在第1章中给出了两个算法，但是它们都不是很有效的算法。第一个算法我们称为1A，是把这些元素读入数组并将它们排序，返回适当的元素。假设使用的是简单的排序算法，则运行时间为 $O(N^2)$。另一个算法叫做1B，是将 k 个元素读入一个数组并将它们排序。这些元素中的最小者在第 k 个位置上。我们一个一个地处理其余的元素。当一个元素开始被处理时，它先与数组中第 k 个元素比较，如果该元素大，那么将第 k 个元素除去，而这个新元素则被放在其余 $k-1$ 个元素中正确的位置上。当算法结束时，第 k 个位置上的元素就是问题的解答。该方法的运行时间为 $O(N \cdot k)$(为什么?)。如果 $k=\lceil N/2 \rceil$，那么这两种算法都是 $O(N^2)$。注意，对于任意的 k，我们可以求解对称的问题：找出第 $(N-k+1)$ 个最小的元素，从而 $k=\lceil N/2 \rceil$ 实际上是这两个算法的最困难的情况。这刚好也是最有趣的情形，因为 k 的这个值称为中位数(median)。

我们在这里给出两个算法，在 $k=\lceil N/2 \rceil$ 的极端情形它们均以 $O(N \log N)$ 运行，这是明显的改进。

算法 6A

为了简单起见，假设我们只考虑找出第 k 个最小的元素。该算法很简单。我们将 N 个元素读入一个数组。然后对该数组应用 buildHeap 算法。最后，执行 k 次 deleteMin 操作。从该堆最后提取的元素就是我们的答案。显然，只要改变堆序性质，就可以求解原始的问题：找出第 k 个最大的元素。

这个算法的正确性应该是显然的。如果使用 buildHeap，则构造堆的最坏情形用时 $O(N)$，而每次 deleteMin 用时 $O(\log N)$。由于有 k 次 deleteMin，因此我们得到总的运行时间为 $O(N+k \log N)$。如果 $k=O(N/\log N)$，那么运行时间取决于 buildHeap 操作，即 $O(N)$。对于大的 k 值，运行时间为 $O(k \log N)$。如果 $k=\lceil N/2 \rceil$，那么运行时间为 $\Theta(N \log N)$。

注意，如果我们对 $k=N$ 运行该程序并在元素离开堆时记录它们的值，那么实际上已经对输入文件以时间 $O(N \log N)$ 做了排序。在第7章，我们将细化该想法，得到一种快速的排序算法，叫做堆排序(heapsort)。

算法 6B

关于第2个算法，我们回到原始问题，找出第 k 个最大的元素。我们使用算法1B的思路。在任一时刻我们都将维持 k 个最大元素的集合 S。在前 k 个元素读入以后，当再读入一个新的元素时，该元素将与第 k 个最大元素进行比较，记这第 k 个最大的元素为 S_k。注意，S_k 是 S 中最小的元素。如果新的元素更大，那么用新元素代替 S 中的 S_k。此时，S 将有一个新的最小元素，它可能是新添加进来的元素，也可能不是。在输入终了时，我们找到 S 中的最小的元素，将其返回，它就是答案。

这基本上与第1章中描述的算法相同。不过，这里我们使用一个堆来实现 S。前 k 个元素通过调用一次 buildHeap 以总时间 $O(k)$ 被置入堆中。处理每个其余的元素的时间为 $O(1)$，用于检测是否元素进入 S，再加上时间 $O(\log k)$，用于在必要时删除 S_k 并插入新元素。因此，总的时间是 $O(k+(N-k)\log k)=O(N \log k)$。该算法也给出找出中位数的时间界 $\Theta(N \log N)$。

在第7章，我们将看到如何以平均时间 $O(N)$ 解决这个问题。在第10章，我们将看到一个以 $O(N)$ 最坏情形时间求解该问题的算法，虽然不实用但却很精妙。

6.4.2 事件模拟

在3.7.3节我们描述了一个重要的排队问题。在那里我们有一个系统，比如银行，顾客们到达并排队等待直到 k 个出纳员有一个腾出手来。顾客的到达情况由概率分布函数控制，服务时间(一旦出纳员腾出时间用于服务的时间量)也是如此。我们的兴趣在于一位顾客平均必须要等多久或所排的队伍可能有多长这类统计问题。