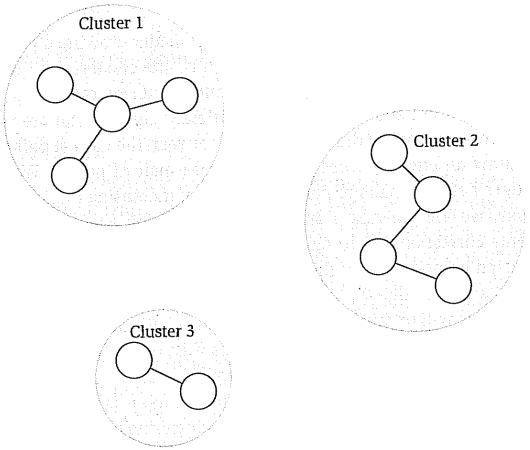

Figure 4.14 An example of single-linkage clustering with $k = 3$ clusters. The clusters are formed by adding edges between points in order of increasing distance.

(4.26) *The components $C_1, C_2, \ldots, C_k$ formed by deleting the $k-1$ most expensive edges of the minimum spanning tree T constitute a k-clustering of maximum spacing.*

Proof. Let $\mathcal{C}$ denote the clustering $C_1, C_2, \ldots, C_k$. The spacing of $\mathcal{C}$ is precisely the length d^* of the $(k-1)^{\text{st}}$ most expensive edge in the minimum spanning tree; this is the length of the edge that Kruskal's Algorithm would have added next, at the moment we stopped it.

Now consider some other k-clustering $\mathcal{C}'$, which partitions U into non-empty sets $C'_1, C'_2, \ldots, C'_k$. We must show that the spacing of $\mathcal{C}'$ is at most d^*.

Since the two clusterings $\mathcal{C}$ and $\mathcal{C}'$ are not the same, it must be that one of our clusters C_r is not a subset of any of the k sets C'_s in $\mathcal{C}'$. Hence there are points $p_i, p_j \in C_r$ that belong to different clusters in $\mathcal{C}'$—say, $p_i \in C'_s$ and $p_j \in C'_t \neq C'_s$.

Now consider the picture in Figure 4.15. Since p_i and p_j belong to the same component C_r, it must be that Kruskal's Algorithm added all the edges of a p_i-p_j path P before we stopped it. In particular, this means that each edge on

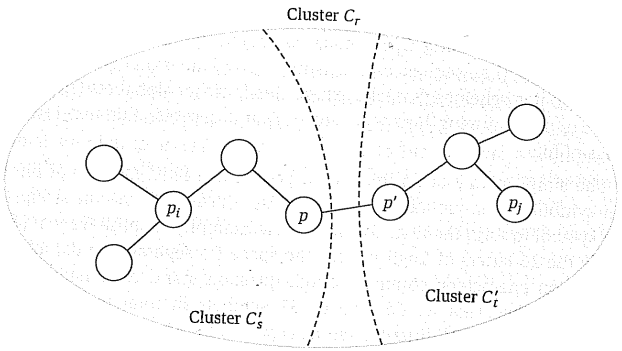

Figure 4.15 An illustration of the proof of (4.26), showing that the spacing of any other clustering can be no larger than that of the clustering found by the single-linkage algorithm.

P has length at most d^*. Now, we know that $p_i \in C'_s$ but $p_j \notin C'_s$; so let p' be the first node on P that does not belong to C'_s, and let p be the node on P that comes just before p'. We have just argued that $d(p, p') \leq d^*$, since the edge (p, p') was added by Kruskal's Algorithm. But p and p' belong to different sets in the clustering $\mathcal{C}'$, and hence the spacing of $\mathcal{C}'$ is at most $d(p, p') \leq d^*$. This completes the proof. ■

4.8 Huffman Codes and Data Compression

In the Shortest-Path and Minimum Spanning Tree Problems, we've seen how greedy algorithms can be used to commit to certain parts of a solution (edges in a graph, in these cases), based entirely on relatively short-sighted considerations. We now consider a problem in which this style of "committing" is carried out in an even looser sense: a greedy rule is used, essentially, to shrink the size of the problem instance, so that an equivalent smaller problem can then be solved by recursion. The greedy operation here is proved to be "safe," in the sense that solving the smaller instance still leads to an optimal solution for the original instance, but the global consequences of the initial greedy decision do not become fully apparent until the full recursion is complete.

The problem itself is one of the basic questions in the area of *data compression*, an area that forms part of the foundations for digital communication.

The Problem

Encoding Symbols Using Bits Since computers ultimately operate on sequences of *bits* (i.e., sequences consisting only of the symbols 0 and 1), one needs encoding schemes that take text written in richer alphabets (such as the alphabets underpinning human languages) and converts this text into long strings of bits.

The simplest way to do this would be to use a fixed number of bits for each symbol in the alphabet, and then just concatenate the bit strings for each symbol to form the text. To take a basic example, suppose we wanted to encode the 26 letters of English, plus the space (to separate words) and five punctuation characters: comma, period, question mark, exclamation point, and apostrophe. This would give us 32 symbols in total to be encoded. Now, you can form 2^b different sequences out of b bits, and so if we use 5 bits per symbol, then we can encode $2^5 = 32$ symbols—just enough for our purposes. So, for example, we could let the bit string 00000 represent a, the bit string 00001 represent b, and so forth up to 11111, which could represent the apostrophe. Note that the mapping of bit strings to symbols is arbitrary; the point is simply that five bits per symbol is sufficient. In fact, encoding schemes like ASCII work precisely this way, except that they use a larger number of bits per symbol so as to handle larger character sets, including capital letters, parentheses, and all those other special symbols you see on a typewriter or computer keyboard.

Let's think about our bare-bones example with just 32 symbols. Is there anything more we could ask for from an encoding scheme? We couldn't ask to encode each symbol using just four bits, since 2^4 is only 16—not enough for the number of symbols we have. Nevertheless, it's not clear that over large stretches of text, we really need to be spending an *average* of five bits per symbol. If we think about it, the letters in most human alphabets do not get used equally frequently. In English, for example, the letters e, t, a, o, i, and n get used much more frequently than q, j, x, and z (by more than an order of magnitude). So it's really a tremendous waste to translate them all into the same number of bits; instead we could use a small number of bits for the frequent letters, and a larger number of bits for the less frequent ones, and hope to end up using fewer than five bits per letter when we average over a long string of typical text.

This issue of reducing the average number of bits per letter is a fundamental problem in the area of *data compression*. When large files need to be shipped across communication networks, or stored on hard disks, it's important to represent them as compactly as possible, subject to the requirement that a subsequent reader of the file should be able to correctly reconstruct it. A huge amount of research is devoted to the design of *compression algorithms* that can take files as input and reduce their space through efficient encoding schemes.

We now describe one of the fundamental ways of formulating this issue, building up to the question of how we might construct the *optimal* way to take advantage of the nonuniform frequencies of the letters. In one sense, such an optimal solution is a very appealing answer to the problem of compressing data: it squeezes all the available gains out of nonuniformities in the frequencies. At the end of the section, we will discuss how one can make further progress in compression, taking advantage of features other than nonuniform frequencies.

Variable-Length Encoding Schemes Before the Internet, before the digital computer, before the radio and telephone, there was the telegraph. Communicating by telegraph was a lot faster than the contemporary alternatives of hand-delivering messages by railroad or on horseback. But telegraphs were only capable of transmitting pulses down a wire, and so if you wanted to send a message, you needed a way to encode the text of your message as a sequence of pulses.

To deal with this issue, the pioneer of telegraphic communication, Samuel Morse, developed *Morse code*, translating each letter into a sequence of *dots* (short pulses) and *dashes* (long pulses). For our purposes, we can think of dots and dashes as zeros and ones, and so this is simply a mapping of symbols into bit strings, just as in ASCII. Morse understood the point that one could communicate more efficiently by encoding frequent letters with short strings, and so this is the approach he took. (He consulted local printing presses to get frequency estimates for the letters in English.) Thus, Morse code maps e to 0 (a single dot), t to 1 (a single dash), a to 01 (dot-dash), and in general maps more frequent letters to shorter bit strings.

In fact, Morse code uses such short strings for the letters that the encoding of words becomes ambiguous. For example, just using what we know about the encoding of e, t, and a, we see that the string 0101 could correspond to any of the sequences of letters *eta*, *aa*, *etet*, or *aet*. (There are other possibilities as well, involving other letters.) To deal with this ambiguity, Morse code transmissions involve short pauses between letters (so the encoding of *aa* would actually be dot-dash-pause-dot-dash-pause). This is a reasonable solution—using very short bit strings and then introducing pauses—but it means that we haven't actually encoded the letters using just 0 and 1; we've actually encoded it using a three-letter alphabet of 0, 1, and "pause." Thus, if we really needed to encode everything using only the bits 0 and 1, there would need to be some further encoding in which the pause got mapped to bits.

Prefix Codes The ambiguity problem in Morse code arises because there exist pairs of letters where the bit string that encodes one letter is a *prefix* of the bit string that encodes another. To eliminate this problem, and hence to obtain an encoding scheme that has a well-defined interpretation for every sequence of bits, it is enough to map letters to bit strings in such a way that no encoding is a prefix of any other.

Concretely, we say that a *prefix code* for a set S of letters is a function γ that maps each letter $x \in S$ to some sequence of zeros and ones, in such a way that for distinct $x, y \in S$, the sequence $\gamma(x)$ is not a prefix of the sequence $\gamma(y)$.

Now suppose we have a text consisting of a sequence of letters $x_1 x_2 x_3 \cdots x_n$. We can convert this to a sequence of bits by simply encoding each letter as a bit sequence using γ and then concatenating all these bit sequences together: $\gamma(x_1)\gamma(x_2)\cdots\gamma(x_n)$. If we then hand this message to a recipient who knows the function γ, they will be able to reconstruct the text according to the following rule.

- Scan the bit sequence from left to right.
- As soon as you've seen enough bits to match the encoding of some letter, output this as the first letter of the text. This must be the correct first letter, since no shorter or longer prefix of the bit sequence could encode any other letter.
- Now delete the corresponding set of bits from the front of the message and iterate.

In this way, the recipient can produce the correct set of letters without our having to resort to artificial devices like pauses to separate the letters.

For example, suppose we are trying to encode the set of five letters $S = \{a, b, c, d, e\}$. The encoding γ_1 specified by

$$\gamma_1(a) = 11$$
$$\gamma_1(b) = 01$$
$$\gamma_1(c) = 001$$
$$\gamma_1(d) = 10$$
$$\gamma_1(e) = 000$$

is a prefix code, since we can check that no encoding is a prefix of any other. Now, for example, the string *cecab* would be encoded as 0010000011101. A recipient of this message, knowing γ_1, would begin reading from left to right. Neither 0 nor 00 encodes a letter, but 001 does, so the recipient concludes that the first letter is c. This is a safe decision, since no longer sequence of bits beginning with 001 could encode a different letter. The recipient now iterates

on the rest of the message, 0000011101; next they will conclude that the second letter is e, encoded as 000.

Optimal Prefix Codes We've been doing all this because some letters are more frequent than others, and we want to take advantage of the fact that more frequent letters can have shorter encodings. To make this objective precise, we now introduce some notation to express the frequencies of letters.

Suppose that for each letter $x \in S$, there is a frequency f_x, representing the fraction of letters in the text that are equal to x. In other words, assuming there are n letters total, nf_x of these letters are equal to x. We notice that the frequencies sum to 1; that is, $\sum_{x \in S} f_x = 1$.

Now, if we use a prefix code γ to encode the given text, what is the total length of our encoding? This is simply the sum, over all letters $x \in S$, of the number of times x occurs times the length of the bit string $\gamma(x)$ used to encode x. Using $|\gamma(x)|$ to denote the length $\gamma(x)$, we can write this as

$$\text{encoding length} = \sum_{x \in S} nf_x|\gamma(x)| = n \sum_{x \in S} f_x|\gamma(x)|.$$

Dropping the leading coefficient of n from the final expression gives us $\sum_{x \in S} f_x|\gamma(x)|$, the *average* number of bits required per letter. We denote this quantity by $\text{ABL}(\gamma)$.

To continue the earlier example, suppose we have a text with the letters $S = \{a, b, c, d, e\}$, and their frequencies are as follows:

$$f_a = .32, \quad f_b = .25, \quad f_c = .20, \quad f_d = .18, \quad f_e = .05.$$

Then the average number of bits per letter using the prefix code γ_1 defined previously is

$$.32 \cdot 2 + .25 \cdot 2 + .20 \cdot 3 + .18 \cdot 2 + .05 \cdot 3 = 2.25.$$

It is interesting to compare this to the average number of bits per letter using a fixed-length encoding. (Note that a fixed-length encoding is a prefix code: if all letters have encodings of the same length, then clearly no encoding can be a prefix of any other.) With a set S of five letters, we would need three bits per letter for a fixed-length encoding, since two bits could only encode four letters. Thus, using the code γ_1 reduces the bits per letter from 3 to 2.25, a savings of 25 percent.

And, in fact, γ_1 is not the best we can do in this example. Consider the prefix code γ_2 given by

$$\gamma_2(a) = 11$$
$$\gamma_2(b) = 10$$
$$\gamma_2(c) = 01$$
$$\gamma_2(d) = 001$$
$$\gamma_2(e) = 000$$

The average number of bits per letter using γ_2 is

$$.32 \cdot 2 + .25 \cdot 2 + .20 \cdot 2 + .18 \cdot 3 + .05 \cdot 3 = 2.23.$$

So now it is natural to state the underlying question. Given an alphabet and a set of frequencies for the letters, we would like to produce a prefix code that is as efficient as possible—namely, a prefix code that minimizes the average number of bits per letter $\text{ABL}(\gamma) = \sum_{x \in S} f_x |\gamma(x)|$. We will call such a prefix code *optimal*.

Designing the Algorithm

The search space for this problem is fairly complicated; it includes all possible ways of mapping letters to bit strings, subject to the defining property of prefix codes. For alphabets consisting of an extremely small number of letters, it is feasible to search this space by brute force, but this rapidly becomes infeasible.

We now describe a greedy method to construct an optimal prefix code very efficiently. As a first step, it is useful to develop a tree-based means of representing prefix codes that exposes their structure more clearly than simply the lists of function values we used in our previous examples.

Representing Prefix Codes Using Binary Trees Suppose we take a rooted tree T in which each node that is not a leaf has at most two children; we call such a tree a *binary tree*. Further suppose that the number of leaves is equal to the size of the alphabet S, and we label each leaf with a distinct letter in S.

Such a labeled binary tree T naturally describes a prefix code, as follows. For each letter $x \in S$, we follow the path from the root to the leaf labeled x; each time the path goes from a node to its left child, we write down a 0, and each time the path goes from a node to its right child, we write down a 1. We take the resulting string of bits as the encoding of x.

Now we observe

(4.27) *The encoding of S constructed from T is a prefix code.*

Proof. In order for the encoding of x to be a prefix of the encoding of y, the path from the root to x would have to be a prefix of the path from the root to y. But this is the same as saying that x would lie on the path from the root to y, which isn't possible if x is a leaf. ■

This relationship between binary trees and prefix codes works in the other direction as well. Given a prefix code γ, we can build a binary tree recursively as follows. We start with a root; all letters $x \in S$ whose encodings begin with a 0 will be leaves in the left subtree of the root, and all letters $y \in S$ whose encodings begin with a 1 will be leaves in the right subtree of the root. We now build these two subtrees recursively using this rule.

For example, the labeled tree in Figure 4.16(a) corresponds to the prefix code γ_0 specified by

$$\gamma_0(a) = 1$$
$$\gamma_0(b) = 011$$
$$\gamma_0(c) = 010$$
$$\gamma_0(d) = 001$$
$$\gamma_0(e) = 000$$

To see this, note that the leaf labeled a is obtained by simply taking the right-hand edge out of the root (resulting in an encoding of 1); the leaf labeled e is obtained by taking three successive left-hand edges starting from the root; and analogous explanations apply for b, c, and d. By similar reasoning, one can see that the labeled tree in Figure 4.16(b) corresponds to the prefix code γ_1 defined earlier, and the labeled tree in Figure 4.16(c) corresponds to the prefix code γ_2 defined earlier. Note also that the binary trees for the two prefix codes γ_1 and γ_2 are identical in structure; only the labeling of the leaves is different. The tree for γ_0, on the other hand, has a different structure.

Thus the search for an optimal prefix code can be viewed as the search for a binary tree T, together with a labeling of the leaves of T, that minimizes the average number of bits per letter. Moreover, this average quantity has a natural interpretation in the terms of the structure of T: the length of the encoding of a letter $x \in S$ is simply the length of the path from the root to the leaf labeled x. We will refer to the length of this path as the *depth* of the leaf, and we will denote the depth of a leaf v in T simply by $\text{depth}_T(v)$. (As two bits of notational convenience, we will drop the subscript T when it is clear from context, and we will often use a letter $x \in S$ to also denote the leaf that is labeled by it.) Thus we are seeking the labeled tree that minimizes the weighted average of the depths of all leaves, where the average is weighted by the frequencies of the letters that label the leaves: $\sum_{x \in S} f_x \cdot \text{depth}_T(x)$. We will use $\text{ABL}(T)$ to denote this quantity.

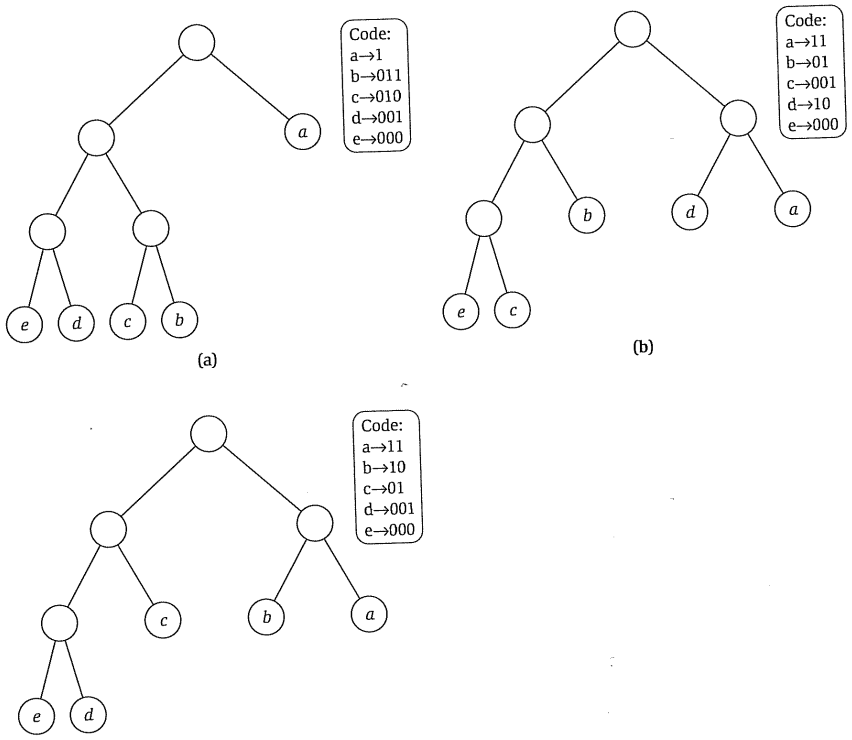

Figure 4.16 Parts (a), (b), and (c) of the figure depict three different prefix codes for the alphabet $S = \{a, b, c, d, e\}$.

As a first step in considering algorithms for this problem, let's note a simple fact about the optimal tree. For this fact, we need a definition: we say that a binary tree is *full* if each node that is not a leaf has two children. (In other words, there are no nodes with exactly one child.) Note that all three binary trees in Figure 4.16 are full.

(4.28) *The binary tree corresponding to the optimal prefix code is full.*

Proof. This is easy to prove using an exchange argument. Let T denote the binary tree corresponding to the optimal prefix code, and suppose it contains

a node u with exactly one child v. Now convert T into a tree T' by replacing node u with v.

To be precise, we need to distinguish two cases. If u was the root of the tree, we simply delete node u and use v as the root. If u is not the root, let w be the parent of u in T. Now we delete node u and make v be a child of w in place of u. This change decreases the number of bits needed to encode any leaf in the subtree rooted at node u, and it does not affect other leaves. So the prefix code corresponding to T' has a smaller average number of bits per letter than the prefix code for T, contradicting the optimality of T. ■

A First Attempt: The Top-Down Approach Intuitively, our goal is to produce a labeled binary tree in which the leaves are as close to the root as possible. This is what will give us a small average leaf depth.

A natural way to do this would be to try building a tree from the top down by "packing" the leaves as tightly as possible. So suppose we try to split the alphabet S into two sets S_1 and S_2, such that the total frequency of the letters in each set is exactly $\frac{1}{2}$. If such a perfect split is not possible, then we can try for a split that is as nearly balanced as possible. We then recursively construct prefix codes for S_1 and S_2 independently, and make these the two subtrees of the root. (In terms of bit strings, this would mean sticking a 0 in front of the encodings we produce for S_1, and sticking a 1 in front of the encodings we produce for S_2.)

It is not entirely clear how we should concretely define this "nearly balanced" split of the alphabet, but there are ways to make this precise. The resulting encoding schemes are called *Shannon-Fano* codes, named after Claude Shannon and Robert Fano, two of the major early figures in the area of *information theory*, which deals with representing and encoding digital information. These types of prefix codes can be fairly good in practice, but for our present purposes they represent a kind of dead end: no version of this top-down splitting strategy is guaranteed to always produce an optimal prefix code. Consider again our example with the five-letter alphabet $S = \{a, b, c, d, e\}$ and frequencies

$$f_a = .32, \quad f_b = .25, \quad f_c = .20, \quad f_d = .18, \quad f_e = .05.$$

There is a unique way to split the alphabet into two sets of equal frequency: $\{a, d\}$ and $\{b, c, e\}$. For $\{a, d\}$, we can use a single bit to encode each. For $\{b, c, e\}$, we need to continue recursively, and again there is a unique way to split the set into two subsets of equal frequency. The resulting code corresponds to the code γ_1, given by the labeled tree in Figure 4.16(b); and we've already seen that γ_1 is not as efficient as the prefix code γ_2 corresponding to the labeled tree in Figure 4.16(c).

Shannon and Fano knew that their approach did not always yield the optimal prefix code, but they didn't see how to compute the optimal code without brute-force search. The problem was solved a few years later by David Huffman, at the time a graduate student who learned about the question in a class taught by Fano.

We now describe the ideas leading up to the greedy approach that Huffman discovered for producing optimal prefix codes.

What If We Knew the Tree Structure of the Optimal Prefix Code? A technique that is often helpful in searching for an efficient algorithm is to assume, as a thought experiment, that one knows something partial about the optimal solution, and then to see how one would make use of this partial knowledge in finding the complete solution. (Later, in Chapter 6, we will see in fact that this technique is a main underpinning of the *dynamic programming* approach to designing algorithms.)

For the current problem, it is useful to ask: What if someone gave us the binary tree T^* that corresponded to an optimal prefix code, but not the labeling of the leaves? To complete the solution, we would need to figure out which letter should label which leaf of T^*, and then we'd have our code. How hard is this?

In fact, this is quite easy. We begin by formulating the following basic fact.

(4.29) *Suppose that u and v are leaves of T^*, such that depth$(u) <$ depth(v). Further, suppose that in a labeling of T^* corresponding to an optimal prefix code, leaf u is labeled with $y \in S$ and leaf v is labeled with $z \in S$. Then $f_y \geq f_z$.*

Proof. This has a quick proof using an exchange argument. If $f_y < f_z$, then consider the code obtained by exchanging the labels at the nodes u and v. In the expression for the average number of bits per letter, $\text{ABL}(T^*) = \sum_{x \in S} f_x \text{ depth}(x)$, the effect of this exchange is as follows: the multiplier on f_y increases (from depth(u) to depth(v)), and the multiplier on f_z decreases by the same amount (from depth(v) to depth(u)).

Thus the change to the overall sum is $(\text{depth}(v) - \text{depth}(u))(f_y - f_z)$. If $f_y < f_z$, this change is a negative number, contradicting the supposed optimality of the prefix code that we had before the exchange. ■

We can see the idea behind (4.29) in Figure 4.16(b): a quick way to see that the code here is not optimal is to notice that it can be improved by exchanging the positions of the labels c and d. Having a lower-frequency letter at a strictly smaller depth than some other higher-frequency letter is precisely what (4.29) rules out for an optimal solution.

Statement (4.29) gives us the following intuitively natural, and optimal, way to label the tree T^* if someone should give it to us. We first take all leaves of depth 1 (if there are any) and label them with the highest-frequency letters in any order. We then take all leaves of depth 2 (if there are any) and label them with the next-highest-frequency letters in any order. We continue through the leaves in order of increasing depth, assigning letters in order of decreasing frequency. The point is that this can't lead to a suboptimal labeling of T^*, since any supposedly better labeling would be susceptible to the exchange in (4.29). It is also crucial to note that, among the labels we assign to a block of leaves all at the same depth, it doesn't matter which label we assign to which leaf. Since the depths are all the same, the corresponding multipliers in the expression $\sum_{x \in S} f_x |\gamma(x)|$ are the same, and so the choice of assignment among leaves of the same depth doesn't affect the average number of bits per letter.

But how is all this helping us? We don't have the structure of the optimal tree T^*, and since there are exponentially many possible trees (in the size of the alphabet), we aren't going to be able to perform a brute-force search over all of them.

In fact, our reasoning about T^* becomes very useful if we think not about the very beginning of this labeling process, with the leaves of minimum depth, but about the very end, with the leaves of maximum depth—the ones that receive the letters with lowest frequency. Specifically, consider a leaf v in T^* whose depth is as large as possible. Leaf v has a parent u, and by (4.28) T^* is a full binary tree, so u has another child w. We refer to v and w as *siblings*, since they have a common parent. Now, we have

(4.30) *w is a leaf of T^*.*

Proof. If w were not a leaf, there would be some leaf w' in the subtree below it. But then w' would have a depth greater than that of v, contradicting our assumption that v is a leaf of maximum depth in T^*. ■

So v and w are sibling leaves that are as deep as possible in T^*. Thus our level-by-level process of labeling T^*, as justified by (4.29), will get to the level containing v and w last. The leaves at this level will get the lowest-frequency letters. Since we have already argued that the order in which we assign these letters to the leaves within this level doesn't matter, there is an optimal labeling in which v and w get the two lowest-frequency letters of all.

We sum this up in the following claim.

(4.31) *There is an optimal prefix code, with corresponding tree T^*, in which the two lowest-frequency letters are assigned to leaves that are siblings in T^*.*

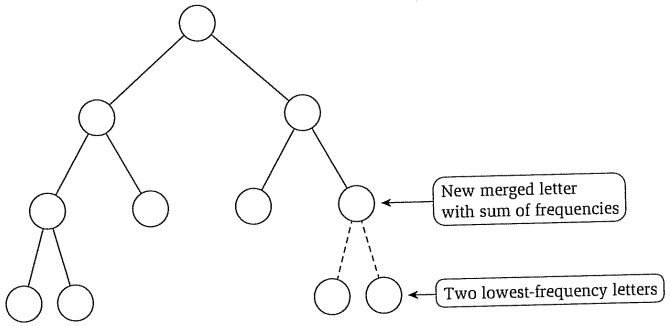

Figure 4.17 There is an optimal solution in which the two lowest-frequency letters label sibling leaves; deleting them and labeling their parent with a new letter having the combined frequency yields an instance with a smaller alphabet.

An Algorithm to Construct an Optimal Prefix Code Suppose that y^* and z^* are the two lowest-frequency letters in S. (We can break ties in the frequencies arbitrarily.) Statement (4.31) is important because it tells us something about where y^* and z^* go in the optimal solution; it says that it is safe to "lock them together" in thinking about the solution, because we know they end up as sibling leaves below a common parent. In effect, this common parent acts like a "meta-letter" whose frequency is the sum of the frequencies of y^* and z^*.

This directly suggests an algorithm: we replace y^* and z^* with this meta-letter, obtaining an alphabet that is one letter smaller. We recursively find a prefix code for the smaller alphabet, and then "open up" the meta-letter back into y^* and z^* to obtain a prefix code for S. This recursive strategy is depicted in Figure 4.17.

A concrete description of the algorithm is as follows.

```
To construct a prefix code for an alphabet S, with given frequencies:
  If S has two letters then
    Encode one letter using 0 and the other letter using 1
  Else
    Let y* and z* be the two lowest-frequency letters
    Form a new alphabet S' by deleting y* and z* and
      replacing them with a new letter ω of frequency f_y* + f_z*
    Recursively construct a prefix code γ' for S', with tree T'
    Define a prefix code for S as follows:
      Start with T'
      Take the leaf labeled ω and add two children below it
        labeled y* and z*
  Endif
```


We refer to this as *Huffman's Algorithm*, and the prefix code that it produces for a given alphabet is accordingly referred to as a *Huffman code*. In general, it is clear that this algorithm always terminates, since it simply invokes a recursive call on an alphabet that is one letter smaller. Moreover, using (4.31), it will not be difficult to prove that the algorithm in fact produces an optimal prefix code. Before doing this, however, we pause to note some further observations about the algorithm.

First let's consider the behavior of the algorithm on our sample instance with $S = \{a, b, c, d, e\}$ and frequencies

$$f_a = .32, \quad f_b = .25, \quad f_c = .20, \quad f_d = .18, \quad f_e = .05.$$

The algorithm would first merge d and e into a single letter—let's denote it (de)—of frequency $.18 + .05 = .23$. We now have an instance of the problem on the four letters $S' = \{a, b, c, (de)\}$. The two lowest-frequency letters in S' are c and (de), so in the next step we merge these into the single letter (cde) of frequency $.20 + .23 = .43$. This gives us the three-letter alphabet $\{a, b, (cde)\}$. Next we merge a and b, and this gives us a two-letter alphabet, at which point we invoke the base case of the recursion. If we unfold the result back through the recursive calls, we get the tree pictured in Figure 4.16(c).

It is interesting to note how the greedy rule underlying Huffman's Algorithm—the merging of the two lowest-frequency letters—fits into the structure of the algorithm as a whole. Essentially, at the time we merge these two letters, we don't know exactly how they will fit into the overall code. Rather, we simply commit to having them be children of the same parent, and this is enough to produce a new, equivalent problem with one less letter.

Moreover, the algorithm forms a natural contrast with the earlier approach that led to suboptimal Shannon-Fano codes. That approach was based on a top-down strategy that worried first and foremost about the top-level split in the binary tree—namely, the two subtrees directly below the root. Huffman's Algorithm, on the other hand, follows a bottom-up approach: it focuses on the leaves representing the two lowest-frequency letters, and then continues by recursion.

Analyzing the Algorithm

The Optimality of the Algorithm We first prove the optimality of Huffman's Algorithm. Since the algorithm operates recursively, invoking itself on smaller and smaller alphabets, it is natural to try establishing optimality by induction

on the size of the alphabet. Clearly it is optimal for all two-letter alphabets (since it uses only one bit per letter). So suppose by induction that it is optimal for all alphabets of size $k-1$, and consider an input instance consisting of an alphabet S of size k.

Let's quickly recap the behavior of the algorithm on this instance. The algorithm merges the two lowest-frequency letters $y^*, z^* \in S$ into a single letter ω, calls itself recursively on the smaller alphabet S' (in which y^* and z^* are replaced by ω), and by induction produces an optimal prefix code for S', represented by a labeled binary tree T'. It then extends this into a tree T for S, by attaching leaves labeled y^* and z^* as children of the node in T' labeled ω.

There is a close relationship between $\text{ABL}(T)$ and $\text{ABL}(T')$. (Note that the former quantity is the average number of bits used to encode letters in S, while the latter quantity is the average number of bits used to encode letters in S'.)

(4.32) $\text{ABL}(T') = \text{ABL}(T) - f_\omega$.

Proof. The depth of each letter x other than y^*, z^* is the same in both T and T'. Also, the depths of y^* and z^* in T are each one greater than the depth of ω in T'. Using this, plus the fact that $f_\omega = f_{y^*} + f_{z^*}$, we have

$$\begin{aligned}
\text{ABL}(T) &= \sum_{x \in S} f_x \cdot \text{depth}_T(x) \\
&= f_{y^*} \cdot \text{depth}_T(y^*) + f_{z^*} \cdot \text{depth}_T(z^*) + \sum_{x \neq y^*, z^*} f_x \cdot \text{depth}_T(x) \\
&= (f_{y^*} + f_{z^*}) \cdot (1 + \text{depth}_{T'}(\omega)) + \sum_{x \neq y^*, z^*} f_x \cdot \text{depth}_{T'}(x) \\
&= f_\omega \cdot (1 + \text{depth}_{T'}(\omega)) + \sum_{x \neq y^*, z^*} f_x \cdot \text{depth}_{T'}(x) \\
&= f_\omega + f_\omega \cdot \text{depth}_{T'}(\omega) + \sum_{x \neq y^*, z^*} f_x \cdot \text{depth}_{T'}(x) \\
&= f_\omega + \sum_{x \in S'} f_x \cdot \text{depth}_{T'}(x) \\
&= f_\omega + \text{ABL}(T'). \quad \blacksquare
\end{aligned}$$

Using this, we now prove optimality as follows.

(4.33) *The Huffman code for a given alphabet achieves the minimum average number of bits per letter of any prefix code.*

Proof. Suppose by way of contradiction that the tree T produced by our greedy algorithm is not optimal. This means that there is some labeled binary tree Z such that $\text{ABL}(Z) < \text{ABL}(T)$; and by (4.31), there is such a tree Z in which the leaves representing y^* and z^* are siblings.

It is now easy to get a contradiction, as follows. If we delete the leaves labeled y^* and z^* from Z, and label their former parent with ω, we get a tree Z' that defines a prefix code for S'. In the same way that T is obtained from T', the tree Z is obtained from Z' by adding leaves for y^* and z^* below ω; thus the identity in (4.32) applies to Z and Z' as well: $\text{ABL}(Z') = \text{ABL}(Z) - f_\omega$.

But we have assumed that $\text{ABL}(Z) < \text{ABL}(T)$; subtracting f_ω from both sides of this inequality we get $\text{ABL}(Z') < \text{ABL}(T')$, which contradicts the optimality of T' as a prefix code for S'. ■

Implementation and Running Time It is clear that Huffman's Algorithm can be made to run in polynomial time in k, the number of letters in the alphabet. The recursive calls of the algorithm define a sequence of $k-1$ iterations over smaller and smaller alphabets, and each iteration except the last consists simply of identifying the two lowest-frequency letters and merging them into a single letter that has the combined frequency. Even without being careful about the implementation, identifying the lowest-frequency letters can be done in a single scan of the alphabet, in time $O(k)$, and so summing this over the $k-1$ iterations gives $O(k^2)$ time.

But in fact Huffman's Algorithm is an ideal setting in which to use a priority queue. Recall that a priority queue maintains a set of k elements, each with a numerical key, and it allows for the insertion of new elements and the extraction of the element with the minimum key. Thus we can maintain the alphabet S in a priority queue, using each letter's frequency as its key. In each iteration we just extract the minimum twice (this gives us the two lowest-frequency letters), and then we insert a new letter whose key is the sum of these two minimum frequencies. Our priority queue now contains a representation of the alphabet that we need for the next iteration.

Using an implementation of priority queues via heaps, as in Chapter 2, we can make each insertion and extraction of the minimum run in time $O(\log k)$; hence, each iteration—which performs just three of these operations—takes time $O(\log k)$. Summing over all k iterations, we get a total running time of $O(k \log k)$.

Extensions

The structure of optimal prefix codes, which has been our focus here, stands as a fundamental result in the area of data compression. But it is important to understand that this optimality result does not by any means imply that we have found the best way to compress data under all circumstances.