快速排序

对于包含 n 个数的输入数组来说，快速排序是一种最坏情况时间复杂度为 $\Theta(n^2)$ 的排序算法。虽然最坏情况时间复杂度很差，但是快速排序通常是实际排序应用中最好的选择，因为它的平均性能非常好：它的期望时间复杂度是 $\Theta(n\lg n)$，而且 $\Theta(n\lg n)$ 中隐含的常数因子非常小。另外，它还能够进行原址排序(见 2.1 节)，甚至在虚存环境中也能很好地工作。

7.1 节将描述快速排序算法及它的一个重要的划分子程序。因为快速排序的运行情况比较复杂，在 7.2 节中，我们先对其性能进行一个直观的讨论，在本章的最后会给出一个准确的分析。在 7.3 节中，我们会介绍一个基于随机抽样的快速排序算法。这一算法的期望时间复杂度较好，而且没有什么特殊的输入会导致最坏情况的发生。7.4 节对这一随机算法的分析表明，其最坏情况时间复杂度是 $\Theta(n^2)$；在元素互异的情况下，期望时间复杂度 $O(n\lg n)$。

7.1 快速排序的描述

与归并排序一样，快速排序也使用了 2.3.1 节介绍的分治思想。下面是对一个典型的子数组 $A[p..r]$ 进行快速排序的三步分治过程：

分解：数组 $A[p..r]$ 被划分为两个(可能为空)子数组 $A[p..q-1]$ 和 $A[q+1..r]$，使得 $A[p..q-1]$ 中的每一个元素都小于等于 $A[q]$，而 $A[q]$ 也小于等于 $A[q+1..r]$ 中的每个元素。其中，计算下标 q 也是划分过程的一部分。

解决：通过递归调用快速排序，对子数组 $A[p..q-1]$ 和 $A[q+1..r]$ 进行排序。 170

合并：因为子数组都是原址排序的，所以不需要合并操作：数组 $A[p..r]$ 已经有序。

下面的程序实现快速排序：

```
QUICKSORT(A, p, r)
1  if  p < r
2       q = PARTITION(A, p, r)
3       QUICKSORT(A, p, q−1)
4       QUICKSORT(A, q+ 1, r)
```

为了排序一个数组 A 的全部元素，初始调用是 QUICKSORT(A，1，$A.length$)。

数组的划分

算法的关键部分是 PARTITION 过程，它实现了对子数组 $A[p..r]$ 的原址重排。

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p−1
3  for j = p to r−1
4       if A[j]≤x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i+ 1] with A[r]
8  return i + 1
```

图 7-1 显示了 PARTITION 如何在一个包含 8 个元素的数组上进行操作的过程。PARTITION 总是选择一个 $x=A[r]$ 作为**主元**(pivot element)，并围绕它来划分子数组 $A[p..r]$。

随着程序的执行，数组被划分成4个(可能有空的)区域。在第3～6行的**for**循环的每一轮迭代的开始，每一个区域都满足一定的性质，如图7-2所示。我们将这些性质作为循环不变量：

在第3～6行循环体的每一轮迭代开始时，对于任意数组下标k，有：

1. 若$p \leqslant k \leqslant i$，则$A[k] \leqslant x$。
2. 若$i+1 \leqslant k \leqslant j-1$，则$A[k] > x$。
3. 若$k=r$，则$A[k]=x$。

171

但是上述三种情况没有覆盖下标j到$r-1$，对应位置的值与主元之间也不存在特定的大小关系。

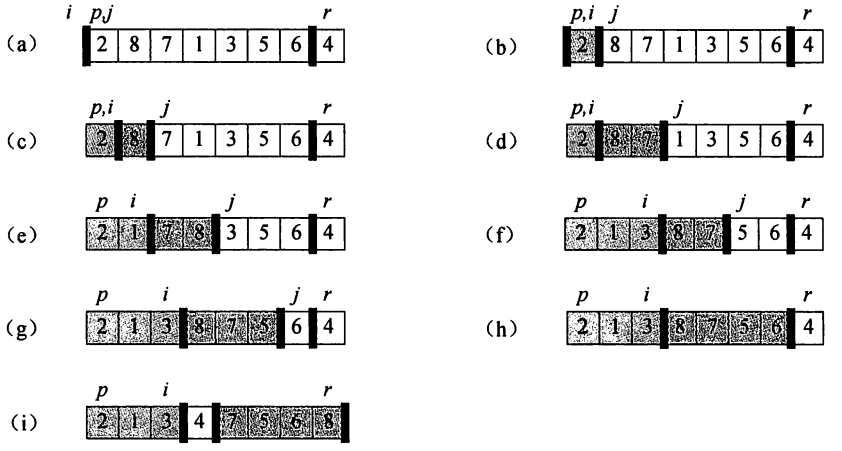

图7-1 在一个样例数组上的PARTITION操作过程。数组项$A[r]$是主元x。浅阴影部分的数组元素都在划分的第一部分，其值都不大于x。深阴影部分的元素都在划分的第二部分，其值都大于x。无阴影的元素则是还未分入这两个部分中的任意一个。最后的白色元素就是主元x。(a)初始的数组和变量设置。数组元素均未被放入前两个部分中的任何一个。(b)2与它自身进行交换，并被放入了元素值较小的那个部分。(c)～(d)8和7被添加到元素值较大的那个部分中。(e)1和8进行交换，数值较小的部分规模增加。(f)数值3和7进行交换，数值较小的部分规模增加。(g)～(h)5和6被包含进较大部分，循环结束。(i)在第7～8行中，主元被交换，这样主元就位于两个部分之间

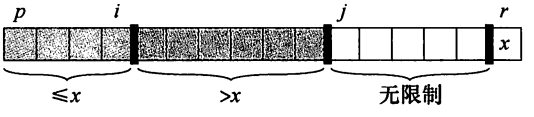

图7-2 在子数组$A[p..r]$上，PARTITION维护了4个区域。$A[p..i]$区间内的所有值都小于等于x，$A[i+1..j-1]$区间内的所有值都大于x，$A[r]=x$。子数组$A[j..r-1]$中的值可能属于任何一种情况

我们需要证明这个循环不变量在第一轮迭代之前是成立的，并且在每一轮迭代后仍然都成立。在循环结束时，该循环不变量还可以为证明正确性提供有用的性质。

172

初始化：在循环的第一轮迭代开始之前，$i=p-1$和$j=p$。因为在p和i之间、$i+1$和$j-1$之间都不存在值，所以循环不变量的前两个条件显然都满足。第1行中的赋值操作满足了第三个条件。

保持：如图7-3所示，根据第4行中条件判断的不同结果，我们需要考虑两种情况。图7-3(a)显示当$A[j] > x$时的情况：循环体的唯一操作是j的值加1。在j值增加后，对$A[j-1]$，条件2成立，且所有其他项都保持不变。图7-3(b)显示当$A[j] \leqslant x$时的情况：将i值加1，交换$A[i]$和$A[j]$，再将j值加1。因为进行了交换，现在有$A[i] \leqslant x$，所以条件1得到满足。类似地，我们也能得到$A[j-1] > x$。因为根据循环不变量，被交换进$A[j-1]$的值总是大于x的。

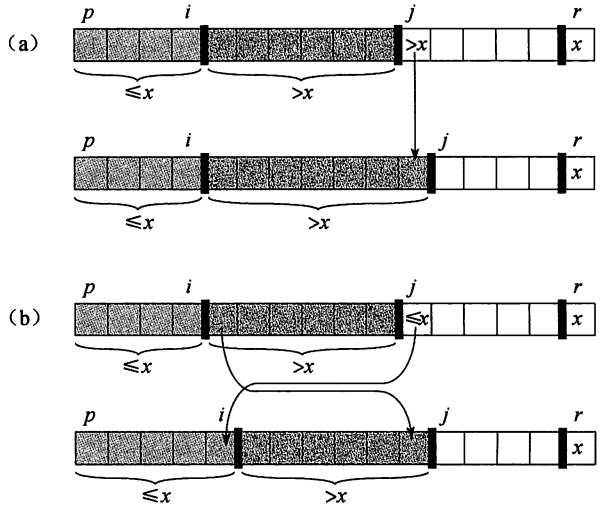

图 7-3 PARTITION 的一次迭代中会有两种可能的情况：(a)如果 $A[j]>x$，需要做的只是将 j 的值加 1，从而使循环不变量继续保持。(b)如果 $A[j]\leqslant x$，则将下标 i 的值加 1，并交换 $A[i]$和 $A[j]$，再将 j 的值加 1。此时，循环不变量同样得到保持

终止：当终止时，$j=r$。于是，数组中的每个元素都必然属于循环不变量所描述的三个集合的一个，也就是说，我们已经将数组中的所有元素划分成了三个集合：包含了所有小于等于 x 的元素的集合、包含了所有大于 x 的元素的集合和只有一个元素 x 的集合。

在 PARTITION 的最后两行中，通过将主元与最左的大于 x 的元素进行交换，就可以将主元移到它在数组中的正确位置上，并返回主元的新下标。此时，PARTITION 的输出满足划分步骤规定的条件。实际上，一个更严格的条件也可以得到满足：在执行完 QUICKSORT 的第 2 行之后，$A[q]$严格小于 $A[q+1..r]$内的每一个元素。

PARTITION 在子数组 $A[p..r]$上的时间复杂度是 $\Theta(n)$，其中 $n=r-p+1$(见练习 7.1-3)。

练习

7.1-1 参照图 7-1 的方法，说明 PARTITION 在数组 $A=\langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11\rangle$上的操作过程。

7.1-2 当数组 $A[p..r]$中的元素都相同时，PARTITION 返回的 q 值是什么？修改 PARTITION，使得当数组 $A[p..r]$中所有元素的值都相同时，$q=\lfloor (p+r)/2\rfloor$。

7.1-3 请简要地证明：在规模为 n 的子数组上，PARTITION 的时间复杂度为 $\Theta(n)$。

7.1-4 如何修改 QUICKSORT，使得它能够以非递增序进行排序？

7.2 快速排序的性能

173 ~ 174

快速排序的运行时间依赖于划分是否平衡，而平衡与否又依赖于用于划分的元素。如果划分是平衡的，那么快速排序算法性能与归并排序一样。如果划分是不平衡的，那么快速排序的性能就接近于插入排序了。在本节中，我们将给出划分为平衡或不平衡时快速排序性能的非形式化的分析。

最坏情况划分

当划分产生的两个子问题分别包含了 $n-1$ 个元素和 0 个元素时，快速排序的最坏情况发生了(证明见 7.4.1 节)。不妨假设算法的每一次递归调用中都出现了这种不平衡划分。划分操作的时间复杂度是 $\Theta(n)$。由于对一个大小为 0 的数组进行递归调用会直接返回，因此 $T(0)=\Theta(1)$，

于是算法运行时间的递归式可以表示为：

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$$

从直观上来看，每一层递归的代价可以被累加起来，从而得到一个算术级数(公式(A.2))，其结果为 $\Theta(n^2)$。实际上，利用代入法可以直接得到递归式 $T(n)=T(n-1)+\Theta(n)$ 的解为 $T(n)=\Theta(n^2)$(见练习 7.2-1)。

因此，如果在算法的每一层递归上，划分都是最大程度不平衡的，那么算法的时间复杂度就是 $\Theta(n^2)$。也就是说，在最坏情况下，快速排序算法的运行时间并不比插入排序更好。此外，当输入数组已经完全有序时，快速排序的时间复杂度仍然为 $\Theta(n^2)$。而在同样情况下，插入排序的时间复杂度为 $O(n)$。

最好情况划分

在可能的最平衡的划分中，PARTITION 得到的两个子问题的规模都不大于 $n/2$。这是因为其中一个子问题的规模为 $\lfloor n/2 \rfloor$，而另一个子问题的规模为 $\lceil n/2 \rceil - 1$。在这种情况下，快速排序的性能非常好。此时，算法运行时间的递归式为：

$$T(n) = 2T(n/2) + \Theta(n)$$

在上式中，我们忽略了一些余项以及减 1 操作的影响。根据主定理(定理 4.1)的情况 2，上述递归式的解为 $T(n)=\Theta(n\lg n)$。通过在每一层递归中都平衡划分子数组，我们得到了一个渐近时间上更快的算法。

平衡的划分

快速排序的平均运行时间更接近于其最好情况，而非最坏情况。详细的分析可以参看本书 7.4 节。理解这一点的关键就是理解划分的平衡性是如何反映到描述运行时间的递归式上的。

175

例如，假设划分算法总是产生 9∶1 的划分，乍一看，这种划分是很不平衡的。此时，我们得到的快速排序时间复杂度的递归式为：

$$T(n) = T(9n/10) + T(n/10) + cn$$

这里，我们显式地写出了 $\Theta(n)$ 项中所隐含的常数 c。图 7-4 显示了这一递归调用所对应的递归树。注意，树中每一层的代价都是 cn，直到在深度 $\log_{10} n = \Theta(\lg n)$ 处达到递归的边界条件时为

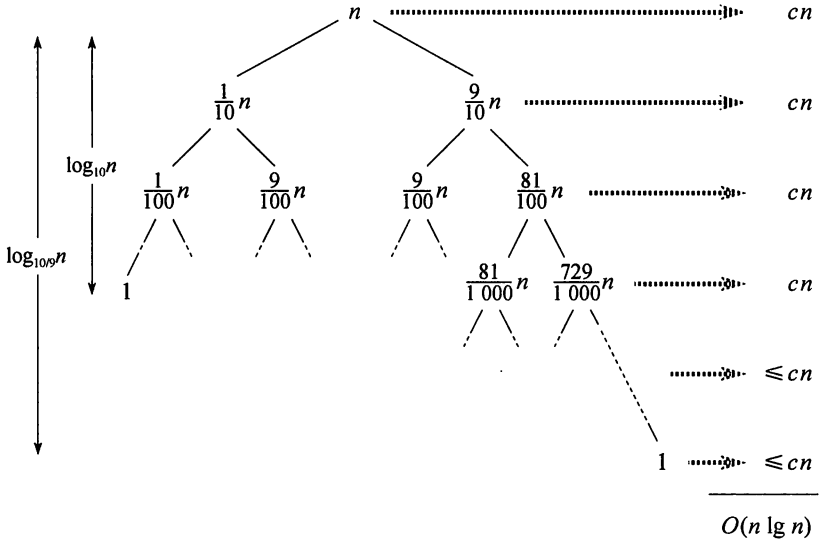

图 7-4 QUICKSORT 的一棵递归树，其中 PARTITION 总是产生 9∶1 的划分。该树的时间复杂度为 $O(n\lg n)$。每个结点的值表示子问题的规模，每一层的代价显示在最右边。每一层的代价包含了 $\Theta(n)$ 项中隐含的常数 c

止，之后每层代价至多为 cn。递归在深度为 $\log_{10/9} n = \Theta(\lg n)$ 处终止。因此，快速排序的总代价为 $O(n \lg n)$。因此，即使在递归的每一层上都是 9∶1 的划分，直观上看起来非常不平衡，但快速排序的运行时间是 $O(n \lg n)$，与恰好在中间划分的渐近运行时间是一样的。实际上，即使是 99∶1的划分，其时间复杂度仍然是 $O(n \lg n)$。事实上，任何一种常数比例的划分都会产生深度为 $\Theta(\lg n)$的递归树，其中每一层的时间代价都是 $O(n)$。因此，只要划分是常数比例的，算法的运行时间总是 $O(n \lg n)$。 176

对于平均情况的直观观察

为了对快速排序的各种随机情况有一个清楚的认识，我们需要对遇到各种输入的出现频率做出假设。快速排序的行为依赖于输入数组中元素的值的相对顺序，而不是某些特定值本身。与 5.2 节中对雇用问题所做的概率分析类似，这里我们也假设输入数据的所有排列都是等概率的。

当对一个随机输入的数组运行快速排序时，想要像前面非形式化分析中所假设的那样，在每一层上都有同样的划分是不太可能的。我们预期某些划分会比较平衡，而另一些则会很不平衡。例如，在练习 7.2-6 中，会要求读者说明 PARTITION 所产生的划分中 80%以上都比 9∶1 更平衡，而另 20%的划分则比 9∶1 更不平衡。

在平均情况下，PARTITION 所产生的划分同时混合有“好”和“差”的划分。此时，在与 PARTITION 平均情况执行过程所对应的递归树中，好和差的划分是随机分布的。基于直觉，假设好和差的划分交替出现在树的各层上，并且好的划分是最好情况划分，而差的划分是最坏情况划分，图 7-5(a)显示出了递归树的连续两层上的划分情况。在根结点处，划分的代价为 n，划分产生的两个子数组的大小为 $n-1$ 和 0，即最坏情况。在下一层上，大小为 $n-1$ 的子数组按最好情况划分成大小分别为 $(n-1)/2-1$ 和 $(n-1)/2$ 的子数组。在这里，我们假设大小为 0 的子数组的边界条件代价为 1。

在一个差的划分后面接着一个好的划分，这种组合产生出三个子数组，大小分别为 0、$(n-1)/2-1$ 和 $(n-1)/2$。这一组合的划分代价为 $\Theta(n)+\Theta(n-1)=\Theta(n)$。该代价并不比图 7-5(b)中的更差。在图 7-5(b)中，一层划分就产生出大小为 $(n-1)/2$ 的两个子数组，划分代价为 $\Theta(n)$。但是，后者的划分是平衡的！从直观上看，差划分的代价 $\Theta(n-1)$可以被吸收到好划分的代价$\Theta(n)$中去，而得到的划分结果也是好的。因此，当好和差的划分交替出现时，快速排序的时间复杂度与全是好的划分时一样，仍然是 $O(n \lg n)$。区别只是 O 符号中隐含的常数因子要略大一些。在 7.4.2 节中，我们将给出一个关于随机输入情况下快速排序的期望时间复杂度的更严格的分析。

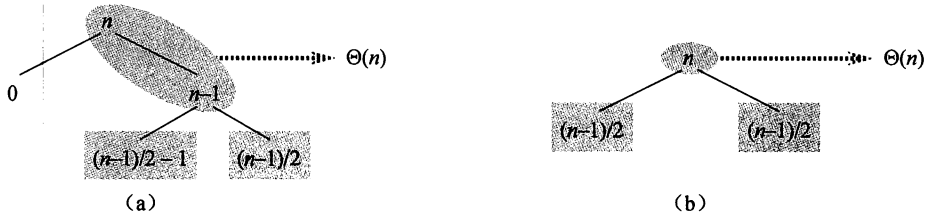

图 7-5 (a)一棵快速排序递归树的两层。在根结点这一层的划分代价是 n，产生了一个“坏”的划分：两个子数组的大小分别为 0 和 $n-1$。对大小为 $n-1$ 的子数组的划分代价为 $n-1$，并产生了一个“好”的划分：大小分别为 $(n-1)/2-1$ 和 $(n-1)/2$ 的子数组。(b)一棵非常平衡的递归树中的一层。在两棵树中，椭圆阴影所示的子问题的划分代价都是 $\Theta(n)$。可以看出，(a)中以矩形阴影显示的待解决子问题的规模并不大于(b)中对应的待解决子问题

练习

7.2-1 利用代入法证明：正如 7.2 节开头提到的那样，递归式 $T(n)=T(n-1)+\Theta(n)$的解为 $T(n)=\Theta(n^2)$。

7.2-2 当数组 A 的所有元素都具有相同值时，QUICKSORT 的时间复杂度是什么？

7.2-3 证明：当数组 A 包含的元素不同，并且是按降序排列的时候，QUICKSORT 的时间复杂度为 $\Theta(n^2)$。

7.2-4 银行一般会按照交易时间来记录某一账户的交易情况。但是，很多人却喜欢收到的银行对账单是按照支票号码的顺序来排列的。这是因为，人们通常都是按照支票号码的顺序来开出支票的，而商人也通常都是根据支票编号的顺序兑付支票。这一问题是将按交易时间排序的序列转换成按支票号排序的序列，它实质上是一个对几乎有序的输入序列进行排序的问题。请证明：在这个问题上，INSERTION-SORT 的性能往往要优于 QUICKSORT？

177 ~ 178

7.2-5 假设快速排序的每一层所做的划分的比例都是 $1-\alpha:\alpha$，其中 $0<\alpha\leqslant 1/2$ 且是一个常数。试证明：在相应的递归树中，叶结点的最小深度大约是 $-\lg n/\lg\alpha$，最大深度大约是 $-\lg n/\lg(1-\alpha)$（无需考虑整数舍入问题）。

***7.2-6** 试证明：在一个随机输入数组上，对于任何常数 $0<\alpha\leqslant 1/2$，PARTITION 产生比 $1-\alpha:\alpha$ 更平衡的划分的概率约为 $1-2\alpha$。

7.3 快速排序的随机化版本

在讨论快速排序的平均情况性能的时候，我们的前提假设是：输入数据的所有排列都是等概率的。但是在实际工程中，这个假设并不会总是成立（见练习 7.2-4）。正如在 5.3 节中我们所看到的那样，有时我们可以通过在算法中引入随机性，从而使得算法对于所有的输入都能获得较好的期望性能。很多人都选择随机化版本的快速排序作为大数据输入情况下的排序算法。

在 5.3 节中，我们通过显式地对输入进行重新排列，使得算法实现随机化。当然，对于快速排序我们也可以这么做。但如果采用一种称为**随机抽样**(random sampling)的随机化技术，那么可以使得分析变得更加简单。与始终采用 $A[r]$ 作为主元的方法不同，随机抽样是从子数组 $A[p..r]$ 中随机选择一个元素作为主元。为达到这一目的，首先将 $A[r]$ 与从 $A[p..r]$ 中随机选出的一个元素交换。通过对序列 p，…，r 的随机抽样，我们可以保证主元元素 $x=A[r]$ 是等概率地从子数组的 $r-p+1$ 个元素中选取的。因为主元元素是随机选取的，我们期望在平均情况下，对输入数组的划分是比较均衡的。

对 PARTITION 和 QUICKSORT 的代码的改动非常小。在新的划分程序中，我们只是在真正进行划分前进行一次交换：

```
RANDOMIZED-PARTITION (A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

新的快速排序不再调用 PARTITION，而是调用 RANDOMIZED-PARTITION：

```
RANDOMIZED-QUICKSORT (A, p, r)
1  if p<r
2       q = RANDOMIZED-PARTITION (A, p, r)
3       RANDOMIZED-QUICKSORT (A, p, q-1)
4       RANDOMIZED-QUICKSORT (A, q+1, r)
```

179 我们将在下一节中分析这一算法。

练习

7.3-1 为什么我们分析随机化算法的期望运行时间，而不是其最坏运行时间呢？

7.3-2 在RANDOMIZED-QUICKSORT的运行过程中，在最坏情况下，随机数生成器RANDOM被调用了多少次？在最好情况下呢？以Θ符号的形式给出你的答案？

7.4 快速排序分析

在7.2节中，我们给出了在最坏情况下快速排序性能的直观分析，以及它速度比较快的原因。在本节中，我们要给出快速排序性能的更严谨的分析。我们首先从最坏情况分析开始，其方法可以用于QUICKSORT和RANDOMIZED-QUICKSORT的分析，然后给出RANDOMIZED-QUICKSORT的期望运行时间。

7.4.1 最坏情况分析

在7.2节中，我们可以看到，在最坏情况下，快速排序的每一层递归的时间复杂度是$\Theta(n^2)$。从直观上来看，这也就是最坏情况下的运行时间。下面来证明这一点。

利用代入法(见4.3节)，我们可以证明快速排序的时间复杂度为$O(n^2)$。假设$T(n)$是最坏情况下QUICKSORT在输入规模为n的数据集合上所花费的时间，则有递归式：

$$T(n) = \max_{0\leqslant q\leqslant n-1}(T(q)+T(n-q-1))+\Theta(n) \tag{7.1}$$

因为PARTITION函数生成的两个子问题的规模加总为$n-1$，所以参数q的变化范围是0到$n-1$。我们不妨猜测$T(n)\leqslant cn^2$成立，其中c为常数。将此式代入递归式(7.1)中，得：

$$\begin{aligned}T(n) &\leqslant \max_{0\leqslant q\leqslant n-1}(cq^2+c(n-q-1)^2)+\Theta(n)\\ &= c\cdot\max_{0\leqslant q\leqslant n-1}(q^2+(n-q-1)^2)+\Theta(n)\end{aligned}$$

表达式$q^2+(n-q-1)^2$在参数取值区间$0\leqslant q\leqslant n-1$的端点上取得最大值。由于该表达式对于$q$的二阶导数是正的(见练习7.4-3)，我们可以得到表达式的上界$\max_{0\leqslant q\leqslant n-1}(q^2+(n-q-1)^2)\leqslant$ 180
$(n-1)^2=n^2-2n+1$，将其代入上式的$T(n)$中，我们得到：

$$T(n)\leqslant cn^2-c(2n-1)+\Theta(n)\leqslant cn^2$$

因为我们可以选择一个足够大的常数c，使得$c(2n-1)$项能显著大于$\Theta(n)$项，所以有$T(n)=O(n^2)$。在7.2节中，我们看到了特例：当划分非平衡的时候，快速排序的运行时间为$\Omega(n^2)$。此外，在练习7.4-1中，要求你证明递归式(7.1)有另一个解$T(n)=\Omega(n^2)$。因此，快速排序的(最坏情况)运行时间是$\Theta(n^2)$。

7.4.2 期望运行时间

我们已经从直观上了解了为什么RANDOMIZED-QUICKSORT的期望运行时间是$O(n\lg n)$：如果在递归的每一层上，RANDOMIZED-PARTITION将任意常数比例的元素划分到一个子数组中，则算法的递归树的深度为$\Theta(\lg n)$，并且每一层上的工作量都是$O(n)$。即使在最不平衡的划分情况下，会增加一些新的层次，但总的运行时间仍然保持是$O(n\lg n)$。要准确地分析RANDOMIZED-QUICKSORT的期望运行时间，首先要理解划分操作是如何进行的；然后，在此基础之上，推导出期望运行时间的一个$O(\lg n)$的界。有了这一期望运行时间的上界，再加上7.2节中得到的最好情况界$\Theta(n\lg n)$，我们就能得到$\Theta(n\lg n)$这一期望运行时间。在这里，假设待排序的元素始终是互异的。

运行时间和比较操作

QUICKSORT和RANDOMIZED-QUICKSORT除了如何选择主元元素有差异以外，其他方面完全相同。因此，我们可以在讨论QUICKSORT和PARTITION的基础上分析RANDOMIZED-QUICKSORT。其中，RANDOMIZED-QUICKSORT随机地从子数组中选择元素作为主元元素。

QUICKSORT 的运行时间是由在 PARTITION 操作上所花费的时间决定的。每次对 PARTITION 的调用时，都会选择一个主元元素，而且该元素不会被包含在后续的对 QUICKSORT 和 PARTITION 的递归调用中。因此，在快速排序算法的整个执行期间，至多只可能调用 PARTITION 操作 n 次。调用一次 PARTITION 的时间为 $O(1)$再加上一段循环时间。
181 这段时间与第 3～6 行中 **for** 循环的迭代次数成正比。这一 **for** 循环的每一轮迭代都要在第 4 行进行一次比较：比较主元元素与数组 A 中另一个元素。因此，如果我们可以统计第 4 行被执行的总次数，就能够给出在 QUICKSORT 的执行过程中，**for** 循环所花时间的界了。

引理 7.1 *当在一个包含 n 个元素的数组上运行 QUICKSORT 时，假设在 PARTITION 的第 4 行中所做比较的次数为 X，那么 QUICKSORT 的运行时间为 $O(n+X)$。*

证明 根据上面的讨论，算法最多对 PARTITION 调用 n 次。每次调用都包括一个固定的工作量和执行若干次 **for** 循环。在每一次 **for** 循环中，都要执行第 4 行。 ■

因此，我们的目标是计算出 X，即所有对 PARTITION 的调用中，所执行的总的比较次数。我们并不打算分析在每一次 PARTITION 调用中做了多少次比较，而是希望能够推导出关于总的比较次数的一个界。为此，我们必须了解算法在什么时候对数组中的两个元素进行比较，什么时候不进行比较。为了便于分析，我们将数组 A 的各个元素重新命名为 z_1，z_2，…，z_n，其中 z_i 是数组 A 中第 i 小的元素。此外，我们还定义 $Z_{ij}=\{z_i, z_{i+1}, \cdots, z_j\}$为 z_i 与 z_j 之间(含 i 和 j)的元素集合。

算法什么时候会比较 z_i 和 z_j 呢？为了回答这个问题，我们首先注意到每一对元素至多比较一次。为什么呢？因为各个元素只与主元元素进行比较，并且在某一次 PARTITION 调用结束之后，该次调用中所用到的主元元素就再也不会与任何其他元素进行比较了。

我们的分析要用到指示器随机变量(见 5.2 节)。定义

$$X_{ij} = \mathrm{I}\{z_i \text{ 与 } z_j \text{ 进行比较}\}$$

其中我们考虑的是比较操作是否在算法执行过程中任意时间发生，而不是局限在循环的一次迭代或对 PARTITION 的一次调用中是否发生。因为每一对元素至多被比较一次，所以我们可以很容易地刻画出算法的总比较次数：

$$X = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}$$

对上式两边取期望，再利用期望值的线性特性和引理 5.1，可以得到：

182
$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\mathrm{E}[X_{ij}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\Pr\{z_i \text{ 与 } z_j \text{ 进行比较}\} \tag{7.2}$$

上式中的 $\Pr\{z_i$ 与 z_j 进行比较$\}$还需要进一步计算。在我们的分析中，假设 RANDOMIZED-PARTITION 随机且独立地选择主元。

让我们考虑两个元素何时不会进行比较的情况。考虑快速排序的一个输入，它是由数字 1 到 10 所构成(顺序可以是任意的)，并假设第一个主元是 7。那么，对 PARTITION 的第一次调用就将这些输入数字划分成两个集合：{1，2，3，4，5，6}和{8，9，10}。在这一过程中，主元 7 要与所有其他元素进行比较。但是，第一个集合中任何一个元素(例如 2)没有(也不会)与第二个集合中的任何元素(例如 9)进行比较。

通常我们假设每个元素的值是互异的，因此，一旦一个满足 $z_i<x<z_j$ 的主元 x 被选择后，我们就知道 z_i 和 z_j 以后再也不可能被比较了。另一种情况，如果 z_i 在 Z_{ij} 中的所有其他元素之前被选为主元，那么 z_j 就将与 Z_{ij} 中除了它自身以外的所有元素进行比较。类似地，如果 z_j 在 Z_{ij} 中其他元素之前被选为主元，那么 z_j 将与 Z_{ij} 中除自身以外的所有元素进行比较。在我们的例子中，值 7 和 9 要进行比较，因为 7 是 $Z_{7,9}$ 中被选为主元的第一个元素。与之相反的是，值 2 和 9 则始终不会被比较，因为从 $Z_{2,9}$ 中选择的第一个主元为 7。因此，z_i 与 z_j 会进行比较，当且仅当

Z_{ij} 中将被选为主元的第一个元素是 z_i 或者 z_j。

我们现在来计算这一事件发生的概率。在 Z_{ij} 中的某个元素被选为主元之前，整个集合 Z_{ij} 的元素都属于某一划分的同一分区。因此，Z_{ij} 中的任何元素都会等可能地被首先选为主元。因为集合 Z_{ij} 中有 $j-i+1$ 个元素，并且主元的选择是随机且独立的，所以任何元素被首先选为主元的概率是 $1/(j-i+1)$。于是，我们有：

$$\begin{aligned}\Pr\{z_i \text{ 与 } z_j \text{ 进行比较}\} &= \Pr\{z_i \text{ 或 } z_j \text{ 是集合 } Z_{ij} \text{ 中选出的第一个主元}\} \\ &= \Pr\{z_i \text{ 是集合 } Z_{ij} \text{ 中选出的第一个主元}\} \\ &\quad + \Pr\{z_j \text{ 是集合 } Z_{ij} \text{ 中选出的第一个主元}\} \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1} \qquad (7.3)\end{aligned}$$

183

上式中第二行成立的原因在于其中涉及的两个事件是互斥的。将公式(7.2)和公式(7.3)综合起来，有：

$$\mathrm{E}[X] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{2}{j-i+1}$$

在求这个累加和时，可以将变量做个变换($k=j-i$)，并利用公式(A.7)中给出的有关调和级数的界，得到：

$$\mathrm{E}[X] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{2}{j-i+1} = \sum_{i=1}^{n-1}\sum_{k=1}^{n-i}\frac{2}{k+1} < \sum_{i=1}^{n-1}\sum_{k=1}^{n}\frac{2}{k} = \sum_{i=1}^{n-1}O(\lg n) = O(n\lg n) \qquad (7.4)$$

于是，我们可以得出结论：使用 RANDOMIZED-PARTITION，在输入元素互异的情况下，快速排序算法的期望运行时间为 $O(n\lg n)$。

练习

7.4-1 证明：在递归式

$$T(n) = \max_{0\leqslant q\leqslant n-1}(T(q)+T(n-q-1)) + \Theta(n)$$

中，$T(n)=\Omega(n^2)$。

7.4-2 证明：在最好情况下，快速排序的运行时间为 $\Omega(n\lg n)$。 184

7.4-3 证明：在 $q=0, 1, \cdots, n-1$ 区间内，当 $q=0$ 或 $q=n-1$ 时，$q^2+(n-q-1)^2$ 取得最大值。

7.4-4 证明：RANDOMIZED-QUICKSORT 期望运行时间是 $\Omega(n\lg n)$。

7.4-5 当输入数据已经“几乎有序”时，插入排序速度很快。在实际应用中，我们可以利用这一特点来提高快速排序的速度。当对一个长度小于 k 的子数组调用快速排序时，让它不做任何排序就返回。当上层的快速排序调用返回后，对整个数组运行插入排序来完成排序过程。试证明：这一排序算法的期望时间复杂度为 $O(nk+n\lg(n/k))$。分别从理论和实践的角度说明我们应该如何选择 k?

***7.4-6** 考虑对 PARTITION 过程做这样的修改：从数组 A 中随机选出三个元素，并用这三个元素的中位数(即这三个元素按大小排在中间的值)对数组进行划分。求以 α 的函数形式表示的、最坏划分比例为 $\alpha:(1-\alpha)$ 的近似概率，其中 $0<\alpha<1$。

思考题

7-1 (Hoare 划分的正确性) 本章中的 PARTITION 算法并不是其最初的版本。下面给出的是最早由 C. R. Hoare 所设计的划分算法：

```
HOARE-PARTITION(A, p, r)
1  x = A[p]
```

```
2  i = p − 1
3  j = r + 1
4  while TRUE
5      repeat
6          j = j − 1
7      until A[j]≤x
8      repeat
9          i = i + 1
10     until A[j]≥x
11     if i<j
12         exchange A[i] with A[j]
13     else return j
```

a. 试说明 HOARE-PARTITION 在数组 $A=\langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21\rangle$ 上的操作过程，并说明在每一次执行第 4～13 行 **while** 循环时数组元素的值和辅助变量的值。

185

后续的三个问题要求读者仔细论证 HOARE-PARTITION 的正确性。在这里假设子数组 $A[p..r]$ 至少包含来 2 个元素，试证明下列问题：

b. 下标 i 和 j 可以使我们不会访问在子数组 $A[p..r]$ 以外的数组 A 的元素。

c. 当 HOARE-PARTITION 结束时，它返回的值 j 满足 $p\leqslant j<r$。

d. 当 HOARE-PARTITION 结束时，$A[p..j]$ 中的每一个元素都小于或等于 $A[j+1..r]$ 中的元素。

在 7.1 节的 PARTITION 过程中，主元(原来存储在 $A[r]$ 中)是与它所划分的两个分区分离的。与之对应，在 HOARE-PARTITION 中，主元(原来存储在 $A[p]$ 中)是存在于分区 $A[p..j]$ 或 $A[j+1..r]$ 中的。因为有 $p\leqslant j<r$，所以这一划分总是非平凡的。

e. 利用 HOARE-PARTITION，重写 QUICKSORT 算法。

7-2 (*针对相同元素值的快速排序*) 在 7.4.2 节对随机化快速排序的分析中，我们假设输入元素的值是互异的，在本题中，我们将看看如果这一假设不成立会出现什么情况。

a. 如果所有输入元素的值都相同，那么随机化快速排序的运行时间会是多少？

b. PARTITION 过程返回一个数组下标 q，使得 $A[p..q-1]$ 中的每个元素都小于或等于 $A[q]$，而 $A[q+1..r]$ 中的每个元素都大于 $A[q]$。修改 PARTITION 代码来构造一个新的 PARTITION′(A, p, r)，它排列 $A[p..r]$ 的元素，返回值是两个数组下标 q 和 t，其中 $p\leqslant q\leqslant t\leqslant r$，且有

- $A[q..t]$ 中的所有元素都相等。
- $A[p..q-1]$ 中的每个元素都小于 $A[q]$。
- $A[t+1..r]$ 中的每个元素都大于 $A[q]$。

与 PARTITION 类似，新构造的 PARTITION′的时间复杂度是 $\Theta(r-p)$。

c. 将 RANDOMIZED-QUICKSORT 过程改为调用 PARTITION′，并重新命名为 RANDOMIZED-QUICKSORT′。修改 QUICKSORT 的代码构造一个新的 QUICKSORT′(A, p, r)，它调用 RANDOMIZED-PARTITION′，并且只有分区内的元素互不相同的时候才做递归调用。

186

d. 在 QUICKSORT′中，应该如何改变 7.4.2 节中的分析方法，从而避免所有元素都是互异的这一假设？

7-3 (*另一种快速排序的分析方法*) 对随机化版本的快速排序算法，还有另一种性能分析方法，这一方法关注于每一次单独递归调用的期望运行时间，而不是比较的次数。

a. 证明：给定一个大小为 n 的数组，任何特定元素被选为主元的概率为 $1/n$。利用这一点来定义指示器随机变量 $X_i=\mathrm{I}\{\text{第 } i \text{ 小的元素被选为主元}\}$，$\mathrm{E}[X_i]$ 是什么？

b. 设 $T(n)$是一个表示快速排序在一个大小为 n 的数组上的运行时间的随机变量，试证明：

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{q=1}^{n} X_q (T(q-1) + T(n-q) + \Theta(n))\right] \tag{7.5}$$

c. 证明公式(7.5)可以重写为：

$$\mathrm{E}[T(n)] = \frac{2}{n}\sum_{q=2}^{n-1} \mathrm{E}[T(q)] + \Theta(n) \tag{7.6}$$

d. 证明：

$$\sum_{k=2}^{n-1} k \lg k \leqslant \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \tag{7.7}$$

(提示：可以将该累加式分成两个部分，一部分是 $k=2, 3, \cdots, \lceil n/2 \rceil - 1$，另一部分是 $k=\lceil n/2 \rceil, \cdots, n-1$。)

e. 利用公式(7.7)中给出的界证明：公式(7.6)中的递归式有解 $\mathrm{E}[T(n)] = \Theta(n \lg n)$。(提示：使用代入法，证明对于某个正常数 a 和足够大的 n，有 $\mathrm{E}[T(n)] \leqslant an \lg n$。) 187

7-4 (快速排序的栈深度) 7.1 节中的 QUICKSORT 算法包含了两个对其自身的递归调用。在调用 PARTITION 后，QUICKSORT 分别递归调用了左边的子数组和右边的子数组。QUICKSORT 中的第二个递归调用并不是必须的。我们可以用一个循环控制结构来代替它。这一技术称为**尾递归**，好的编译器都提供这一功能。考虑下面这个版本的快速排序，它模拟了尾递归情况：

```
TAIL-RECURSIVE-QUICKSORT(A, p, r)
1  while p < r
2      // Partition and sort left subarray.
3      q = PARTITION(A, p, r)
4      TAIL-RECURSIVE-QUICKSORT(A, p, q-1)
5      p = q+1
```

a. 证明：TAIL-RECURSIVE-QUICKSORT(A, 1, $A.length$)能正确地对数组 A 进行排序。编译器通常使用**栈**来存储递归执行过程中的相关信息，包括每一次递归调用的参数等。最新调用的信息存在栈的顶部，而第一次调用的信息存在栈的底部。当一个过程被调用时，其相关信息被**压入**栈中；当它结束时，其信息则被**弹出**。因为我们假设数组参数是用指针来指示的，所以每次过程调用只需要 $O(1)$的栈空间。**栈深度**是在一次计算中会用到的栈空间的最大值。

b. 请描述一种场景，使得针对一个包含 n 个元素数组的 TAIL-RECURSIVE-QUICKSORT 的栈深度是 $\Theta(n)$。

c. 修改 TAIL-RECURSIVE-QUICKSORT 的代码，使其最坏情况下栈深度是 $\Theta(\lg n)$，并且能够保持 $O(n \lg n)$的期望时间复杂度。

7-5 (三数取中划分) 一种改进 RANDOMIZED-QUICKSORT 的方法是在划分时，要从子数组中更细致地选择作为主元的元素(而不是简单地随机选择)。常用的做法是三数取中法：从子数组中随机选出三个元素，取其中位数作为主元(见练习 7.4-6)。对于这个问题的分析，我们不妨假设数组 $A[1..n]$的元素是互异的且有 $n \geqslant 3$。我们用 $A'[1..n]$来表示已排好序的 188
数组。用三数取中法选择主元 x，并定义 $p_i = \Pr\{x = A'[i]\}$。

a. 对于 $i=2, 3, \cdots, n-1$，请给出以 n 和 i 表示的 p_i 的准确表达式(注意 $p_1 = p_n = 0$)。

b. 与平凡实现相比，在这种实现中，选择 $x = A'[\lfloor (n+1)/2 \rfloor]$(即 $A[1..n]$的中位数)的值作为主元的概率增加了多少？假设 $n \to \infty$，请给出这一概率的极限值。

c. 如果我们定义一个“好”划分意味着主元选择 $x = A'[i]$，其中 $n/3 \leqslant i \leqslant 2n/3$。与平凡实现

相比，这种实现中得到一个好划分的概率增加了多少？（提示：用积分来近似累加和。）

d. 证明：对快速排序而言，三数取中法只影响其时间复杂度 $\Omega(n \lg n)$ 的常数项因子。

7-6 （对区间的模糊排序） 考虑这样的一种排序问题：我们无法准确知道待排序的数字是什么。但对于每一个数，我们知道它属于实数轴上的某个区间。也就是说，我们得到了 n 个形如 $[a_i, b_i]$ 的闭区间，其中 $a_i \leqslant b_i$。我们的目标是实现这些区间的**模糊排序**，即对 $j=1, 2, \cdots, n$，生成一个区间的排列 $\langle i_1, i_2, \cdots, i_n \rangle$，且存在 $c_j \in [a_{i_j}, b_{i_j}]$，满足 $c_1 \leqslant c_2 \leqslant \cdots \leqslant c_n$。

a. 为 n 个区间的模糊排序设计一个随机算法。你的算法应该具有算法的一般结构，它可以对左边端点（即 a_i 的值）进行快速排序，同时它也能利用区间的重叠性质来改善时间性能。（当区间重叠越来越多的时候，区间的模糊排序问题会变得越来越容易。你的算法应能充分利用这一重叠性质。）

b. 证明：在一般情况下，你的算法的期望运行时间为 $\Theta(n \lg n)$。但是，当所有的区间都有重叠的时候，算法的期望运行时间为 $\Theta(n)$（也就是说，存在一个值 x，对所有的 i，都有 $x \in [a_i, b_i]$。）你的算法不必显式地检查这种情况，而是随着重叠情况的增加，算法的性能自然地提高。

189

本章注记

快速排序是由 Hoare[170]首先提出的。思考题 7-1 中给出了 Hoare 的原始版本。7.1 节中给出的 PARTITION 是由 N. Lomuto 提出的。而 7.4 节中的分析是由 Avrim Blum 给出的。Sedgewick[305]和 Bentley[43]都对实现的细节及其影响给出了很好的描述。

Mcllroy[248]说明了如何设计出一个“杀手级对手”（killer adversary），它能够产生一个数组，在这个数组上，快速排序的几乎所有实现的运行时间都是 $\Theta(n^2)$。如果实现是随机化的，这一对手可以在观察到快速排序算法的随机选择之后，再产生出这一数组。

190