kth smallest element is in position $k - 1$ (because arrays start at index 0). This destroys the original ordering; if this is not desirable, then a copy must be made.

Using a median-of-three pivoting strategy makes the chance of the worst case occurring almost negligible. By carefully choosing the pivot, however, we can eliminate the quadratic worst case and ensure an $O(N)$ algorithm. The overhead involved in doing this is considerable, so the resulting algorithm is mostly of theoretical interest. In Chapter 10, we will examine the linear-time worst-case algorithm for selection, and we shall also see an interesting technique of choosing the pivot that results in a somewhat faster selection algorithm in practice.

7.8 A General Lower Bound for Sorting

Although we have $O(N \log N)$ algorithms for sorting, it is not clear that this is as good as we can do. In this section, we prove that any algorithm for sorting that uses only comparisons requires $\Omega(N \log N)$ comparisons (and hence time) in the worst case, so that mergesort and heapsort are optimal to within a constant factor. The proof can be extended to show that $\Omega(N \log N)$ comparisons are required, even on average, for any sorting algorithm that uses only comparisons, which means that quicksort is optimal on average to within a constant factor.

Specifically, we will prove the following result: Any sorting algorithm that uses only comparisons requires $\lceil \log(N!) \rceil$ comparisons in the worst case and $\log(N!)$ comparisons on average. We will assume that all N elements are distinct, since any sorting algorithm must work for this case.

7.8.1 Decision Trees

A **decision tree** is an abstraction used to prove lower bounds. In our context, a decision tree is a binary tree. Each node represents a set of possible orderings, consistent with comparisons that have been made, among the elements. The results of the comparisons are the tree edges.

The decision tree in Figure 7.20 represents an algorithm that sorts the three elements a, b, and c. The initial state of the algorithm is at the root. (We will use the terms *state* and *node* interchangeably.) No comparisons have been done, so all orderings are legal. The first comparison that *this particular* algorithm performs compares a and b. The two results lead to two possible states. If $a < b$, then only three possibilities remain. If the algorithm reaches node 2, then it will compare a and c. Other algorithms might do different things; a different algorithm would have a different decision tree. If $a > c$, the algorithm enters state 5. Since there is only one ordering that is consistent, the algorithm can terminate and report that it has completed the sort. If $a < c$, the algorithm cannot do this, because there are two possible orderings and it cannot possibly be sure which is correct. In this case, the algorithm will require one more comparison.

Every algorithm that sorts by using only comparisons can be represented by a decision tree. Of course, it is only feasible to draw the tree for extremely small input sizes. The number of comparisons used by the sorting algorithm is equal to the depth of the deepest

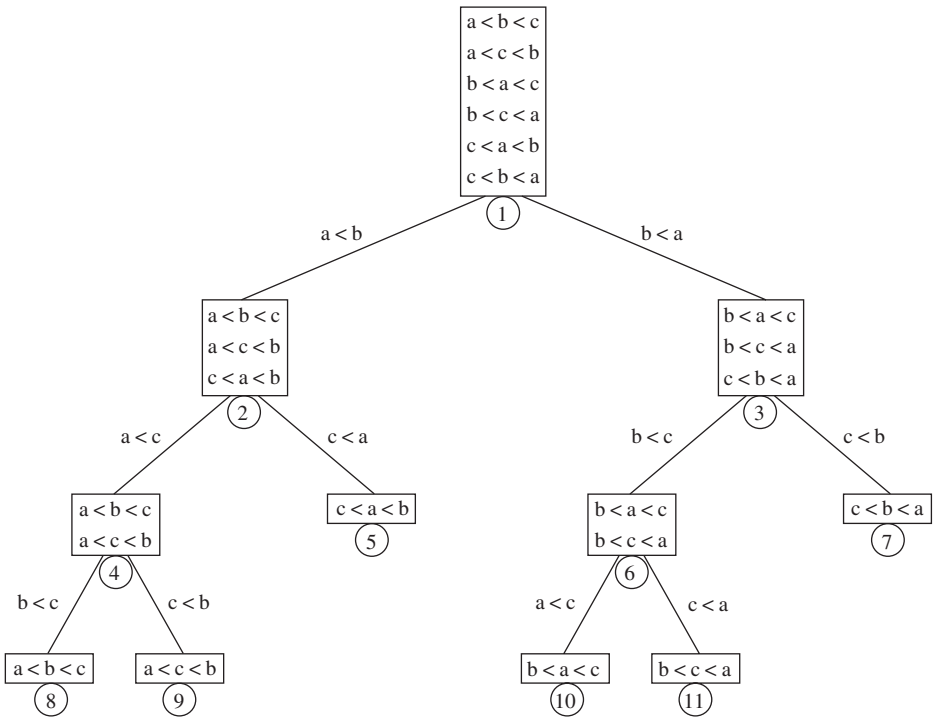

Figure 7.20 A decision tree for three-element sort

leaf. In our case, this algorithm uses three comparisons in the worst case. The average number of comparisons used is equal to the average depth of the leaves. Since a decision tree is large, it follows that there must be some long paths. To prove the lower bounds, all that needs to be shown are some basic tree properties.

Lemma 7.1
Let T be a binary tree of depth d. Then T has at most 2^d leaves.

Proof
The proof is by induction. If $d = 0$, then there is at most one leaf, so the basis is true. Otherwise, we have a root, which cannot be a leaf, and a left and right subtree, each of depth at most $d - 1$. By the induction hypothesis, they can each have at most 2^{d-1} leaves, giving a total of at most 2^d leaves. This proves the lemma.

Lemma 7.2
A binary tree with L leaves must have depth at least $\lceil \log L \rceil$.

Proof
Immediate from the preceding lemma.

Theorem 7.6
Any sorting algorithm that uses only comparisons between elements requires at least $\lceil \log(N!) \rceil$ comparisons in the worst case.

Proof
A decision tree to sort N elements must have $N!$ leaves. The theorem follows from the preceding lemma.

Theorem 7.7
Any sorting algorithm that uses only comparisons between elements requires $\Omega(N \log N)$ comparisons.

Proof
From the previous theorem, $\log(N!)$ comparisons are required.

$$\begin{aligned}
\log(N!) &= \log(N(N-1)(N-2)\cdots(2)(1)) \\
&= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1 \\
&\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log(N/2) \\
&\geq \frac{N}{2} \log \frac{N}{2} \\
&\geq \frac{N}{2} \log N - \frac{N}{2} \\
&= \Omega(N \log N)
\end{aligned}$$

This type of lower-bound argument, when used to prove a worst-case result, is sometimes known as an **information-theoretic** lower bound. The general theorem says that if there are P different possible cases to distinguish, and the questions are of the form YES/NO, then $\lceil \log P \rceil$ questions are always required in some case by any algorithm to solve the problem. It is possible to prove a similar result for the average-case running time of any comparison-based sorting algorithm. This result is implied by the following lemma, which is left as an exercise: Any binary tree with L leaves has an average depth of at least $\log L$.

7.9 Decision-Tree Lower Bounds for Selection Problems

Section 7.8 employed a decision-tree argument to show the fundamental lower bound that any comparison-based sorting algorithm must use roughly $N \log N$ comparisons. In this section, we show additional lower bounds for selection in an N-element collection, specifically

1. $N - 1$ comparisons are necessary to find the smallest item.
2. $N + \lceil \log N \rceil - 2$ comparisons are necessary to find the two smallest items.
3. $\lceil 3N/2 \rceil - O(\log N)$ comparisons are necessary to find the median.