8 Sorting in Linear Time

We have now introduced several algorithms that can sort n numbers in $O(n \lg n)$ time. Merge sort and heapsort achieve this upper bound in the worst case; quicksort achieves it on average. Moreover, for each of these algorithms, we can produce a sequence of n input numbers that causes the algorithm to run in $\Omega(n \lg n)$ time.

These algorithms share an interesting property: *the sorted order they determine is based only on comparisons between the input elements*. We call such sorting algorithms ***comparison sorts***. All the sorting algorithms introduced thus far are comparison sorts.

In Section 8.1, we shall prove that any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case to sort n elements. Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.

Sections 8.2, 8.3, and 8.4 examine three sorting algorithms—counting sort, radix sort, and bucket sort—that run in linear time. Needless to say, these algorithms use operations other than comparisons to determine the sorted order. Consequently, the $\Omega(n \lg n)$ lower bound does not apply to them.

8.1 Lower bounds for sorting

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$. That is, given two elements a_i and a_j, we perform one of the tests $a_i < a_j$, $a_i \le a_j$, $a_i = a_j$, $a_i \ge a_j$, or $a_i > a_j$ to determine their relative order. We may not inspect the values of the elements or gain order information about them in any other way.

In this section, we assume without loss of generality that all of the input elements are distinct. Given this assumption, comparisons of the form $a_i = a_j$ are useless, so we can assume that no comparisons of this form are made. We also note that the comparisons $a_i \le a_j$, $a_i \ge a_j$, $a_i > a_j$, and $a_i < a_j$ are all equivalent in that

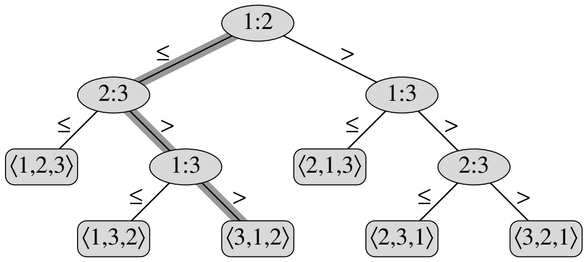

Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i\!:\!j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \le a_1 = 6 \le a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, so the decision tree must have at least 6 leaves.

they yield identical information about the relative order of a_i and a_j. We therefore assume that all comparisons have the form $a_i \le a_j$.

The decision-tree model

Comparison sorts can be viewed abstractly in terms of ***decision trees***. A decision tree is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored. Figure 8.1 shows the decision tree corresponding to the insertion sort algorithm from Section 2.1 operating on an input sequence of three elements.

In a decision tree, each internal node is annotated by $i\!:\!j$ for some i and j in the range $1 \le i, j \le n$, where n is the number of elements in the input sequence. Each leaf is annotated by a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$. (See Section C.1 for background on permutations.) The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf. At each internal node, a comparison $a_i \le a_j$ is made. The left subtree then dictates subsequent comparisons for $a_i \le a_j$, and the right subtree dictates subsequent comparisons for $a_i > a_j$. When we come to a leaf, the sorting algorithm has established the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. Because any correct sorting algorithm must be able to produce each permutation of its input, a necessary condition for a comparison sort to be correct is that each of the $n!$ permutations on n elements must appear as one of the leaves of the decision tree, and that each of these leaves must be reachable from the root by a path corresponding to an actual execution of the comparison sort. (We shall refer to such leaves as "reachable.") Thus, we shall consider only decision trees in which each permutation appears as a reachable leaf.

A lower bound for the worst case

The length of the longest path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs. Consequently, the worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree. A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sort algorithm. The following theorem establishes such a lower bound.

Theorem 8.1
Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof From the preceding discussion, it suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements. Because each of the $n!$ permutations of the input appears as some leaf, we have $n! \le l$. Since a binary tree of height h has no more than 2^h leaves, we have

$$n! \le l \le 2^h ,$$

which, by taking logarithms, implies

$$\begin{aligned} h &\ge \lg(n!) && \text{(since the lg function is monotonically increasing)} \\ &= \Omega(n \lg n) && \text{(by equation (3.18))} . \end{aligned}$$

■

Corollary 8.2
Heapsort and merge sort are asymptotically optimal comparison sorts.

Proof The $O(n \lg n)$ upper bounds on the running times for heapsort and merge sort match the $\Omega(n \lg n)$ worst-case lower bound from Theorem 8.1. ■

Exercises

8.1-1
What is the smallest possible depth of a leaf in a decision tree for a comparison sort?

8.1-2
Obtain asymptotically tight bounds on $\lg(n!)$ without using Stirling's approximation. Instead, evaluate the summation $\sum_{k=1}^{n} \lg k$ using techniques from Section A.2.

8.1-3

Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n. What about a fraction of $1/n$ of the inputs of length n? What about a fraction $1/2^n$?

8.1-4

You are given a sequence of n elements to sort. The input sequence consists of n/k subsequences, each containing k elements. The elements in a given subsequence are all smaller than the elements in the succeeding subsequence and larger than the elements in the preceding subsequence. Thus, all that is needed to sort the whole sequence of length n is to sort the k elements in each of the n/k subsequences. Show an $\Omega(n \lg k)$ lower bound on the number of comparisons needed to solve this variant of the sorting problem. (*Hint:* It is not rigorous to simply combine the lower bounds for the individual subsequences.)

8.2 Counting sort

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k. When $k = O(n)$, the sort runs in $\Theta(n)$ time.

The basic idea of counting sort is to determine, for each input element x, the number of elements less than x. This information can be used to place element x directly into its position in the output array. For example, if there are 17 elements less than x, then x belongs in output position 18. This scheme must be modified slightly to handle the situation in which several elements have the same value, since we don't want to put them all in the same position.

In the code for counting sort, we assume that the input is an array $A[1..n]$, and thus $length[A] = n$. We require two other arrays: the array $B[1..n]$ holds the sorted output, and the array $C[0..k]$ provides temporary working storage.

```
COUNTING-SORT(A, B, k)
for i ← 0 to k
     do C[i] ← 0
for j ← 1 to length[A]
     do C[A[j]] ← C[A[j]] + 1
▷ C[i] now contains the number of elements equal to i.
for i ← 1 to k
     do C[i] ← C[i] + C[i − 1]
▷ C[i] now contains the number of elements less than or equal to i.
for j ← length[A] downto 1
     do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] − 1
```

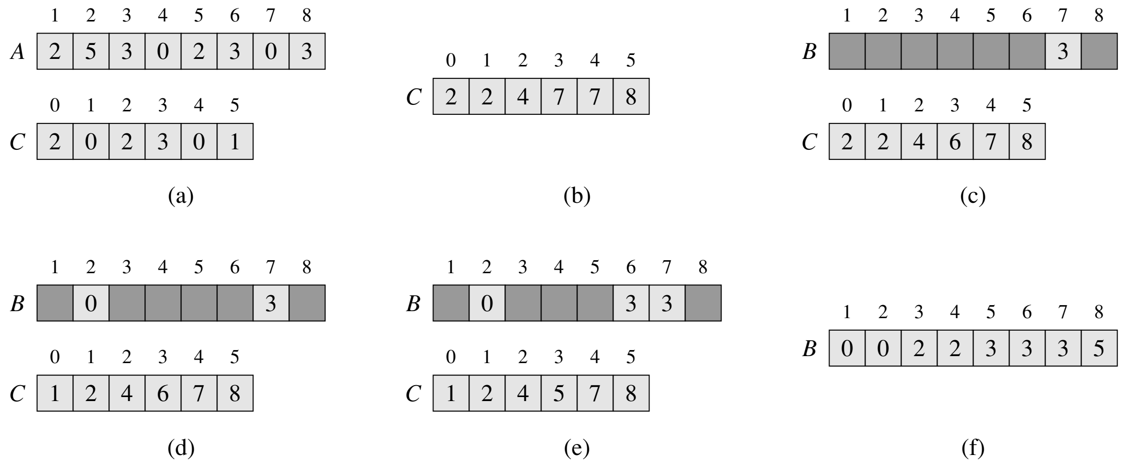

Figure 8.2 The operation of COUNTING-SORT on an input array $A[1 \,..\, 8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 4. **(b)** The array C after line 7. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 9–11, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

Figure 8.2 illustrates counting sort. After the initialization in the **for** loop of lines 1–2, we inspect each input element in the **for** loop of lines 3–4. If the value of an input element is i, we increment $C[i]$. Thus, after line 4, $C[i]$ holds the number of input elements equal to i for each integer $i = 0, 1, \ldots, k$. In lines 6–7, we determine for each $i = 0, 1, \ldots, k$, how many input elements are less than or equal to i by keeping a running sum of the array C.

Finally, in the **for** loop of lines 9–11, we place each element $A[j]$ in its correct sorted position in the output array B. If all n elements are distinct, then when we first enter line 9, for each $A[j]$, the value $C[A[j]]$ is the correct final position of $A[j]$ in the output array, since there are $C[A[j]]$ elements less than or equal to $A[j]$. Because the elements might not be distinct, we decrement $C[A[j]]$ each time we place a value $A[j]$ into the B array. Decrementing $C[A[j]]$ causes the next input element with a value equal to $A[j]$, if one exists, to go to the position immediately before $A[j]$ in the output array.

How much time does counting sort require? The **for** loop of lines 1–2 takes time $\Theta(k)$, the **for** loop of lines 3–4 takes time $\Theta(n)$, the **for** loop of lines 6–7 takes time $\Theta(k)$, and the **for** loop of lines 9–11 takes time $\Theta(n)$. Thus, the overall time is $\Theta(k+n)$. In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$.

Counting sort beats the lower bound of $\Omega(n \lg n)$ proved in Section 8.1 because it is not a comparison sort. In fact, no comparisons between input elements occur anywhere in the code. Instead, counting sort uses the actual values of the elements to index into an array. The $\Omega(n \lg n)$ lower bound for sorting does not apply when we depart from the comparison-sort model.

An important property of counting sort is that it is ***stable***: numbers with the same value appear in the output array in the same order as they do in the input array. That is, ties between two numbers are broken by the rule that whichever number appears first in the input array appears first in the output array. Normally, the property of stability is important only when satellite data are carried around with the element being sorted. Counting sort's stability is important for another reason: counting sort is often used as a subroutine in radix sort. As we shall see in the next section, counting sort's stability is crucial to radix sort's correctness.

Exercises

8.2-1
Using Figure 8.2 as a model, illustrate the operation of COUNTING-SORT on the array $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$.

8.2-2
Prove that COUNTING-SORT is stable.

8.2-3
Suppose that the **for** loop header in line 9 of the COUNTING-SORT procedure is rewritten as

9 **for** $j \leftarrow 1$ **to** *length*$[A]$

Show that the algorithm still works properly. Is the modified algorithm stable?

8.2-4
Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a \,..\, b]$ in $O(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

8.3 Radix sort

Radix sort is the algorithm used by the card-sorting machines you now find only in computer museums. The cards are organized into 80 columns, and in each column a hole can be punched in one of 12 places. The sorter can be mechanically "programmed" to examine a given column of each card in a deck and distribute the

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Figure 8.3 The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

card into one of 12 bins depending on which place has been punched. An operator can then gather the cards bin by bin, so that cards with the first place punched are on top of cards with the second place punched, and so on.

For decimal digits, only 10 places are used in each column. (The other two places are used for encoding nonnumeric characters.) A d-digit number would then occupy a field of d columns. Since the card sorter can look at only one column at a time, the problem of sorting n cards on a d-digit number requires a sorting algorithm.

Intuitively, one might want to sort numbers on their *most significant* digit, sort each of the resulting bins recursively, and then combine the decks in order. Unfortunately, since the cards in 9 of the 10 bins must be put aside to sort each of the bins, this procedure generates many intermediate piles of cards that must be kept track of. (See Exercise 8.3-5.)

Radix sort solves the problem of card sorting counterintuitively by sorting on the *least significant* digit first. The cards are then combined into a single deck, with the cards in the 0 bin preceding the cards in the 1 bin preceding the cards in the 2 bin, and so on. Then the entire deck is sorted again on the second-least significant digit and recombined in a like manner. The process continues until the cards have been sorted on all d digits. Remarkably, at that point the cards are fully sorted on the d-digit number. Thus, only d passes through the deck are required to sort. Figure 8.3 shows how radix sort operates on a "deck" of seven 3-digit numbers.

It is essential that the digit sorts in this algorithm be stable. The sort performed by a card sorter is stable, but the operator has to be wary about not changing the order of the cards as they come out of a bin, even though all the cards in a bin have the same digit in the chosen column.

In a typical computer, which is a sequential random-access machine, radix sort is sometimes used to sort records of information that are keyed by multiple fields. For example, we might wish to sort dates by three keys: year, month, and day. We could run a sorting algorithm with a comparison function that, given two dates, compares years, and if there is a tie, compares months, and if another tie occurs, compares days. Alternatively, we could sort the information three times with a stable sort: first on day, next on month, and finally on year.

The code for radix sort is straightforward. The following procedure assumes that each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

```
RADIX-SORT(A, d)
for i ← 1 to d
     do use a stable sort to sort array A on digit i
```

Lemma 8.3
Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d(n+k))$ time.

Proof The correctness of radix sort follows by induction on the column being sorted (see Exercise 8.3-3). The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit is in the range 0 to $k-1$ (so that it can take on k possible values), and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes time $\Theta(n+k)$. There are d passes, so the total time for radix sort is $\Theta(d(n+k))$. ■

When d is constant and $k = O(n)$, radix sort runs in linear time. More generally, we have some flexibility in how to break each key into digits.

Lemma 8.4
Given n b-bit numbers and any positive integer $r \le b$, RADIX-SORT correctly sorts these numbers in $\Theta((b/r)(n+2^r))$ time.

Proof For a value $r \le b$, we view each key as having $d = \lceil b/r \rceil$ digits of r bits each. Each digit is an integer in the range 0 to $2^r - 1$, so that we can use counting sort with $k = 2^r - 1$. (For example, we can view a 32-bit word as having 4 8-bit digits, so that $b = 32$, $r = 8$, $k = 2^r - 1 = 255$, and $d = b/r = 4$.) Each pass of counting sort takes time $\Theta(n+k) = \Theta(n+2^r)$ and there are d passes, for a total running time of $\Theta(d(n+2^r)) = \Theta((b/r)(n+2^r))$. ■

For given values of n and b, we wish to choose the value of r, with $r \le b$, that minimizes the expression $(b/r)(n+2^r)$. If $b < \lfloor \lg n \rfloor$, then for any value of $r \le b$, we have that $(n+2^r) = \Theta(n)$. Thus, choosing $r = b$ yields a running time of $(b/b)(n+2^b) = \Theta(n)$, which is asymptotically optimal. If $b \ge \lfloor \lg n \rfloor$, then choosing $r = \lfloor \lg n \rfloor$ gives the best time to within a constant factor, which we can see as follows. Choosing $r = \lfloor \lg n \rfloor$ yields a running time of $\Theta(bn/\lg n)$. As we increase r above $\lfloor \lg n \rfloor$, the 2^r term in the numerator increases faster than the r term in the denominator, and so increasing r above $\lfloor \lg n \rfloor$ yields a running time of $\Omega(bn/\lg n)$. If instead we were to decrease r below $\lfloor \lg n \rfloor$, then the b/r term increases and the $n + 2^r$ term remains at $\Theta(n)$.

Is radix sort preferable to a comparison-based sorting algorithm, such as quicksort? If $b = O(\lg n)$, as is often the case, and we choose $r \approx \lg n$, then radix sort's running time is $\Theta(n)$, which appears to be better than quicksort's average-case time of $\Theta(n \lg n)$. The constant factors hidden in the Θ-notation differ, however. Although radix sort may make fewer passes than quicksort over the n keys, each pass of radix sort may take significantly longer. Which sorting algorithm is preferable depends on the characteristics of the implementations, of the underlying machine (e.g., quicksort often uses hardware caches more effectively than radix sort), and of the input data. Moreover, the version of radix sort that uses counting sort as the intermediate stable sort does not sort in place, which many of the $\Theta(n \lg n)$-time comparison sorts do. Thus, when primary memory storage is at a premium, an in-place algorithm such as quicksort may be preferable.

Exercises

8.3-1

Using Figure 8.3 as a model, illustrate the operation of RADIX-SORT on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

8.3-2

Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort? Give a simple scheme that makes any sorting algorithm stable. How much additional time and space does your scheme entail?

8.3-3

Use induction to prove that radix sort works. Where does your proof need the assumption that the intermediate sort is stable?

8.3-4

Show how to sort n integers in the range 0 to $n^2 - 1$ in $O(n)$ time.

8.3-5 ⋆

In the first card-sorting algorithm in this section, exactly how many sorting passes are needed to sort d-digit decimal numbers in the worst case? How many piles of cards would an operator need to keep track of in the worst case?