{3}+\frac{1}{4}+\cdots+\frac{1}{n}+\frac{1}{n+1}\right) \\ &< 2\ln(n+1) \\ &= (2\ln 2)\lg(n+1) \\ &\approx 1.39\lg(n+1) \end{aligned} \tag{3.7}$$

所以，算法平均需要的比较次数和复杂度为

$$T(n) \approx 1.39(n+1)\lg(n+1) = \Theta(n\lg n)$$

3.4.4 快排序算法最好情况复杂度

直观地看，如果每次划分都把序列分为两个有相同大小的部分（或最多差一个数），则快排序所需要的比较次数最少。如果是这样，不妨设 $n = 2^{k+1}-1$，我们有如下递推关系：

$$T(n) = 2T\left(\frac{n-1}{2}\right) + n-1$$

或者

$$T(2^{k+1}-1) = 2T(2^k-1) + 2^{k+1} - 2$$

定义 $S(k) = T(2^{k+1}-1)$，得到等式

$$S(k) = 2S(k-1) + 2^{k+1} - 2 \qquad (3.8)$$

用序列求和法可将（3.8）式求解如下。

$$\begin{aligned}
T(n) &= S(k) = 2S(k-1) + 2^{k+1} - 2 \\
&= 2[2S(k-2) + 2^k - 2] + 2^{k+1} - 2 \\
&= 2^2 S(k-2) + 2^{k+1} - 2^2 + 2^{k+1} - 2 \\
&= \cdots \\
&= 2^k S(0) + k2^{k+1} - 2^k - 2^{k-1} - \cdots - 2 \\
&= k2^{k+1} - 2^{k+1} + 2 \\
&= (k-1)2^{k+1} + 2 \\
&= (n+1)[\lg(n+1) - 2] + 2 \\
&= (n+1)\lg(n+1) - 2n \\
&= \Theta(n\lg n)
\end{aligned}$$

那么直观的结果 $T(n) = (n+1)\lg(n+1) - 2n$ 是不是最好情况呢？有没有更好的情况呢？定理 3.2 回答了这个问题。

定理 3.2 给定序列 $A[p..r]$，快排序 Quicksort($A[p..r]$)需要最少$\lceil (n+1)\lg(n+1) \rceil - 2n$ 次比较，这里 n 为数组中元素的个数，即 $n = p-r+1$。

证明：我们用数学归纳法证明。当 $n \leqslant 3$ 时，可直接验证定理正确。假设当 $0 \leqslant n \leqslant k-1$ 时（$k \geqslant 4$）定理正确。我们证明当 $n = k$ 时，定理也正确，即快排序需要最少$\lceil (k+1)\lg(k+1) \rceil - 2k$ 次比较。假定在快排序第一次划分后，第一部分有 a（$\geqslant 0$）个元素而第二部分有 b（$\geqslant 0$）个元素，$a + b = k-1$。这一划分需要 $k-1$ 次比较。由归纳假设，用快排序对第一部分排序时最少需要$\lceil (a+1)\lg(a+1) \rceil - 2a$ 次比较，而对第二部分排序时最少需要$\lceil (b+1)\lg(b+1) \rceil - 2b$ 次比较，因此整个排序需要最少 $f(a, b) = k-1 + \lceil (a+1)\lg(a+1) \rceil - 2a + \lceil (b+1)\lg(b+1) \rceil - 2b$ 次比较。下面我们来简化这个式子。

$$\begin{aligned}
f(a, b) &= k-1 + \lceil (a+1)\lg(a+1) \rceil - 2a + \lceil (b+1)\lg(b+1) \rceil - 2b \\
&= \lceil (a+1)\lg(a+1) \rceil + \lceil (b+1)\lg(b+1) \rceil - 2(a+b) + (k-1) \\
&= \lceil (a+1)\lg(a+1) \rceil + \lceil (b+1)\lg(b+1) \rceil - (k-1) \qquad (\text{因为 } a+b = k-1) \\
&\geqslant (a+1)\lg(a+1) + (b+1)\lg(b+1) - (k-1)
\end{aligned}$$

因为$(a+1)\lg(a+1) + (b+1)\lg(b+1)$在 $a = b = (k-1)/2$ 时有极小值，所以

$$\begin{aligned}
f(a, b) &\geqslant (a+1)\lg(a+1) + (b+1)\lg(b+1) - (k-1) \\
&\geqslant (k+1)\lg\frac{k+1}{2} - (k-1) \\
&= (k+1)[\lg(k+1) - 1] - (k-1) \\
&= (k+1)\lg(k+1) - 2k
\end{aligned}$$

因为比较次数 $f(a, b)$必须是整数，所以有

$$f(a, b) \geqslant \lceil (k+1)\lg(k+1) \rceil - 2k$$

归纳成功。 ■

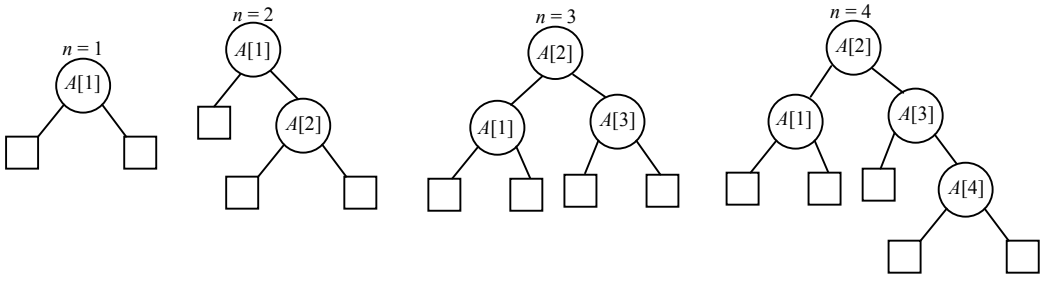
由定理 3.2 可知，快排序的最好情况复杂度是$\Theta(n\lg n)$。在第 4 章的习题 5 中，我们把快排序的运算过程用一棵二叉树来描述，从而给出最好情况复杂度是$\Theta(n\lg n)$的另一个证明。

3.4.5 快排序算法优缺点

如上面所分析，快排序是一个就地操作的算法，并且它的平均复杂度是渐近最优的，即$\Theta(n\lg n)$。它实际使用的比较次数平均为 $1.39n\lg n$，应该优于堆排序所需次数，但它在最坏情况有复杂度$\Theta(n^2)$，高于堆排序。另外，它不是稳定的排序。

习题

1. 给出一个例子，证明在最坏情况下，合并算法至少需要 n_1+n_2-1 次比较。
2. （a）设计一个复杂度为 $O(n\lg n)$的算法，以确定数组 $A[1..n]$中的 n 个数是否有相同的数字。
   （b）设计一个复杂度为 $O(n\lg n)$的算法，把数组 $A[1..n]$中出现奇数次的数字挑选出来。
3. （a）一个高为 h 的堆最少和最多能含有多少个结点（包括所有内结点和叶结点）？
   （b）证明一个含有 n 个数的堆的高为$\lfloor \lg n \rfloor$。
4. 假设 Heap-Delete ($A[1..n]$, i)表示将 $A[i]$这个数从数组 $A[1..n]$构成的堆中删去并使所余 $n-1$ 个数形成一个堆的操作。用伪码设计一个复杂度为 $O(\lg n)$ 的算法来实现 Heap-Delete ($A[1..n]$, i)。
5. 假设 Heap-Decrease-Key(A, i, key) 表示在数组 $A[1..n]$构成的堆中把 $A[i]$的值减少为 key 并把 $A[1..n]$修复为一个堆。用伪码设计一个复杂度为 $O(\lg n)$ 的算法来实现 Heap-Decrease-Key(A, i, key)。
6. 给定一个排好序的数组 $A[1] \leqslant A[2] \leqslant \cdots \leqslant A[n]$，例 2-1 中的二元搜索算法可以用一棵二元搜索树来描述。树中每个内结点含有一个数组中的数。树根里的数是算法进行比较的第一个数，即 $A[midpoint]$。根据结果是 $x < A[midpoint]$ 还是 $x > A[midpoint]$，算法决定是递归搜索左半部分，还是递归搜索右半部分。因而这棵二元搜索树的左右两棵子树可相应地递归构造。下图给出了当 n =1，2，3，4 时二元搜索树的例子。其中叶结点表示搜索失败的情况。



   （a）证明一棵二元搜索树 T 的叶结点只出现在最底下两层。
   （b）证明一棵含 n 个数的二元搜索树的高度为 $h = \lceil \log(n+1) \rceil$。
*7. 一个结点在一棵树中的高度就是以这个结点为根的子树的高度。证明在一个有 n 个数字的堆中，高度为 h 的结点数最多为$\left\lceil \frac{n}{2^{h+1}} \right\rceil$。
8. （**K 路合并问题**）设计一个 $O(n\lg k)$ 时间算法，将 k 个排好序的序列合并为单一排好序的序列。这里 n 是所有 k 个序列中数字的总数。
9. 我们知道数组 $A[1..n]$形成的堆里，第 i 个数 $A[i]$， $1 \leqslant i \leqslant n$，的左儿子、右儿子及父亲的所在

位置可以由下面公式算出：

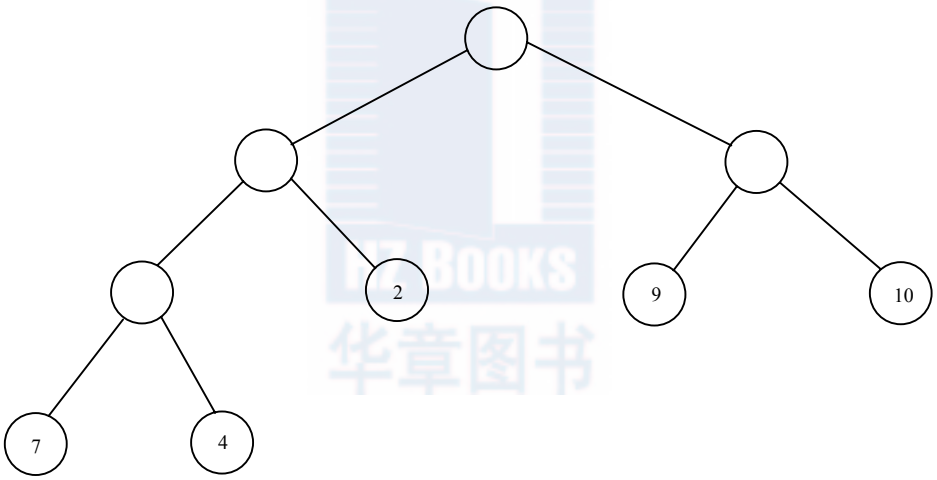
$$\text{Left}(A[i]) = A[2i]$$
$$\text{Right}(A[i]) = A[2i+1]$$
$$\text{Father}(A[i]) = A[\lfloor i/2 \rfloor]$$

但是很多时候我们不能把这个堆存放在从 $A[1]$开始的数组中，而是存放在从 $A[p]$开始的 n 个单元中，即存放在$A[p..r]$中，这里$r = p + n - 1$。这相当于把这n个数在数组中向右平移了（$p-1$）个位置。请给出在这种情况下，确定数字 $A[i]$（$p \leqslant i \leqslant r$），的左儿子、右儿子及父亲的所在位置的公式。

10．证明一个有 n 个数字的堆的左子树最多含有$\lfloor 2n/3 \rfloor$个结点。

11．假设给定一个 n 个数的数组 $A[1..n]$和一个常数 x。我们希望确定数组中是否存在两个数，$A[i]$和 $A[j]$（$1 \leqslant i < j \leqslant n$），使得 $A[i]+A[j] = x$。设计一个复杂度为 $O(n\lg n)$ 的算法解决这个问题。如果这样两个数存在，则报告这两个数，否则报告不存在。

12．锦标赛排序法是一个基于比较的排序算法。它可以用一棵称为锦标赛树的完全二叉树来描述。这棵二叉树要求正好有 n 个叶子来存储 n 个要排序的数字，并且所有叶子在底层或倒数第二层。下面是一个有 5 个叶子的一个锦标赛树图例。



算法开始前，将要排序的 n 个数字放在这 n 个叶子中。每个内结点代表一次比较。每次比较中胜者，即较小的数，参加下一轮在其父结点处的比较。在每个内结点处，当它的两个子结点处的比较有了结果之后，该结点处的比较即可进行。最后，在根结点处的比较决出冠军，即最小的数。因为一共有（$n-1$）个内结点，所以只需（$n-1$）次比较就可以找到最小数。当确定了最小的数后，即可把它送到输出序列中。另外，把它原来所在的叶子中的值改为 ∞。显然，重复上面的过程可得到下一个最小的数。

（a）如果重复所有在内结点处的比较去找下一个最小的数，我们又需要（$n-1$）次比较。这个复杂度太高。请设计一个只需 $O(\lg n)$次比较的算法去找下一个最小的数。（只需解释步骤，不要求伪码。）

（b）请用伪码设计一个用数组来实现锦标赛排序的算法，使其复杂度为 $O(n\lg n)$。

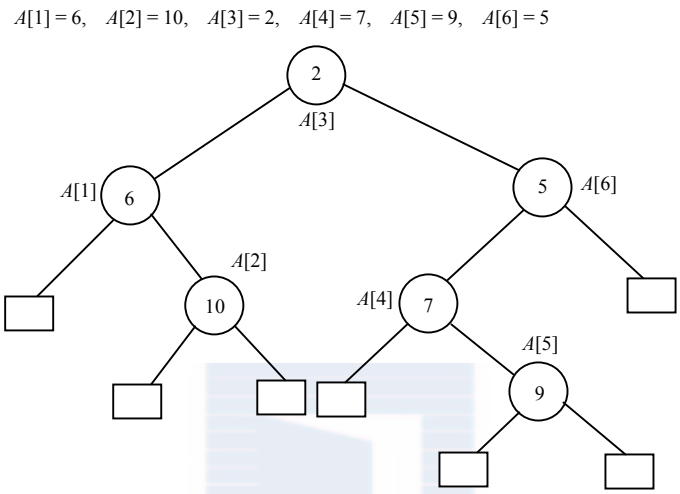
13．给定一个数组 $A[1..n]$，我们希望建一个有 n 个内结点的完全二叉树 T，它满足以下条件：

1）T 的根中存有数组 $A[1..n]$中最小的数。

2）假设根中的数为 $A[r]$，则根的左子树由数组 $A[1..r-1]$中的数递归建立，而根的右子树由数组 $A[r+1..n]$中的数递归建立。

3）当数组为空时，对应的子树为叶结点而过程停止。

我们称这样的二叉树为**最小优先树**。下面图示给出一个例子。



（a）画出对应于下面序列的最小优先树：6，5，2，9，7，1，3，10，9。

（b）设计一个构造最小优先树的算法，其平均复杂度为 $O(n\lg n)$。我们假设最小数出现在序列中任何位置的概率相同。（注：可以有 $O(n)$算法。）

14. 如图所示，平面上有上、下两条平行线。每条线上各自分布有 n 个点，并从左到右顺序标为1，2，…，n。我们把上面的 n 个点与下面的 n 个点配成一一对应的 n 个对子后，用一个直线段把每对点连上。若上面的点 i 与下面的点 k 相连，则记为 $k=\pi(i)$（$1 \leqslant i \leqslant n$）。如果 $i<j$ 但 $\pi(i)>\pi(j)$，那么线段$(i, \pi(i))$和线段$(j, \pi(j))$会有交叉点。例如，在下图中一共有 10 个交叉点。请设计一个 $O(n\lg n)$的算法计算给定的 n 对连线$(i, \pi(i))$（$1 \leqslant i \leqslant n$）一共有多少个交叉点。