running out of water. (Because North Dakota is relatively flat, the professor does not have to worry about drinking water at a greater rate on uphill sections than on flat or downhill sections.) The professor will start in Grand Forks with two full liters of water. The professor has an official North Dakota state map, which shows all the places along U.S. 2 to refill water and the distances between these locations.

The professor's goal is to minimize the number of water stops along the route across the state. Give an efficient method by which the professor can determine which water stops to make. Prove that your strategy yields an optimal solution, and give its running time.

15.2-5

Describe an efficient algorithm that, given a set $\{x_1, x_2, \ldots, x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

★ ***15.2-6***

Show how to solve the fractional knapsack problem in $O(n)$ time.

15.2-7

You are given two sets A and B, each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the ith element of set A, and let b_i be the ith element of set B. You then receive a payoff of $\prod_{i=1}^{n} a_i^{b_i}$. Give an algorithm that maximizes your payoff. Prove that your algorithm maximizes the payoff, and state its running time, omitting the time for reordering the sets.

15.3 Huffman codes

Huffman codes compress data well: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed. The data arrive as a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (its frequency) to build up an optimal way of representing each character as a binary string.

Suppose that you have a 100,000-character data file that you wish to store compactly and you know that the 6 distinct characters in the file occur with the frequencies given by Figure 15.4. The character `a` occurs 45,000 times, the character `b` occurs 13,000 times, and so on.

You have many options for how to represent such a file of information. Here, we consider the problem of designing a ***binary character code*** (or ***code*** for short)

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 15.4 A character-coding problem. A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated. With each character represented by a 3-bit codeword, encoding the file requires 300,000 bits. With the variable-length code shown, the encoding requires only 224,000 bits.

in which each character is represented by a unique binary string, which we call a ***codeword***. If you use a ***fixed-length code***, you need $\lceil \lg n \rceil$ bits to represent $n \geq 2$ characters. For 6 characters, therefore, you need 3 bits: a = 000, b = 001, c = 010, d = 011, e = 100, and f = 101. This method requires 300,000 bits to encode the entire file. Can you do better?

A ***variable-length code*** can do considerably better than a fixed-length code. The idea is simple: give frequent characters short codewords and infrequent characters long codewords. Figure 15.4 shows such a code. Here, the 1-bit string 0 represents a, and the 4-bit string 1100 represents f. This code requires

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

to represent the file, a savings of approximately 25%. In fact, this is an optimal character code for this file, as we shall see.

Prefix-free codes

We consider here only codes in which no codeword is also a prefix of some other codeword. Such codes are called ***prefix-free codes***. Although we won't prove it here, a prefix-free code can always achieve the optimal data compression among any character code, and so we suffer no loss of generality by restricting our attention to prefix-free codes.

Encoding is always simple for any binary character code: just concatenate the codewords representing each character of the file. For example, with the variable-length prefix-free code of Figure 15.4, the 4-character file face has the encoding $1100 \cdot 0 \cdot 100 \cdot 1101 = 110001001101$, where "$\cdot$" denotes concatenation.

Prefix-free codes are desirable because they simplify decoding. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. You can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file. In our example, the string 100011001101 parses uniquely as $100 \cdot 0 \cdot 1100 \cdot 1101$, which decodes to cafe.

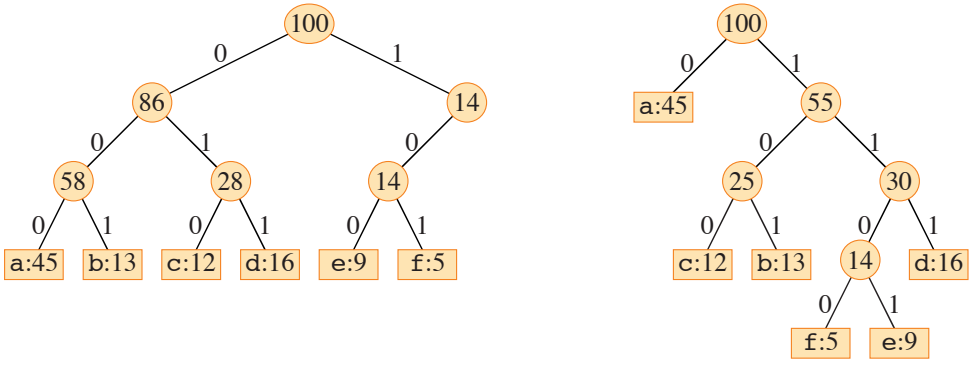

Figure 15.5 Trees corresponding to the coding schemes in Figure 15.4. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. All frequencies are in thousands. **(a)** The tree corresponding to the fixed-length code a = 000, b = 001, c = 010, d = 011, e = 100, f = 101. **(b)** The tree corresponding to the optimal prefix-free code a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100.

The decoding process needs a convenient representation for the prefix-free code so that you can easily pick off the initial codeword. A binary tree whose leaves are the given characters provides one such representation. Interpret the binary codeword for a character as the simple path from the root to that character, where 0 means "go to the left child" and 1 means "go to the right child." Figure 15.5 shows the trees for the two codes of our example. Note that these are not binary search trees, since the leaves need not appear in sorted order and internal nodes do not contain character keys.

An optimal code for a file is always represented by a *full* binary tree, in which every nonleaf node has two children (see Exercise 15.3-2). The fixed-length code in our example is not optimal since its tree, shown in Figure 15.5(a), is not a full binary tree: it contains codewords beginning with 10, but none beginning with 11. Since we can now restrict our attention to full binary trees, we can say that if C is the alphabet from which the characters are drawn and all character frequencies are positive, then the tree for an optimal prefix-free code has exactly $|C|$ leaves, one for each letter of the alphabet, and exactly $|C| - 1$ internal nodes (see Exercise B.5-3 on page 1175).

Given a tree T corresponding to a prefix-free code, we can compute the number of bits required to encode a file. For each character c in the alphabet C, let the attribute $c.freq$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree. Note that $d_T(c)$ is also the length of the codeword for character c. The number of bits required to encode a file is thus

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c) \,, \tag{15.4}$$

which we define as the ***cost*** of the tree T.

Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix-free code, called a ***Huffman code*** in his honor. In line with our observations in Section 15.2, its proof of correctness relies on the greedy-choice property and optimal substructure. Rather than demonstrating that these properties hold and then developing pseudocode, we present the pseudocode first. Doing so will help clarify how the algorithm makes greedy choices.

The procedure HUFFMAN assumes that C is a set of n characters and that each character $c \in C$ is an object with an attribute $c.freq$ giving its frequency. The algorithm builds the tree T corresponding to an optimal code in a bottom-up manner. It begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ "merging" operations to create the final tree. The algorithm uses a min-priority queue Q, keyed on the *freq* attribute, to identify the two least-frequent objects to merge together. The result of merging two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

```
HUFFMAN(C)
 1  n = |C|
 2  Q = C
 3  for i = 1 to n − 1
 4       allocate a new node z
 5       x = EXTRACT-MIN(Q)
 6       y = EXTRACT-MIN(Q)
 7       z.left = x
 8       z.right = y
 9       z.freq = x.freq + y.freq
10       INSERT(Q, z)
11  return EXTRACT-MIN(Q)      // the root of the tree is the only node left
```

For our example, Huffman's algorithm proceeds as shown in Figure 15.6. Since the alphabet contains 6 letters, the initial queue size is $n = 6$, and 5 merge steps build the tree. The final tree represents the optimal prefix-free code. The codeword for a letter is the sequence of edge labels on the simple path from the root to the letter.

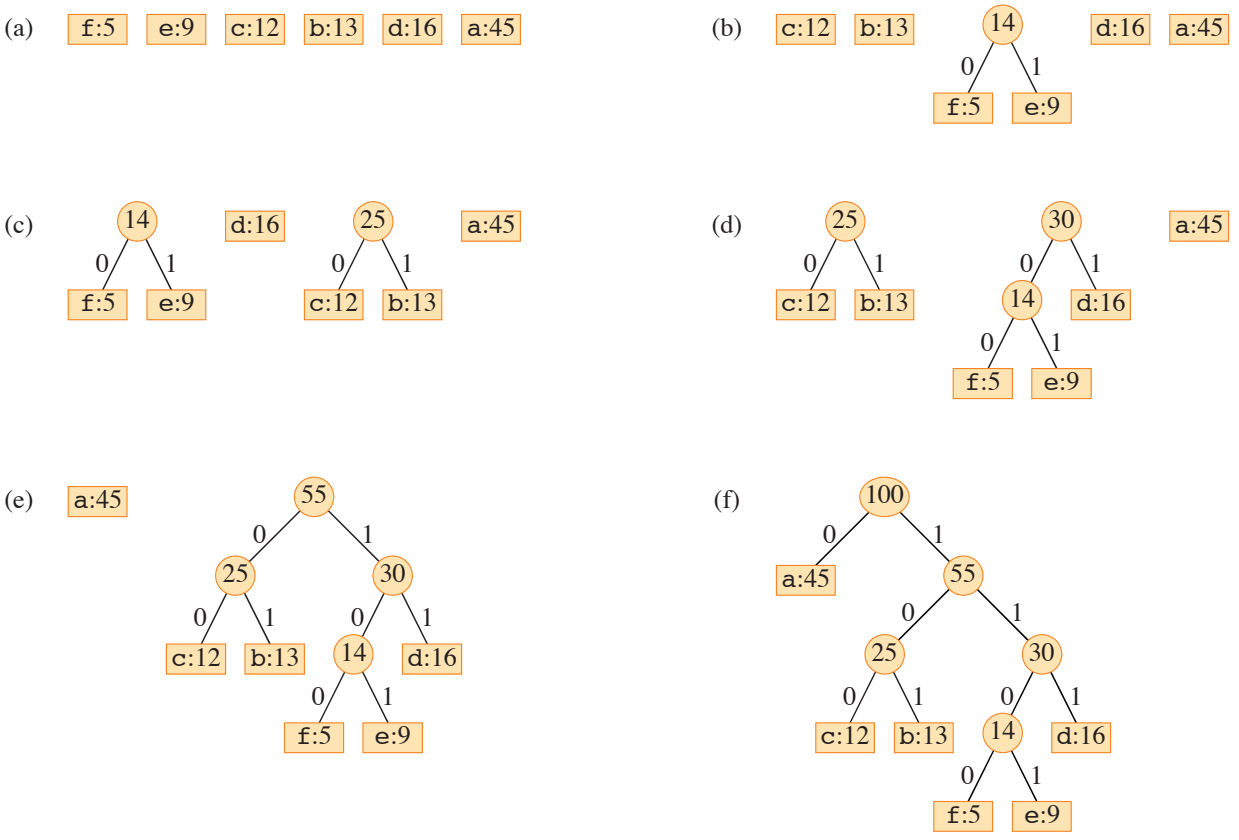

Figure 15.6 The steps of Huffman's algorithm for the frequencies given in Figure 15.4. Each part shows the contents of the queue sorted into increasing order by frequency. Each step merges the two trees with the lowest frequencies. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of their children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child. The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter. **(a)** The initial set of $n = 6$ nodes, one for each letter. **(b)–(e)** Intermediate stages. **(f)** The final tree.

The HUFFMAN procedure works as follows. Line 2 initializes the min-priority queue Q with the characters in C. The **for** loop in lines 3–10 repeatedly extracts the two nodes x and y of lowest frequency from the queue and replaces them in the queue with a new node z representing their merger. The frequency of z is computed as the sum of the frequencies of x and y in line 9. The node z has x as its left child and y as its right child. (This order is arbitrary. Switching the left and right child of any node yields a different code of the same cost.) After $n - 1$ mergers, line 11 returns the one node left in the queue, which is the root of the code tree.

The algorithm produces the same result without the variables x and y, assigning the values returned by the EXTRACT-MIN calls directly to $z.left$ and $z.right$ in lines 7 and 8, and changing line 9 to $z.freq = z.left.freq + z.right.freq$. We'll use the node names x and y in the proof of correctness, however, so we leave them in.

The running time of Huffman's algorithm depends on how the min-priority queue Q is implemented. Let's assume that it's implemented as a binary min-heap (see Chapter 6). For a set C of n characters, the BUILD-MIN-HEAP procedure discussed in Section 6.3 can initialize Q in line 2 in $O(n)$ time. The **for** loop in lines 3–10 executes exactly $n - 1$ times, and since each heap operation runs in $O(\lg n)$ time, the loop contributes $O(n \lg n)$ to the running time. Thus, the total running time of HUFFMAN on a set of n characters is $O(n \lg n)$.

Correctness of Huffman's algorithm

To prove that the greedy algorithm HUFFMAN is correct, we'll show that the problem of determining an optimal prefix-free code exhibits the greedy-choice and optimal-substructure properties. The next lemma shows that the greedy-choice property holds.

Lemma 15.2 (Optimal prefix-free codes have the greedy-choice property)
Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix-free code for C in which the codewords for x and y have the same length and differ only in the last bit.

Proof The idea of the proof is to take the tree T representing an arbitrary optimal prefix-free code and modify it to make a tree representing another optimal prefix-free code such that the characters x and y appear as sibling leaves of maximum depth in the new tree. In such a tree, the codewords for x and y have the same length and differ only in the last bit.

Let a and b be any two characters that are sibling leaves of maximum depth in T. Without loss of generality, assume that $a.freq \le b.freq$ and $x.freq \le y.freq$. Since $x.freq$ and $y.freq$ are the two lowest leaf frequencies, in order, and $a.freq$ and $b.freq$ are two arbitrary frequencies, in order, we have $x.freq \le a.freq$ and $y.freq \le b.freq$.

In the remainder of the proof, it is possible that we could have $x.freq = a.freq$ or $y.freq = b.freq$, but $x.freq = b.freq$ implies that $a.freq = b.freq = x.freq = y.freq$ (see Exercise 15.3-1), and the lemma would be trivially true. Therefore, assume that $x.freq \ne b.freq$, which means that $x \ne b$.

As Figure 15.7 shows, imagine exchanging the positions in T of a and x to produce a tree T', and then exchanging the positions in T' of b and y to produce a

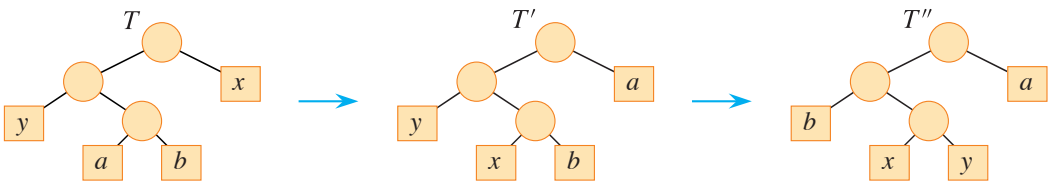

Figure 15.7 An illustration of the key step in the proof of Lemma 15.2. In the optimal tree T, leaves a and b are two siblings of maximum depth. Leaves x and y are the two characters with the lowest frequencies. They appear in arbitrary positions in T. Assuming that $x \neq b$, swapping leaves a and x produces tree T', and then swapping leaves b and y produces tree T''. Since each swap does not increase the cost, the resulting tree T'' is also an optimal tree.

tree T'' in which x and y are sibling leaves of maximum depth. (Note that if $x = b$ but $y \neq a$, then tree T'' does not have x and y as sibling leaves of maximum depth. Because we assume that $x \neq b$, this situation cannot occur.) By equation (15.4), the difference in cost between T and T' is

$$\begin{aligned}
&B(T) - B(T') \\
&\quad = \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&\quad = x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
&\quad = x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\
&\quad = (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\quad \geq 0 \,,
\end{aligned}$$

because both $a.freq - x.freq$ and $d_T(a) - d_T(x)$ are nonnegative. More specifically, $a.freq - x.freq$ is nonnegative because x is a minimum-frequency leaf, and $d_T(a) - d_T(x)$ is nonnegative because a is a leaf of maximum depth in T. Similarly, exchanging y and b does not increase the cost, and so $B(T') - B(T'')$ is nonnegative. Therefore, $B(T'') \leq B(T') \leq B(T)$, and since T is optimal, we have $B(T) \leq B(T'')$, which implies $B(T'') = B(T)$. Thus, T'' is an optimal tree in which x and y appear as sibling leaves of maximum depth, from which the lemma follows. ■

Lemma 15.2 implies that the process of building up an optimal tree by mergers can, without loss of generality, begin with the greedy choice of merging together those two characters of lowest frequency. Why is this a greedy choice? We can view the cost of a single merger as being the sum of the frequencies of the two items being merged. Exercise 15.3-4 shows that the total cost of the tree constructed equals the sum of the costs of its mergers. Of all possible mergers at each step, HUFFMAN chooses the one that incurs the least cost.

The next lemma shows that the problem of constructing optimal prefix-free codes has the optimal-substructure property.

Lemma 15.3 (Optimal prefix-free codes have the optimal-substructure property)
Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with the characters x and y removed and a new character z added, so that $C' = (C - \{x, y\}) \cup \{z\}$. Define $freq$ for all characters in C' with the same values as in C, along with $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix-free code for alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix-free code for the alphabet C.

Proof We first show how to express the cost $B(T)$ of tree T in terms of the cost $B(T')$ of tree T', by considering the component costs in equation (15.4). For each character $c \in C - \{x, y\}$, we have that $d_T(c) = d_{T'}(c)$, and hence $c.freq \cdot d_T(c) = c.freq \cdot d_{T'}(c)$. Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$\begin{aligned} x.freq \cdot d_T(x) + y.freq \cdot d_T(y) &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq) , \end{aligned}$$

from which we conclude that

$$B(T) = B(T') + x.freq + y.freq$$

or, equivalently,

$$B(T') = B(T) - x.freq - y.freq .$$

We now prove the lemma by contradiction. Suppose that T does not represent an optimal prefix-free code for C. Then there exists an optimal tree T'' such that $B(T'') < B(T)$. Without loss of generality (by Lemma 15.2), T'' has x and y as siblings. Let T''' be the tree T'' with the common parent of x and y replaced by a leaf z with frequency $z.freq = x.freq + y.freq$. Then

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T') , \end{aligned}$$

yielding a contradiction to the assumption that T' represents an optimal prefix-free code for C'. Thus, T must represent an optimal prefix-free code for the alphabet C. ■

Theorem 15.4
Procedure HUFFMAN produces an optimal prefix-free code.