8 Sorting in Linear Time

We have now seen a handful of algorithms that can sort n numbers in $O(n \lg n)$ time. Whereas 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'll 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 on certain types of inputs. Of course, 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

A comparison sort uses 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, it performs 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. It may not inspect the values of the elements or gain order information about them in any other way.

Since we are proving a lower bound, we assume without loss of generality in this section that all the input elements are distinct. After all, a lower bound for distinct elements applies when elements may or may not be distinct. Consequently,

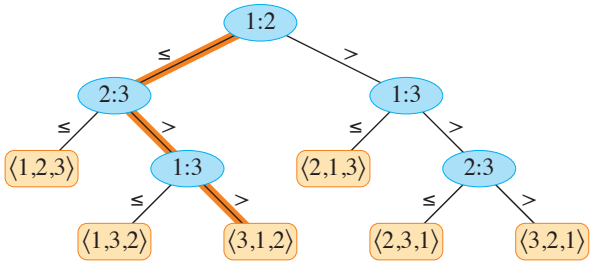

Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node (shown in blue) annotated by $i{:}j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. The highlighted path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$. Going left from the root node, labeled 1:2, indicates that $a_1 \le a_2$. Going right from the node labeled 2:3 indicates that $a_2 > a_3$. Going right from the node labeled 1:3 indicates that $a_1 > a_3$. Therefore, we have the ordering $a_3 \le a_1 \le a_2$, as indicated in the leaf labeled $\langle 3, 1, 2 \rangle$. Because the three input elements have $3! = 6$ possible permutations, the decision tree must have at least 6 leaves.

comparisons of the form $a_i = a_j$ are useless, which means that we can assume that no comparisons for exact equality occur. Moreover, 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 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

We can view comparison sorts abstractly in terms of decision trees. A ***decision tree*** is a full binary tree (each node is either a leaf or has both children) 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.

A decision tree has each internal node 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. We also annotate each leaf by a permutation $\langle \pi(1), \pi(2), \pi(n) \rangle$. (See Section C.1 for background on permutations.) Indices in the internal nodes and the leaves always refer to the original positions of the array elements at the start of the sorting algorithm. The execution of the comparison sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf. Each internal node indicates a comparison $a_i \le a_j$. The left subtree then dictates sub-

sequent comparisons once we know that $a_i \leq a_j$, and the right subtree dictates subsequent comparisons when $a_i > a_j$. Arriving at a leaf, the sorting algorithm has established the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on n elements must appear as at least one of the leaves of the decision tree for a comparison sort to be correct. Furthermore, each of these leaves must be reachable from the root by a downward path corresponding to an actual execution of the comparison sort. (We call such leaves "reachable.") Thus, we 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 simple 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 one or more leaves, we have $n! \leq l$. Since a binary tree of height h has no more than 2^h leaves, we have

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

which, by taking logarithms, implies

$$\begin{aligned} h &\geq \lg(n!) && \text{(since the lg function is monotonically increasing)} \\ &= \Omega(n \lg n) && \text{(by equation (3.28) on page 67) .} \end{aligned}$$

■

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 an n-element input sequence, and you know in advance that it is partly sorted in the following sense. Each element initially in position i such that $i \bmod 4 = 0$ is either already in its correct position, or it is one place away from its correct position. For example, you know that after sorting, the element initially in position 12 belongs in position 11, 12, or 13. You have no advance information about the other elements, in positions i where $i \bmod 4 \neq 0$. Show that an $\Omega(n \lg n)$ lower bound on comparison-based sorting still holds in this case.

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. It runs in $\Theta(n + k)$ time, so that when $k = O(n)$, counting sort runs in $\Theta(n)$ time.

Counting sort first determines, for each input element x, the number of elements less than or equal to x. It then uses this information to place element x directly into its position in the output array. For example, if 17 elements are less than or equal to x, then x belongs in output position 17. We must modify this scheme slightly to handle the situation in which several elements have the same value, since we do not want them all to end up in the same position.

The COUNTING-SORT procedure on the facing page takes as input an array $A[1:n]$, the size n of this array, and the limit k on the nonnegative integer values in A. It returns its sorted output in the array $B[1:n]$ and uses an array $C[0:k]$ for temporary working storage.

```
COUNTING-SORT(A, n, k)
 1  let B[1 : n] and C[0 : k] be new arrays
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to n
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

Figure 8.2 illustrates counting sort. After the **for** loop of lines 2–3 initializes the array C to all zeros, the **for** loop of lines 4–5 makes a pass over the array A to inspect each input element. Each time it finds an input element whose value is i, it increments $C[i]$. Thus, after line 5, $C[i]$ holds the number of input elements equal to i for each integer $i = 0, 1, \ldots, k$. Lines 7–8 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, the **for** loop of lines 11–13 makes another pass over A, but in reverse, to place each element $A[j]$ into its correct sorted position in the output array B. If all n elements are distinct, then when line 11 is first entered, 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, the loop decrements $C[A[j]]$ each time it places a value $A[j]$ into B. Decrementing $C[A[j]]$ causes the previous element in A with a value equal to $A[j]$, if one exists, to go to the position immediately before $A[j]$ in the output array B.

How much time does counting sort require? The **for** loop of lines 2–3 takes $\Theta(k)$ time, the **for** loop of lines 4–5 takes $\Theta(n)$ time, the **for** loop of lines 7–8 takes $\Theta(k)$ time, and the **for** loop of lines 11–13 takes $\Theta(n)$ time. 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 can beat 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

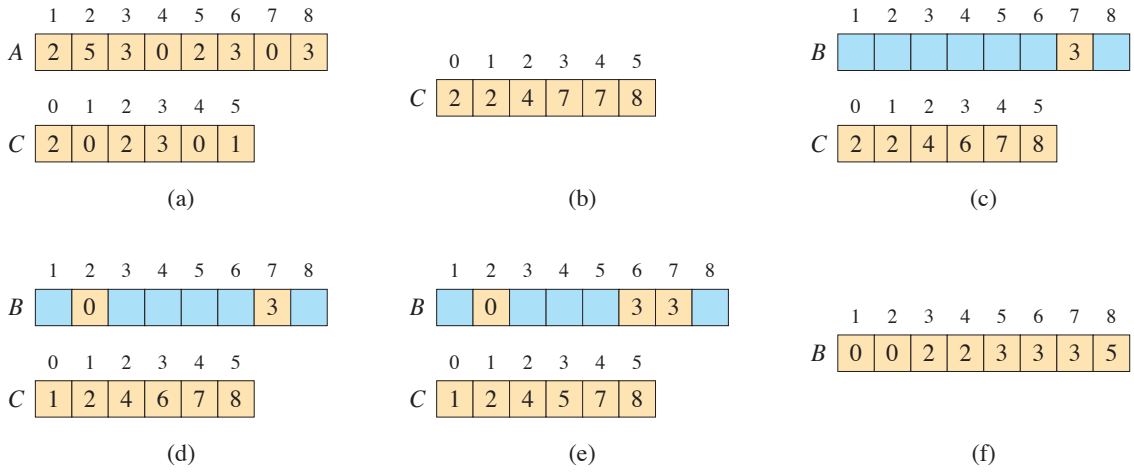

Figure 8.2 The operation of COUNTING-SORT on an input array $A[1:8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 11–13, respectively. Only the tan elements of array B have been filled in. **(f)** The final sorted output array B.

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***: elements with the same value appear in the output array in the same order as they do in the input array. That is, it breaks ties between two elements by the rule that whichever element 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, in order for radix sort to work correctly, counting sort must be stable.

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 we were to rewrite the **for** loop header in line 11 of the COUNTING-SORT as

11 **for** $j = 1$ **to** n

Show that the algorithm still works properly, but that it is not stable. Then rewrite the pseudocode for counting sort so that elements with the same value are written into the output array in order of increasing index and the algorithm is stable.

8.2-4
Prove the following loop invariant for COUNTING-SORT:

> At the start of each iteration of the **for** loop of lines 11–13, the last element in A with value i that has not yet been copied into B belongs in $B[C[i]]$.

8.2-5
Suppose that the array being sorted contains only integers in the range 0 to k and that there are no satellite data to move with those keys. Modify counting sort to use just the arrays A and C, putting the sorted result back into array A instead of into a new array B.

8.2-6
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.

8.2-7
Counting sort can also work efficiently if the input values have fractional parts, but the number of digits in the fractional part is small. Suppose that you are given n numbers in the range 0 to k, each with at most d decimal (base 10) digits to the right of the decimal point. Modify counting sort to run in $\Theta(n + 10^d k)$ 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 have 80 columns, and in each column a machine can punch a hole 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 card into one

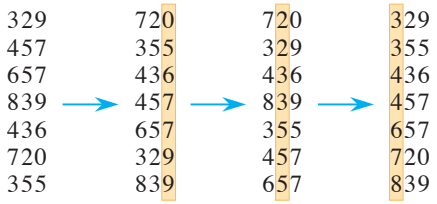

Figure 8.3 The operation of radix sort on seven 3-digit numbers. The leftmost column is the input. The remaining columns show the numbers after successive sorts on increasingly significant digit positions. Tan shading indicates the digit position sorted on to produce each list from the previous one.

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, each column uses only 10 places. (The other two places are reserved for encoding nonnumeric characters.) A d-digit number occupies 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, you might sort numbers on their *most significant* (leftmost) 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 you would have to keep track of. (See Exercise 8.3-6.)

Radix sort solves the problem of card sorting—counterintuitively—by sorting on the *least significant* digit first. The algorithm then combines the cards 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 it sorts the entire deck again on the second-least significant digit and recombines the deck 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.

In order for radix sort to work correctly, the digit sorts must be stable. The sort performed by a card sorter is stable, but the operator must be careful not to change 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, we sometimes use radix sort 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 (the "least significant" part), next on month, and finally on year.

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

```
RADIX-SORT(A, n, d)
1  for i = 1 to d
2      use a stable sort to sort array A[1:n] on digit i
```

Although the pseudocode for RADIX-SORT does not specify which stable sort to use, COUNTING-SORT is commonly used. If you use COUNTING-SORT as the stable sort, you can make RADIX-SORT a little more efficient by revising COUNTING-SORT to take a pointer to the output array as a parameter, having RADIX-SORT preallocate this array, and alternating input and output between the two arrays in successive iterations of the **for** loop in RADIX-SORT.

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 if the stable sort it uses takes $\Theta(n+k)$ 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 lies 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 $\Theta(n+k)$ time. There are d passes, and so the total time for radix sort is $\Theta(d(n+k))$. ■

When d is constant and $k = O(n)$, we can make radix sort run 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 $r \leq b$, RADIX-SORT correctly sorts these numbers in $\Theta((b/r)(n+2^r))$ time if the stable sort it uses takes $\Theta(n+k)$ time for inputs in the range 0 to k.

Proof For a value $r \leq b$, 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 four 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 $\Theta(n + k) = \Theta(n + 2^r)$ time and there are d passes, for a total running time of $\Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r))$. ■

Given n and b, what value of $r \leq b$ minimizes the expression $(b/r)(n + 2^r)$? As r decreases, the factor b/r increases, but as r increases so does 2^r. The answer depends on whether $b < \lfloor \lg n \rfloor$. If $b < \lfloor \lg n \rfloor$, then $r \leq b$ implies $(n+2^r) = \Theta(n)$. Thus, choosing $r = b$ yields a running time of $(b/b)(n + 2^b) = \Theta(n)$, which is asymptotically optimal. If $b \geq \lfloor \lg n \rfloor$, then choosing $r = \lfloor \lg n \rfloor$ gives the best running time to within a constant factor, which we can see as follows.[1] Choosing $r = \lfloor \lg n \rfloor$ yields a running time of $\Theta(bn/\lg n)$. As r increases 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 r were to decrease 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 $r \approx \lg n$, then radix sort's running time is $\Theta(n)$, which appears to be better than quicksort's expected running 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 to prefer 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 could be the better choice.

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.

---

[1] The choice of $r = \lfloor \lg n \rfloor$ assumes that $n > 1$. If $n \leq 1$, there is nothing to sort.

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 comparison sort 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
Suppose that COUNTING-SORT is used as the stable sort within RADIX-SORT. If RADIX-SORT calls COUNTING-SORT d times, then since each call of COUNTING-SORT makes two passes over the data (lines 4–5 and 11–13), altogether $2d$ passes over the data occur. Describe how to reduce the total number of passes to $d + 1$.

8.3-5
Show how to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

★ ***8.3-6***
In the first card-sorting algorithm in this section, which sorts on the most significant digit first, exactly how many sorting passes are needed to sort d-digit decimal numbers in the worst case? How many piles of cards does an operator need to keep track of in the worst case?

8.4 Bucket sort

Bucket sort assumes that the input is drawn from a uniform distribution and has an average-case running time of $O(n)$. Like counting sort, bucket sort is fast because it assumes something about the input. Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval $[0, 1)$. (See Section C.2 for a definition of a uniform distribution.)

Bucket sort divides the interval $[0, 1)$ into n equal-sized subintervals, or ***buckets***, and then distributes the n input numbers into the buckets. Since the inputs are uniformly and independently distributed over $[0, 1)$, we do not expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

The BUCKET-SORT procedure on the next page assumes that the input is an array $A[1 : n]$ and that each element $A[i]$ in the array satisfies $0 \le A[i] < 1$. The code requires an auxiliary array $B[0 : n - 1]$ of linked lists (buckets) and assumes

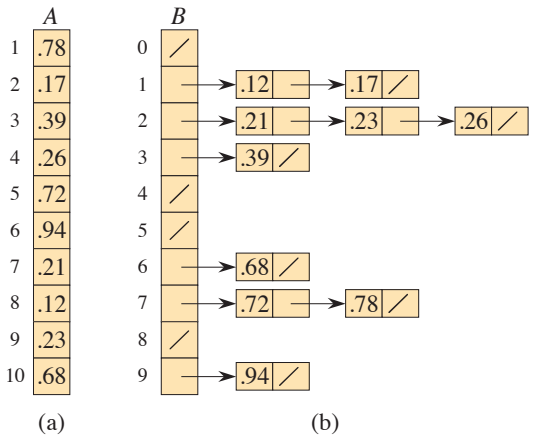

Figure 8.4 The operation of BUCKET-SORT for $n = 10$. **(a)** The input array $A[1:10]$. **(b)** The array $B[0:9]$ of sorted lists (buckets) after line 7 of the algorithm, with slashes indicating the end of each bucket. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation of the lists $B[0], B[1], \ldots, B[9]$ in order.

that there is a mechanism for maintaining such lists. (Section 10.2 describes how to implement basic operations on linked lists.) Figure 8.4 shows the operation of bucket sort on an input array of 10 numbers.

```
BUCKET-SORT(A, n)
1  let B[0 : n − 1] be a new array
2  for i = 0 to n − 1
3      make B[i] an empty list
4  for i = 1 to n
5      insert A[i] into list B[⌊n · A[i]⌋]
6  for i = 0 to n − 1
7      sort list B[i] with insertion sort
8  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
9  return the concatenated lists
```

To see that this algorithm works, consider two elements $A[i]$ and $A[j]$. Assume without loss of generality that $A[i] \le A[j]$. Since $\lfloor n \cdot A[i] \rfloor \le \lfloor n \cdot A[j] \rfloor$, either element $A[i]$ goes into the same bucket as $A[j]$ or it goes into a bucket with a lower index. If $A[i]$ and $A[j]$ go into the same bucket, then the **for** loop of lines 6–7 puts them into the proper order. If $A[i]$ and $A[j]$ go into different buckets, then line 8 puts them into the proper order. Therefore, bucket sort works correctly.

To analyze the running time, observe that, together, all lines except line 7 take $O(n)$ time in the worst case. We need to analyze the total time taken by the n calls to insertion sort in line 7.

To analyze the cost of the calls to insertion sort, let n_i be the random variable denoting the number of elements placed in bucket $B[i]$. Since insertion sort runs in quadratic time (see Section 2.2), the running time of bucket sort is

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \; . \tag{8.1}$$

We now analyze the average-case running time of bucket sort, by computing the expected value of the running time, where we take the expectation over the input distribution. Taking expectations of both sides and using linearity of expectation (equation (C.24) on page 1192), we have

$$\begin{aligned} \mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O(n_i^2)\right] \quad \text{(by linearity of expectation)} \\ &= \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}\left[n_i^2\right]\right) \quad \text{(by equation (C.25) on page 1193)} \; . \end{aligned} \tag{8.2}$$

We claim that

$$\mathrm{E}\left[n_i^2\right] = 2 - 1/n \tag{8.3}$$

for $i = 0, 1, \ldots, n-1$. It is no surprise that each bucket i has the same value of $\mathrm{E}[n_i^2]$, since each value in the input array A is equally likely to fall in any bucket.

To prove equation (8.3), view each random variable n_i as the number of successes in n Bernoulli trials (see Section C.4). Success in a trial occurs when an element goes into bucket $B[i]$, with a probability $p = 1/n$ of success and $q = 1 - 1/n$ of failure. A binomial distribution counts n_i, the number of successes, in the n trials. By equations (C.41) and (C.44) on pages 1199–1200, we have $\mathrm{E}[n_i] = np = n(1/n) = 1$ and $\mathrm{Var}[n_i] = npq = 1 - 1/n$. Equation (C.31) on page 1194 gives

$$\begin{aligned} \mathrm{E}\left[n_i^2\right] &= \mathrm{Var}[n_i] + \mathrm{E}^2[n_i] \\ &= (1 - 1/n) + 1^2 \\ &= 2 - 1/n \; , \end{aligned}$$

which proves equation (8.3). Using this expected value in equation (8.2), we get that the average-case running time for bucket sort is $\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$.

Even if the input is not drawn from a uniform distribution, bucket sort may still run in linear time. As long as the input has the property that the sum of the squares of the bucket sizes is linear in the total number of elements, equation (8.1) tells us that bucket sort runs in linear time.

Exercises

8.4-1

Using Figure 8.4 as a model, illustrate the operation of BUCKET-SORT on the array $A = \langle .79, .13, .16, .64, .39, .20, .89, .53, .71, .42 \rangle$.

8.4-2

Explain why the worst-case running time for bucket sort is $\Theta(n^2)$. What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time $O(n \lg n)$?

8.4-3

Let X be a random variable that is equal to the number of heads in two flips of a fair coin. What is $\mathrm{E}[X^2]$? What is $\mathrm{E}^2[X]$?

8.4-4

An array A of size $n > 10$ is filled in the following way. For each element $A[i]$, choose two random variables x_i and y_i uniformly and independently from $[0, 1)$. Then set

$$A[i] = \frac{\lfloor 10x_i \rfloor}{10} + \frac{y_i}{n} .$$

Modify bucket sort so that it sorts the array A in $O(n)$ expected time.

★ ***8.4-5***

You are given n points in the unit disk, $p_i = (x_i, y_i)$, such that $0 < x_i^2 + y_i^2 \leq 1$ for $i = 1, 2, \ldots, n$. Suppose that the points are uniformly distributed, that is, the probability of finding a point in any region of the disk is proportional to the area of that region. Design an algorithm with an average-case running time of $\Theta(n)$ to sort the n points by their distances $d_i = \sqrt{x_i^2 + y_i^2}$ from the origin. (*Hint:* Design the bucket sizes in BUCKET-SORT to reflect the uniform distribution of the points in the unit disk.)

★ ***8.4-6***

A ***probability distribution function*** $P(x)$ for a random variable X is defined by $P(x) = \Pr\{X \leq x\}$. Suppose that you draw a list of n random variables

$X_1, X_2, \ldots, X_n$ from a continuous probability distribution function P that is computable in $O(1)$ time (given y you can find x such that $P(x) = y$ in $O(1)$ time). Give an algorithm that sorts these numbers in linear average-case time.

Problems

8-1 Probabilistic lower bounds on comparison sorting

In this problem, you will prove a probabilistic $\Omega(n \lg n)$ lower bound on the running time of any deterministic or randomized comparison sort on n distinct input elements. You'll begin by examining a deterministic comparison sort A with decision tree T_A. Assume that every permutation of A's inputs is equally likely.

a. Suppose that each leaf of T_A is labeled with the probability that it is reached given a random input. Prove that exactly $n!$ leaves are labeled $1/n!$ and that the rest are labeled 0.

b. Let $D(T)$ denote the external path length of a decision tree T—the sum of the depths of all the leaves of T. Let T be a decision tree with $k > 1$ leaves, and let LT and RT be the left and right subtrees of T. Show that $D(T) = D(LT) + D(RT) + k$.

c. Let $d(k)$ be the minimum value of $D(T)$ over all decision trees T with $k > 1$ leaves. Show that $d(k) = \min\{d(i) + d(k-i) + k : 1 \le i \le k-1\}$. (*Hint:* Consider a decision tree T with k leaves that achieves the minimum. Let i_0 be the number of leaves in LT and $k - i_0$ the number of leaves in RT.)

d. Prove that for a given value of $k > 1$ and i in the range $1 \le i \le k-1$, the function $i \lg i + (k-i)\lg(k-i)$ is minimized at $i = k/2$. Conclude that $d(k) = \Omega(k \lg k)$.

e. Prove that $D(T_A) = \Omega(n! \lg(n!))$, and conclude that the average-case time to sort n elements is $\Omega(n \lg n)$.

Now consider a *randomized* comparison sort B. We can extend the decision-tree model to handle randomization by incorporating two kinds of nodes: ordinary comparison nodes and "randomization" nodes. A randomization node models a random choice of the form $\text{RANDOM}(1, r)$ made by algorithm B. The node has r children, each of which is equally likely to be chosen during an execution of the algorithm.

f. Show that for any randomized comparison sort B, there exists a deterministic comparison sort A whose expected number of comparisons is no more than those made by B.

8-2 Sorting in place in linear time

You have an array of n data records to sort, each with a key of 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:

1. The algorithm runs in $O(n)$ time.
2. The algorithm is stable.
3. The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.

a. Give an algorithm that satisfies criteria 1 and 2 above.

b. Give an algorithm that satisfies criteria 1 and 3 above.

c. Give an algorithm that satisfies criteria 2 and 3 above.

d. Can you use any of your sorting algorithms from parts (a)–(c) as the sorting method used in line 2 of RADIX-SORT, so that RADIX-SORT sorts n records with b-bit keys in $O(bn)$ time? Explain how or why not.

e. Suppose that the n records have keys in the range from 1 to k. Show how to modify counting sort so that it sorts the records in place in $O(n + k)$ time. You may use $O(k)$ storage outside the input array. Is your algorithm stable?

8-3 Sorting variable-length items

a. You are given an array of integers, where different integers may have different numbers of digits, but the total number of digits over *all* the integers in the array is n. Show how to sort the array in $O(n)$ time.

b. You are given an array of strings, where different strings may have different numbers of characters, but the total number of characters over all the strings is n. Show how to sort the strings in $O(n)$ time. (The desired order is the standard alphabetical order: for example, `a` < `ab` < `b`.)

8-4 Water jugs

You are given n red and n blue water jugs, all of different shapes and sizes. All the red jugs hold different amounts of water, as do all the blue jugs, and you cannot tell from the size of a jug how much water it holds. Moreover, for every jug of one color, there is a jug of the other color that holds the same amount of water.

Your task is to group the jugs into pairs of red and blue jugs that hold the same amount of water. To do so, you may perform the following operation: pick a pair

of jugs in which one is red and one is blue, fill the red jug with water, and then pour the water into the blue jug. This operation tells you whether the red jug or the blue jug can hold more water, or that they have the same volume. Assume that such a comparison takes one time unit. Your goal is to find an algorithm that makes a minimum number of comparisons to determine the grouping. Remember that you may not directly compare two red jugs or two blue jugs.

a. Describe a deterministic algorithm that uses $\Theta(n^2)$ comparisons to group the jugs into pairs.

b. Prove a lower bound of $\Omega(n \lg n)$ for the number of comparisons that an algorithm solving this problem must make.

c. Give a randomized algorithm whose expected number of comparisons is $O(n \lg n)$, and prove that this bound is correct. What is the worst-case number of comparisons for your algorithm?

8-5 Average sorting

Suppose that, instead of sorting an array, we just require that the elements increase on average. More precisely, we call an n-element array A ***k-sorted*** if, for all $i = 1, 2, \ldots, n-k$, the following holds:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k} .$$

a. What does it mean for an array to be 1-sorted?

b. Give a permutation of the numbers $1, 2, \ldots, 10$ that is 2-sorted, but not sorted.

c. Prove that an n-element array is k-sorted if and only if $A[i] \leq A[i+k]$ for all $i = 1, 2, \ldots, n-k$.

d. Give an algorithm that k-sorts an n-element array in $O(n \lg(n/k))$ time.

We can also show a lower bound on the time to produce a k-sorted array, when k is a constant.

e. Show how to sort a k-sorted array of length n in $O(n \lg k)$ time. (*Hint:* Use the solution to Exercise 6.5-11.)

f. Show that when k is a constant, k-sorting an n-element array requires $\Omega(n \lg n)$ time. (*Hint:* Use the solution to part (e) along with the lower bound on comparison sorts.)

8-6 Lower bound on merging sorted lists

The problem of merging two sorted lists arises frequently. We have seen a procedure for it as the subroutine MERGE in Section 2.3.1. In this problem, you will prove a lower bound of $2n - 1$ on the worst-case number of comparisons required to merge two sorted lists, each containing n items. First, you will show a lower bound of $2n - o(n)$ comparisons by using a decision tree.

a. Given $2n$ numbers, compute the number of possible ways to divide them into two sorted lists, each with n numbers.

b. Using a decision tree and your answer to part (a), show that any algorithm that correctly merges two sorted lists must perform at least $2n - o(n)$ comparisons.

Now you will show a slightly tighter $2n - 1$ bound.

c. Show that if two elements are consecutive in the sorted order and from different lists, then they must be compared.

d. Use your answer to part (c) to show a lower bound of $2n - 1$ comparisons for merging two sorted lists.

8-7 The 0-1 sorting lemma and columnsort

A ***compare-exchange*** operation on two array elements $A[i]$ and $A[j]$, where $i < j$, has the form

```
COMPARE-EXCHANGE(A, i, j)
1  if A[i] > A[j]
2      exchange A[i] with A[j]
```

After the compare-exchange operation, we know that $A[i] \le A[j]$.

An ***oblivious compare-exchange algorithm*** operates solely by a sequence of prespecified compare-exchange operations. The indices of the positions compared in the sequence must be determined in advance, and although they can depend on the number of elements being sorted, they cannot depend on the values being sorted, nor can they depend on the result of any prior compare-exchange operation. For example, the COMPARE-EXCHANGE-INSERTION-SORT procedure on the facing page shows a variation of insertion sort as an oblivious compare-exchange algorithm. (Unlike the INSERTION-SORT procedure on page 19, the oblivious version runs in $\Theta(n^2)$ time in all cases.)

The ***0-1 sorting lemma*** provides a powerful way to prove that an oblivious compare-exchange algorithm produces a sorted result. It states that if an oblivious compare-exchange algorithm correctly sorts all input sequences consisting of only 0s and 1s, then it correctly sorts all inputs containing arbitrary values.

```
COMPARE-EXCHANGE-INSERTION-SORT(A, n)
1  for i = 2 to n
2      for j = i − 1 downto 1
3          COMPARE-EXCHANGE(A, j, j + 1)
```

You will prove the 0-1 sorting lemma by proving its contrapositive: if an oblivious compare-exchange algorithm fails to sort an input containing arbitrary values, then it fails to sort some 0-1 input. Assume that an oblivious compare-exchange algorithm X fails to correctly sort the array $A[1:n]$. Let $A[p]$ be the smallest value in A that algorithm X puts into the wrong location, and let $A[q]$ be the value that algorithm X moves to the location into which $A[p]$ should have gone. Define an array $B[1:n]$ of 0s and 1s as follows:

$$B[i] = \begin{cases} 0 & \text{if } A[i] \le A[p] \,, \\ 1 & \text{if } A[i] > A[p] \,. \end{cases}$$

a. Argue that $A[q] > A[p]$, so that $B[p] = 0$ and $B[q] = 1$.

b. To complete the proof of the 0-1 sorting lemma, prove that algorithm X fails to sort array B correctly.

Now you will use the 0-1 sorting lemma to prove that a particular sorting algorithm works correctly. The algorithm, ***columnsort***, works on a rectangular array of n elements. The array has r rows and s columns (so that $n = rs$), subject to three restrictions:

- r must be even,
- s must be a divisor of r, and
- $r \ge 2s^2$.

When columnsort completes, the array is sorted in ***column-major order***: reading down each column in turn, from left to right, the elements monotonically increase.

Columnsort operates in eight steps, regardless of the value of n. The odd steps are all the same: sort each column individually. Each even step is a fixed permutation. Here are the steps:

1. Sort each column.
2. Transpose the array, but reshape it back to r rows and s columns. In other words, turn the leftmost column into the top r/s rows, in order; turn the next column into the next r/s rows, in order; and so on.

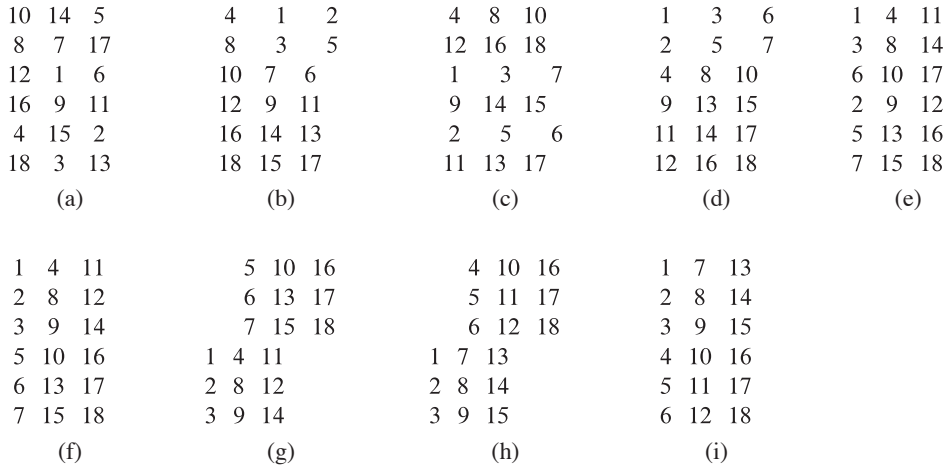

Figure 8.5 The steps of columnsort. **(a)** The input array with 6 rows and 3 columns. (This example does not obey the $r \geq 2s^2$ requirement, but it works.) **(b)** After sorting each column in step 1. **(c)** After transposing and reshaping in step 2. **(d)** After sorting each column in step 3. **(e)** After performing step 4, which inverts the permutation from step 2. **(f)** After sorting each column in step 5. **(g)** After shifting by half a column in step 6. **(h)** After sorting each column in step 7. **(i)** After performing step 8, which inverts the permutation from step 6. Steps 6–8 sort the bottom half of each column with the top half of the next column. After step 8, the array is sorted in column-major order.

3. Sort each column.
4. Perform the inverse of the permutation performed in step 2.
5. Sort each column.
6. Shift the top half of each column into the bottom half of the same column, and shift the bottom half of each column into the top half of the next column to the right. Leave the top half of the leftmost column empty. Shift the bottom half of the last column into the top half of a new rightmost column, and leave the bottom half of this new column empty.
7. Sort each column.
8. Perform the inverse of the permutation performed in step 6.

You can think of steps 6–8 as a single step that sorts the bottom half of each column and the top half of the next column. Figure 8.5 shows an example of the steps of columnsort with $r = 6$ and $s = 3$. (Even though this example violates the requirement that $r \geq 2s^2$, it happens to work.)

c. Argue that we can treat columnsort as an oblivious compare-exchange algorithm, even if we do not know what sorting method the odd steps use.

Although it might seem hard to believe that columnsort actually sorts, you will use the 0-1 sorting lemma to prove that it does. The 0-1 sorting lemma applies because we can treat columnsort as an oblivious compare-exchange algorithm. A couple of definitions will help you apply the 0-1 sorting lemma. We say that an area of an array is ***clean*** if we know that it contains either all 0s or all 1s or if it is empty. Otherwise, the area might contain mixed 0s and 1s, and it is ***dirty***. From here on, assume that the input array contains only 0s and 1s, and that we can treat it as an array with r rows and s columns.

d. Prove that after steps 1–3, the array consists of clean rows of 0s at the top, clean rows of 1s at the bottom, and at most s dirty rows between them. (One of the clean rows could be empty.)

e. Prove that after step 4, the array, read in column-major order, starts with a clean area of 0s, ends with a clean area of 1s, and has a dirty area of at most s^2 elements in the middle. (Again, one of the clean areas could be empty.)

f. Prove that steps 5–8 produce a fully sorted 0-1 output. Conclude that columnsort correctly sorts all inputs containing arbitrary values.

g. Now suppose that s does not divide r. Prove that after steps 1–3, the array consists of clean rows of 0s at the top, clean rows of 1s at the bottom, and at most $2s - 1$ dirty rows between them. (Once again, one of the clean areas could be empty.) How large must r be, compared with s, for columnsort to correctly sort when s does not divide r?

h. Suggest a simple change to step 1 that allows us to maintain the requirement that $r \geq 2s^2$ even when s does not divide r, and prove that with your change, columnsort correctly sorts.

Chapter notes

The decision-tree model for studying comparison sorts was introduced by Ford and Johnson [150]. Knuth's comprehensive treatise on sorting [261] covers many variations on the sorting problem, including the information-theoretic lower bound on the complexity of sorting given here. Ben-Or [46] studied lower bounds for sorting using generalizations of the decision-tree model.

Knuth credits H. H. Seward with inventing counting sort in 1954, as well as with the idea of combining counting sort with radix sort. Radix sorting starting with the least significant digit appears to be a folk algorithm widely used by operators of

mechanical card-sorting machines. According to Knuth, the first published reference to the method is a 1929 document by L. J. Comrie describing punched-card equipment. Bucket sorting has been in use since 1956, when the basic idea was proposed by Isaac and Singleton [235].

Munro and Raman [338] give a stable sorting algorithm that performs $O(n^{1+\epsilon})$ comparisons in the worst case, where $0 < \epsilon \leq 1$ is any fixed constant. Although any of the $O(n \lg n)$-time algorithms make fewer comparisons, the algorithm by Munro and Raman moves data only $O(n)$ times and operates in place.

The case of sorting n b-bit integers in $o(n \lg n)$ time has been considered by many researchers. Several positive results have been obtained, each under slightly different assumptions about the model of computation and the restrictions placed on the algorithm. All the results assume that the computer memory is divided into addressable b-bit words. Fredman and Willard [157] introduced the fusion tree data structure and used it to sort n integers in $O(n \lg n / \lg \lg n)$ time. This bound was later improved to $O(n \sqrt{\lg n})$ time by Andersson [17]. These algorithms require the use of multiplication and several precomputed constants. Andersson, Hagerup, Nilsson, and Raman [18] have shown how to sort n integers in $O(n \lg \lg n)$ time without using multiplication, but their method requires storage that can be unbounded in terms of n. Using multiplicative hashing, we can reduce the storage needed to $O(n)$, but then the $O(n \lg \lg n)$ worst-case bound on the running time becomes an expected-time bound. Generalizing the exponential search trees of Andersson [17], Thorup [434] gave an $O(n(\lg \lg n)^2)$-time sorting algorithm that does not use multiplication or randomization, and it uses linear space. Combining these techniques with some new ideas, Han [207] improved the bound for sorting to $O(n \lg \lg n \lg \lg \lg n)$ time. Although these algorithms are important theoretical breakthroughs, they are all fairly complicated and at the present time seem unlikely to compete with existing sorting algorithms in practice.

The columnsort algorithm in Problem 8-7 is by Leighton [286].