3 Huffman Coding

3.1 Overview

In this chapter we describe a very popular coding algorithm called the Huffman coding algorithm. We first present a procedure for building Huffman codes when the probability model for the source is known, and then we introduce a procedure for building codes when the source statistics are unknown. We also describe a few techniques for code design that are in some sense similar to the Huffman coding approach. Finally, we give some examples of using the Huffman code for image compression, audio compression, and text compression.

3.2 The Huffman Coding Algorithm

This technique was developed by David Huffman as part of a class assignment; the class was the first ever in the area of information theory and was taught by Robert Fano at MIT [16]. The codes generated using this technique or procedure are called *Huffman codes*. These codes are prefix codes and are optimum for a given model (set of probabilities).

The Huffman procedure is based on two observations regarding optimum prefix codes.

1. In an optimum code, symbols that occur more frequently (have a higher probability of occurrence) will have shorter codewords than symbols that occur less frequently.
2. In an optimum code, the two symbols that occur least frequently will have the same length.

Introduction to Data Compression. DOI: http://dx.doi.org/10.1016/B978-0-12-415796-5.00003-X

It is easy to see that the first observation is correct. If symbols that occur more often had codewords that were longer than the codewords for symbols that occurred less often, the average number of bits per symbol would be larger than if the conditions were reversed. Therefore, a code that assigns longer codewords to symbols that occur more frequently cannot be optimum.

To see why the second observation holds true, consider the following situation. Suppose an optimum code $\mathcal{C}$ exists in which the two codewords corresponding to the two least probable symbols do not have the same length. Suppose the longer codeword is k bits longer than the shorter codeword. Because this is a prefix code, the shorter codeword cannot be a prefix of the longer codeword. This means that even if we drop the last k bits of the longer codeword, the two codewords would still be distinct. As these codewords correspond to the least probable symbols in the alphabet, no other codeword can be longer than these codewords; therefore, there is no danger that the shortened codeword would become the prefix of some other codeword. Furthermore, by dropping these k bits we obtain a new code that has a shorter average length than $\mathcal{C}$. But this violates our initial contention that $\mathcal{C}$ is an optimal code. Therefore, for an optimal code the second observation also holds true.

The Huffman procedure adds a simple requirement to these two observations. This requirement is that the codewords corresponding to the two lowest probability symbols differ only in the last bit. Assume that γ and δ are the two least probable symbols in an alphabet. If the codeword for γ is $\mathbf{m} * 0$, the codeword for δ would be $\mathbf{m} * 1$. Here, $\mathbf{m}$ is a string of 1s and 0s, and $*$ denotes concatenation.

This requirement does not violate our two observations and leads to a very simple encoding procedure. We describe this procedure with the help of the following example.

Example 3.2.1: Design of a Huffman Code

Let us design a Huffman code for a source that puts out letters from an alphabet $\mathcal{A} = \{a_1, a_2, a_3, a_4, a_5\}$ with $P(a_1) = P(a_3) = 0.2$, $P(a_2) = 0.4$, and $P(a_4) = P(a_5) = 0.1$. The entropy for this source is 2.122 bits/symbol. To design the Huffman code, we first sort the letters in a descending probability order as shown in Table 3.1. Here $c(a_i)$ denotes the codeword for a_i.

The two symbols with the lowest probability are a_4 and a_5. Therefore, we can assign their codewords as

TABLE 3.1 The initial five-letter alphabet.

Letter	Probability	Codeword
a_2	0.4	$c(a_2)$
a_1	0.2	$c(a_1)$
a_3	0.2	$c(a_3)$
a_4	0.1	$c(a_4)$
a_5	0.1	$c(a_5)$

TABLE 3.2 The reduced four-letter alphabet.

Letter	Probability	Codeword
a_2	0.4	$c(a_2)$
a_1	0.2	$c(a_1)$
a_3	0.2	$c(a_3)$
a'_4	0.2	α_1

TABLE 3.3 The reduced three-letter alphabet.

Letter	Probability	Codeword
a_2	0.4	$c(a_2)$
a'_3	0.4	α_2
a_1	0.2	$c(a_1)$

$$c(a_4) = \alpha_1 * 0$$
$$c(a_5) = \alpha_1 * 1$$

where α_1 is a binary string, and $*$ denotes concatenation.

We now define a new alphabet A' with a four-letter alphabet a_1, a_2, a_3, a'_4, where a'_4 is composed of a_4 and a_5 and has a probability $P(a'_4) = P(a_4) + P(a_5) = 0.2$. We sort this new alphabet in descending order to obtain Table 3.2.

In this alphabet, a_3 and a'_4 are the two letters at the bottom of the sorted list. We assign their codewords as

$$c(a_3) = \alpha_2 * 0$$
$$c(a'_4) = \alpha_2 * 1$$

but $c(a'_4) = \alpha_1$. Therefore,

$$\alpha_1 = \alpha_2 * 1$$

which means that

$$c(a_4) = \alpha_2 * 10$$
$$c(a_5) = \alpha_2 * 11$$

At this stage, we again define a new alphabet A'' that consists of three letters a_1, a_2, a'_3, where a'_3 is composed of a_3 and a'_4 and has a probability $P(a'_3) = P(a_3) + P(a'_4) = 0.4$. We sort this new alphabet in descending order to obtain Table 3.3.

In this case, the two least probable symbols are a_1 and a'_3. Therefore,

$$c(a'_3) = \alpha_3 * 0$$
$$c(a_1) = \alpha_3 * 1$$

TABLE 3.4 The reduced two-letter alphabet.

Letter	Probability	Codeword
a_3''	0.6	α_3
a_2	0.4	$c(a_2)$

TABLE 3.5 Huffman code for the original five-letter alphabet.

Letter	Probability	Codeword
a_2	0.4	1
a_1	0.2	01
a_3	0.2	000
a_4	0.1	0010
a_5	0.1	0011

But $c(a_3') = \alpha_2$. Therefore,

$$\alpha_2 = \alpha_3 * 0$$

which means that

$$c(a_3) = \alpha_3 * 00$$
$$c(a_4) = \alpha_3 * 010$$
$$c(a_5) = \alpha_3 * 011$$

Again we define a new alphabet, this time with only two letters a_3'' and a_2. Here a_3'' is composed of the letters a_3' and a_1 and has probability $P(a_3'') = P(a_3') + P(a_1) = 0.6$. We now have Table 3.4.

As we have only two letters, the codeword assignment is straightforward:

$$c(a_3'') = 0$$
$$c(a_2) = 1$$

which means that $\alpha_3 = 0$, which in turn means that

$$c(a_1) = 01$$
$$c(a_3) = 000$$
$$c(a_4) = 0010$$
$$c(a_5) = 0011$$

The Huffman code is given by Table 3.5. The procedure can be summarized as shown in Figure 3.1. ♦

The average length for this code is

$$l = .4 \times 1 + .2 \times 2 + .2 \times 3 + .1 \times 4 + .1 \times 4 = 2.2 \text{ bits/symbol}$$

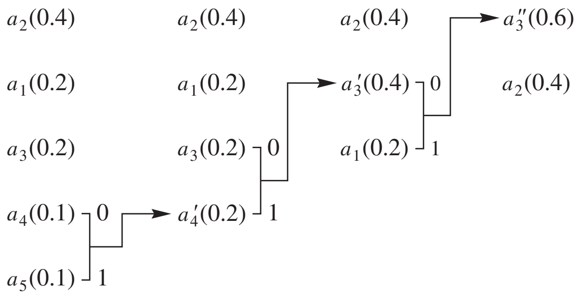

FIGURE 3.1 The Huffman encoding procedure. The symbol probabilities are listed in parentheses.

A measure of the efficiency of this code is its *redundancy*—the difference between the entropy and the average length. In this case, the redundancy is 0.078 bits/symbol. The redundancy is zero when the probabilities are negative powers of two.

An alternative way of building a Huffman code is to use the fact that the Huffman code, by virtue of being a prefix code, can be represented as a binary tree in which the external nodes or leaves correspond to the symbols. The Huffman code for any symbol can be obtained by traversing the tree from the root node to the leaf corresponding to the symbol, adding a 0 to the codeword every time the traversal takes us over an upper branch and a 1 every time the traversal takes us over a lower branch.

We build the binary tree starting at the leaf nodes. We know that the codewords for the two symbols with smallest probabilities are identical except for the last bit. This means that the traversal from the root to the leaves corresponding to these two symbols must be the same except for the last step. This in turn means that the leaves corresponding to the two symbols with the lowest probabilities are offspring of the same node. Once we have connected the leaves corresponding to the symbols with the lowest probabilities to a single node, we treat this node as a symbol of a reduced alphabet. The probability of this symbol is the sum of the probabilities of its offspring. We can now sort the nodes corresponding to the reduced alphabet and apply the same rule to generate a parent node for the nodes corresponding to the two symbols in the reduced alphabet with lowest probabilities. Continuing in this manner, we end up with a single node, which is the root node. To obtain the code for each symbol, we traverse the tree from the root to each leaf node, assigning a 0 to the upper branch and a 1 to the lower branch. This procedure, as applied to the alphabet of Example 3.2.1, is shown in Figure 3.2. Notice the similarity between Figures 3.1 and 3.2. This is not surprising, as they are a result of viewing the same procedure in two different ways.

3.2.1 Minimum Variance Huffman Codes

By performing the sorting procedure in a slightly different manner, we could have found a different Huffman code. In the first re-sort, we could place a_4' higher in the list, as shown in Table 3.6.

TABLE 3.6 Reduced four-letter alphabet.

Letter	Probability	Codeword
a_2	0.4	$c(a_2)$
a'_4	0.2	α_1
a_1	0.2	$c(a_1)$
a_3	0.2	$c(a_3)$

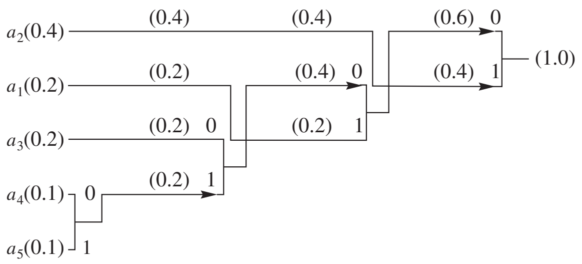

FIGURE 3.2 Building the binary Huffman tree.

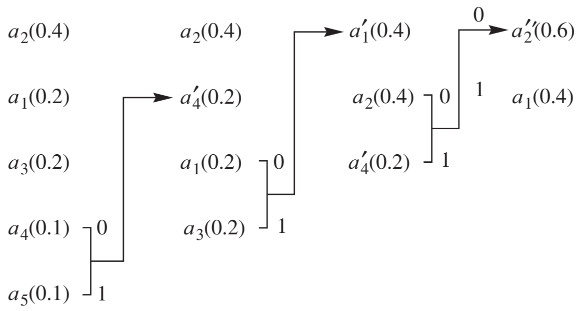

FIGURE 3.3 The minimum variance Huffman encoding procedure.

Now combine a_1 and a_3 into a'_1, which has a probability of 0.4. Sorting the alphabet a_2, a'_4, and a'_1 and putting a'_1 as far up the list as possible, we get Table 3.7. Finally, by combining a_2 and a'_4 and re-sorting, we get Table 3.8. If we go through the unbundling procedure, we get the codewords in Table 3.9. The procedure is summarized in Figure 3.3. The average length of the code is

$$l = .4 \times 2 + .2 \times 2 + .2 \times 2 + .1 \times 3 + .1 \times 3 = 2.2 \text{ bits/symbol.}$$

The two codes are identical in terms of their redundancy. However, the variance of the length of the codewords is significantly different. This can be clearly seen from Figure 3.4.

Remember that in many applications, although you might be using a variable-length code, the available transmission rate is generally fixed. For example, if we were going to transmit symbols from the alphabet we have been using at 10,000 symbols per second, we might ask for a transmission capacity of 22,000 bits per second. This means that during each second, the

TABLE 3.7 Reduced three-letter alphabet.

Letter	Probability	Codeword
a_1'	0.4	α_2
a_2	0.4	$c(a_2)$
a_4'	0.2	α_1

TABLE 3.8 Reduced two-letter alphabet.

Letter	Probability	Codeword
a_2'	0.6	α_3
a_1'	0.4	α_2

TABLE 3.9 Minimum variance Huffman code.

Letter	Probability	Codeword
a_1	0.2	10
a_2	0.4	00
a_3	0.2	11
a_4	0.1	010
a_5	0.1	011

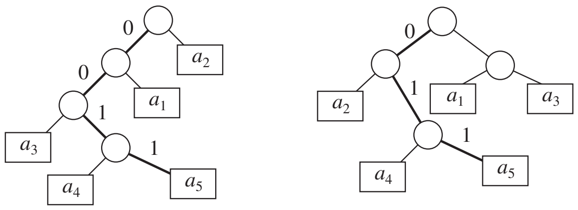

FIGURE 3.4 Two Huffman trees corresponding to the same probabilities.

channel expects to receive 22,000 bits, no more and no less. As the bit generation rate will vary around 22,000 bits per second, the output of the source coder is generally fed into a buffer. The purpose of the buffer is to smooth out the variations in the bit generation rate. However, the buffer has to be of finite size, and the greater the variance in the codewords, the more difficult the buffer design problem becomes. Suppose that the source we are discussing generates a string of a_4s and a_5s for several seconds. If we are using the first code, this means that we will be generating bits at a rate of 40,000 bits per second. For each second, the buffer has to store 18,000 bits. On the other hand, if we use the second code, we would be generating 30,000 bits per second, and the buffer would have to store 8000 bits for every second this condition persisted. If we have a string of a_2s instead of a string of a_4s and a_5s, the first code would result in the generation of 10,000 bits per second. Remember that the channel will still be expecting 22,000 bits every second, so somehow we will have to make up a deficit of 12,000

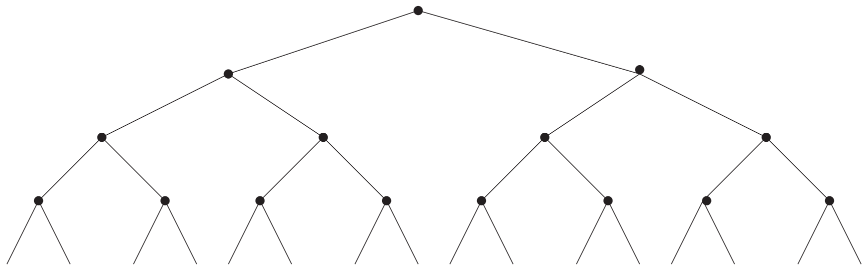

FIGURE 3.5 A binary tree of depth four.

bits per second. The same situation using the second code would lead to a deficit of 2000 bits per second. Thus, it seems reasonable to elect to use the second code instead of the first. To obtain the Huffman code with minimum variance, we always put the combined letter as high in the list as possible.

3.2.2 Canonical Huffman Codes

To be effective, a Huffman code needs to reflect the statistics of the source being encoded. When encoding sources with widely varying statistics, it is sometimes necessary to include the Huffman code along with the compressed representation of the source. Depending on the alphabet size, this can be a substantial burden, perhaps even cancelling out any advantages of compression. Therefore, it is useful to develop Huffman codes that can be stored in an efficient manner. Consider the Huffman code in Table 3.5. One way to store this code is to write the code in lexicographic order of the symbols. Thus, we first write the code for a_1, then the code for a_2, and so on. Each codeword could be represented by the length of the codeword followed by the codeword. Thus the code would be stored as [2, 01, 1, 1, 3, 000, 4, 0010, 5, 0011]. This is certainly manageable for the Huffman code of a small alphabet, but it is clearly going to have an impact on compression performance when we have Huffman codes for large alphabets.

We can substantially reduce the storage requirement for the code by using a version of the Huffman code known as the *canonical Huffman code*. Before we describe how to construct a canonical Huffman code, we examine a property of Huffman trees. Consider a Huffman tree such as one of the trees in Figure 3.4. For convenience, let us use the tree on the left. Notice that like all Huffman trees, this is a fully populated tree. That is, there is a codeword assigned to each leaf of the tree. What we will show is that if we are only given the lengths of the codewords moving from left to right on the tree we can reconstruct the code. For example, in this tree the lengths are 3, 4, 4, 2, and 1. In order to regenerate the code from the lengths, we begin with a tree of depth four (the longest codeword), as shown in Figure 3.5.

We then prune away the branches starting on the left, to match the lengths of the codewords. We first prune away the leftmost two branches, as shown in Figure 3.6, to leave a leaf at a depth of three. This corresponds to the first codeword. The next two codewords have a length of four, so we leave the next two leaves at a depth of four. The next codeword has a length of two so we prune all the lower branches as indicated in Figure 3.6, leaving a leaf at depth two. Finally, we prune most of the right half of the tree to get a leaf corresponding to the codeword of length one. The pruned tree can be populated by zeros on the left branches and

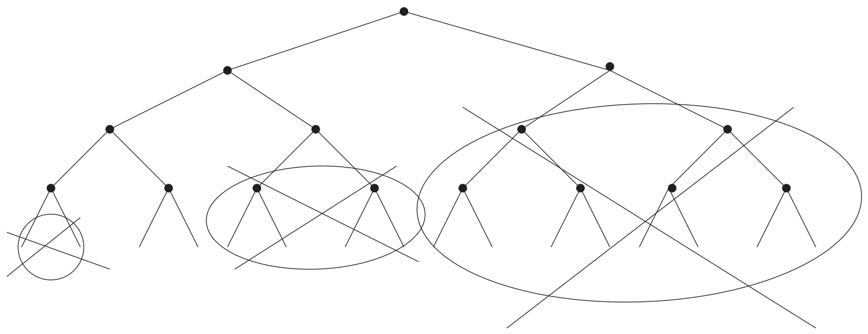

FIGURE 3.6 **Pruning the binary tree to get codewords of a specified length.**

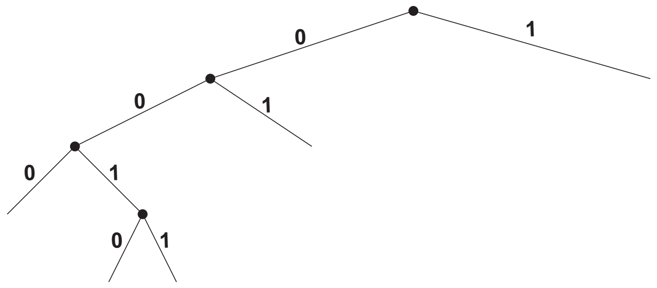

FIGURE 3.7 **Generating the code.**

ones on the right branches to generate the code as shown in Figure 3.7. From the figure, we can see that the codewords are 000, 0010, 0011, 01, and 1. So, just knowing the codeword lengths in a particular order permits us to regenerate the Huffman code. The problem is that we do not know which codeword belongs to which letter. The canonical procedure provides us with a way to generate a code that implicitly contains that information. In order to embed this additional information, we need some additional constraints to our Huffman coding procedure. For example, the *deflate* algorithm in zlib imposes the following conditions on a Huffman code [17]:

1. All codes of a given length have lexicographically consecutive values in the same order as the symbols they represent.
2. Shorter codes lexicographically precede longer codes.

Recall that when we generated Huffman codes, we could choose to assign zeros and ones to the branches as we wished. These constraints can be viewed as taking that freedom away.

We can incorporate these constraints into the Huffman coding procedure, or we can generate a Huffman code using the procedure described previously and transform the code thus produced into a canonical Huffman code. It is simpler to use the latter approach. In order to do this, we will begin by designing a Huffman code. From this design, we will extract the lengths of the codewords. We will use these lengths with the canonical constraints to design the code.

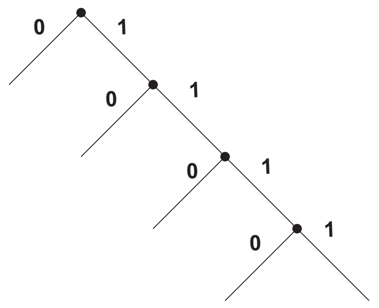

FIGURE 3.8 **Tree for the canonical Huffman code.**

We illustrate this procedure using the example we have been considering in this section. Our alphabet consists of five letters $\{a_1, a_2, a_3, a_4, a_5\}$. The lengths of the corresponding Huffman codewords are $\{2, 1, 3, 4, 4\}$. We generate the codewords from the shortest to the longest keeping in mind the constraint that the shorter codewords lexicographically precede longer codewords. The shortest codeword is that assigned to a_2. The lexicographically smallest codeword of length one is 0, so the codeword for a_2 is 0. The codewords of length two have to be of the form 1x. There is only one codeword of length two required, that being the codeword for a_1, so we assign the codeword 10 to a_1. The codewords of length three now have to be of the form 11x. We only need a single codeword of length three, the codeword for a_3; therefore, the codeword for a_3 is 110. There are two codewords, a_4 and a_5, of length four. Therefore, the codeword for a_4 is 1110, and the codeword for a_5 is 1111. This code can now be encoded by just sending the lengths. Unlike the previous code we know exactly which codeword belongs to which letter in the alphabet. This is because of the lexicographic constraints. As can be seen from the tree for this code in Figure 3.8, the tree increases in depth as we move from left to right. We know from the sequence of lengths that the shortest codeword is for the letter a_2, the next longest codeword is for the letter a_1, and so on. Thus simply encoding the list of lengths is sufficient for storing the Huffman code. For an alphabet of size five, this is not a significant saving; however, when the alphabet size is on the order of 256, the savings can be very significant.

3.2.3 Length-Limited Huffman Codes

The Huffman coding algorithm tries to minimize the *average* length of codewords. However, there are no limits on the *maximum* length of an individual codeword. This is not necessarily a problem when dealing with limited alphabet sizes. However, if the alphabet size is relatively large there is a possibility of getting very long Huffman codes. This could result in codewords that do not fit in a single machine word. This in turn can lead to inefficiencies in the coding process. These problems can be avoided by adding an additional constraint to the variable length code design problem: a requirement that all codewords be less than or equal to some maximum codeword length l_{max}. If m is the size of the source alphabet then clearly l_{max} has to be greater than or equal to $\lceil \log_2 m \rceil$ for the code to be a valid code.

The most widely used algorithm for constructing length-limited Huffman codes is the *package-merge* algorithm due to Larmore and Hirchberg [18]. The package-merge algorithm is more general than what we describe here, our only interest being its application to designing length-limited Huffman codes. Our description is based on the work of Turpin and Moffat [19]. We will use a couple of facts from our prior discussions to design length-limited Huffman codes. First, as we have seen in the previous section, all we need to obtain a Huffman code is the length of the codewords. Second, as we showed in Chapter 2, for an alphabet of size m the lengths of the codewords $\{l_1, l_2, \cdots, l_m\}$ in any variable length code have to satisfy the Kraft-McMillan inequality

$$\sum_{i=1}^{m} 2^{-l_i} \leqslant 1$$

and, given a set of l_i that satisfy the Kraft-McMillan inequality, we can always find a uniquely decodable code with codewords of length l_i.

Suppose our alphabet consists of the letters $\{a_1, a_2, \ldots, a_m\}$ with probabilities $\{p_1, p_2, \ldots, p_m\}$. We will assume that the letters have been sorted based on their probabilities such that $p_1 \leqslant p_2 \leqslant \cdots \leqslant p_m$. Our design process will involve incrementing the lengths l_i until the Kraft-McMillan inequality is satisfied while at the same time keeping the average length $\bar{l}$ as small as we can. We start with assigning zero bits for all lengths. Clearly, this will result in the lowest possible length of zero. However, it certainly does not give us a uniquely decodable code. Not that we need to justify the latter point mathematically, but if we did, we could compute the Kraft-McMillan sum as

$$\sum_{i=1}^{m} 2^{-l_i} = \sum_{i=1}^{m} 2^{0} = m$$

which is clearly greater than one. We can decrement this sum by 0.5 by incrementing the codeword for one of the letters to one. We want to pick the codeword that will have the minimal impact on the average codeword length. As the letters are sorted in nondecreasing probability order and the average codeword length is given by

$$\bar{l} = \sum_{i=1}^{m} p_i l_i$$

we will increase l_1 to one. This will give us an average codeword length of p_1 and a Kraft-McMillan sum of $m - 0.5$. Actually, we could have picked any of the letters as we will need to increment each length to at least one if we are to have any hope of reducing the Kraft-McMillan sum to one. However, once we have incremented each codeword length to one how do we decide which codeword lengths to increment further? The package-merge algorithm is an iterative algorithm that solves this problem by generating a list of choices that can be sorted in order of increasing cost. Each iteration consists of two steps operating on the list, a packaging step and a merge step. In the packaging step the list from the previous step is partitioned, or packaged, into groups of two neighboring items. The cost of a package is the sum of the costs of the items in the package where the cost of a letter is the probability of that letter. If the number of items is odd the item with the highest cost is discarded. In the merge step the packages from the previous step are merged with the list of items that consists of the individual letters. This list is then sorted in order of increasing cost. After $l_{max} - 1$ iterations the lowest cost $2m - 2$ items are selected. The length of the codeword for each letter is equal to the number of items containing that letter. As there are $2m - 2$ items, each of which decrements the Kraft-McMillan sum by 0.5, we end up with a set of lengths $\{l_i\}$ such that

$$\sum_{i=1}^{m} l_i = 1$$

Example 3.2.2:

Let us design a length-limited Huffman code for a source that puts out letters from an alphabet $\mathcal{A} = \{a_1, a_2, a_3, a_4, a_5, a_6\}$ with $P(a_1) = 0.05$, $P(a_2) = 0.1$, $P(a_3) = 0.15$, $P(a_4) = P(a_5) = 0.2$, and $P(a_6) = 0.3$. The entropy for this source is 2.409 bits/symbol. Using the standard Huffman procedure we can generate the code shown in Table 3.10 with an average length $\bar{l} = 2.45$. The longest codewords are four bits long. Let us design a code with the added constraint that $l_{max} = 3$.

We begin with the list of letters. The costs are listed next to the letters:

$$L_0 = [a_1(0.05), a_2(0.1), a_3(0.15), a_4(0.2), a_5(0.2), a_6(0.3)]$$

We then proceed with the first packaging step. The packages are denoted by $a_{jkl}(p)$ where the subscripts indicate the letters in the package and the quantity in the argument is the total cost of the item. In the first packaging step we form packages by grouping the letters two-by-two to form

$$\text{Package}_1 : [a_{12}(.15), a_{34}(.35), a_{56}(.5)]$$

We then merge these packages with our original list to get the merged list:

$$\text{Merge}_1 \text{: } [a_1(0.05), a_2(0.1), a_3(0.15), a_{12}(0.15), a_4(0.2), a_5(0.2), a_6(0.3), a_{34}(0.35), a_{56}(0.5)]$$

To get a code limited to three bits we need to go through one more iteration. In the next packaging step we take the items in the previous merged list and group them two-by-two:

$$\text{Package}_2 : [a_{12}(0.15), a_{312}(0.3), a_{45}(0.4), a_{634}(0.65)]$$

TABLE 3.10 Huffman code.

Letter	Probability	Codeword
a_1	0.05	0100
a_2	0.1	0101
a_3	0.15	011
a_4	0.2	10
a_5	0.2	11
a_6	0.3	00

TABLE 3.11 Length-limited Huffman code.

Letter	Probability	Codeword
a_1	0.05	100
a_2	0.1	101
a_3	0.15	110
a_4	0.2	111
a_5	0.2	00
a_6	0.3	01

Finally, we go through one more merge operation in which we merge the packages with the original list of letters:

$$\text{Merge}_2 : [a_1(0.05), a_2(0.1), a_3(0.15), a_{12}(0.15), a_4(0.2), a_5(0.2),$$
$$a_6(0.3), a_{312}(0.3), a_{45}(0.4), a_{634}(0.65)]$$

We now pick the ten items ($2 \times 6 - 2 = 10$) with the lowest cost and count the number of times each letter appears in the ten items. The letter a_1 appears in a_1, a_{12}, and a_{312}. Therefore, the number of bits in the codeword for a_1 is three. Continuing in this fashion we obtain the lengths of the codewords as $\{3, 3, 3, 3, 2, 2\}$. A code with these lengths is shown in Table 3.11. The average codeword length is 2.5 bits. Comparing this code with the Huffman code in Table 3.10, the cost of limiting the length of the longest codeword to three bits is $2.5 - 2.45 = 0.05$. ♦

3.2.4 Optimality of Huffman Codes ★

The optimality of Huffman codes can be proven rather simply by first writing down the necessary conditions that an optimal code has to satisfy and then showing that satisfying these conditions necessarily leads to designing a Huffman code. The proof we present here is based on the proof shown in [20] and is obtained for the binary case (for a more general proof, see [20]).

The necessary conditions for an optimal variable-length binary code are as follows:

- **Condition 1:** Given any two letters, a_j and a_k, if $P[a_j] \geqslant P[a_k]$, then $l_j \leqslant l_k$, where l_j is the number of bits in the codeword for a_j.
- **Condition 2:** The two least probable letters have codewords with the same maximum length l_m.

We have provided the justification for these two conditions in the opening sections of this chapter.

- **Condition 3:** In the tree corresponding to the optimum code, there must be two branches stemming from each intermediate node.

If there were any intermediate node with only one branch coming from that node, we could remove it without affecting the decipherability of the code while reducing its average length.

- **Condition 4:** Suppose we change an intermediate node into a leaf node by combining all of the leaves descending from it into a composite word of a reduced alphabet. Then, if the original tree was optimal for the original alphabet, the reduced tree would be optimal for the reduced alphabet.

If this condition were not satisfied, we could find a code with a smaller average code length for the reduced alphabet and then simply expand the composite word again to get a new code tree that would have a shorter average length than our original "optimum" tree. This would contradict our statement about the optimality of the original tree.

In order to satisfy conditions 1, 2, and 3, the two least probable letters would have to be assigned codewords of maximum length l_m. Furthermore, the leaves corresponding to these letters arise from the same intermediate node. This is the same as saying that the codewords for these letters are identical except for the last bit. Consider the common prefix as the codeword for the composite letter of a reduced alphabet. Since the code for the reduced alphabet needs to be optimum for the code of the original alphabet to be optimum, we follow the same procedure again. To satisfy the necessary conditions, the procedure needs to be iterated until we have a reduced alphabet of size one. But this is exactly the Huffman procedure. Therefore, the necessary conditions above, which are all satisfied by the Huffman procedure, are also sufficient conditions.

3.2.5 Length of Huffman Codes ★

We have said that the Huffman coding procedure generates an optimum code, but we have not said what the average length of an optimum code is. The length of any code will depend on a number of things, including the size of the alphabet and the probabilities of individual letters. In this section, we will show that the optimal code for a source $\mathcal{S}$, and hence the Huffman code for the source $\mathcal{S}$, has an average code length $\bar{l}$ bounded below by the entropy and bounded above by the entropy plus 1 bit. In other words,

$$H(\mathcal{S}) \leqslant \bar{l} < H(\mathcal{S}) + 1 \tag{1}$$

In order for us to do this, we will need to use the Kraft-McMillan inequality introduced in Chapter 2. Recall that the first part of this result, due to McMillan, states that if we have a uniquely decodable code $\mathcal{C}$ with K codewords of length $\{l_i\}_{i=1}^{K}$, then the following inequality holds:

$$\sum_{i=1}^{K} 2^{-l_i} \leqslant 1 \tag{2}$$

Example 3.2.3:

Examining the code generated in Example 3.2.1 (Table 3.5), the lengths of the codewords are $\{1, 2, 3, 4, 4\}$. Substituting these values into the left-hand side of Equation (2), we get

$$2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-4} = 1$$

which satisfies the Kraft-McMillan inequality.

If we use the minimum variance code (Table 3.9), the lengths of the codewords are $\{2, 2, 2, 3, 3\}$. Substituting these values into the left-hand side of Equation (2), we get

$$2^{-2} + 2^{-2} + 2^{-2} + 2^{-3} + 2^{-3} = 1$$

which again satisfies the inequality. ♦

The second part of this result, due to Kraft, states that if we have a sequence of positive integers $\{l_i\}_{i=1}^K$ that satisfies (2), then there exists a uniquely decodable code whose codeword lengths are given by the sequence $\{l_i\}_{i=1}^K$.

Using this result, we will now show the following:

1. The average codeword length $\bar{l}$ of an optimal code for a source $\mathcal{S}$ is greater than or equal to $H(\mathcal{S})$.
2. The average codeword length $\bar{l}$ of an optimal code for a source $\mathcal{S}$ is strictly less than $H(\mathcal{S}) + 1$.

For a source $\mathcal{S}$ with alphabet $\mathcal{A} = \{a_1, a_2, \ldots a_K\}$, and probability model $\{P(a_1), P(a_2), \ldots, P(a_K)\}$, the average codeword length is given by

$$\bar{l} = \sum_{i=1}^{K} P(a_i) l_i$$

Therefore, we can write the difference between the entropy of the source $H(\mathcal{S})$ and the average length as

$$\begin{aligned} H(\mathcal{S}) - \bar{l} &= -\sum_{i=1}^{K} P(a_i) \log_2 P(a_i) - \sum_{i=1}^{K} P(a_i) l_i \\ &= \sum_{i=1}^{K} P(a_i) \left(\log_2 \left[\frac{1}{P(a_i)} \right] - l_i \right) \\ &= \sum_{i=1}^{K} P(a_i) \left(\log_2 \left[\frac{1}{P(a_i)} \right] - \log_2 [2^{l_i}] \right) \\ &= \sum_{i=1}^{K} P(a_i) \log_2 \left[\frac{2^{-l_i}}{P(a_i)} \right] \\ &\leq \log_2 \left[\sum_{i=1}^{K} 2^{-l_i} \right] \end{aligned}$$

The last inequality is obtained using Jensen's inequality, which states that if $f(x)$ is a concave (convex cap, convex $\cap$) function, then $E[f(X)] \leqslant f(E[X])$. The log function is a concave function.

As the code is an optimal code $\sum_{i=1}^{K} 2^{-l_i} \leqslant 1$, therefore

$$H(\mathcal{S}) - \bar{l} \leqslant 0 \tag{3}$$

We will prove the upper bound by showing that there exists a uniquely decodable code with average codeword length less than $H(\mathcal{S}) + 1$. Therefore, if we have an optimal code, this code must have an average length that is less than $H(\mathcal{S}) + 1$.

Given a source, alphabet, and probability model as before, define

$$l_i = \left\lceil \log_2 \frac{1}{P(a_i)} \right\rceil$$

where $\lceil x \rceil$ is the smallest integer greater than or equal to x. For example, $\lceil 3.3 \rceil = 4$ and $\lceil 5 \rceil = 5$. Thus,

$$\lceil x \rceil = x + \epsilon \qquad \text{where } 0 \leqslant \epsilon < 1$$

Therefore,

$$\log_2 \frac{1}{P(a_i)} \leqslant l_i < \log_2 \frac{1}{P(a_i)} + 1 \tag{4}$$

From the left inequality of (4), we can see that

$$2^{-l_i} \leqslant P(a_i)$$

Accordingly,

$$\sum_{i=1}^{K} 2^{-l_i} \leqslant \sum_{i=1}^{K} P(a_i) = 1$$

and by the Kraft-McMillan inequality, there exists a uniquely decodable code with codeword lengths $\{l_i\}$. The average length of this code can be upper-bounded by using the right inequality of (4):

$$\bar{l} = \sum_{i=1}^{K} P(a_i) l_i < \sum_{i=1}^{K} P(a_i) \left[\log_2 \frac{1}{P(a_i)} + 1 \right]$$

or

$$\bar{l} < H(\mathcal{S}) + 1$$

We can see from the way the upper bound was derived that this is a rather loose upper bound. In fact, it can be shown that if $p_{\max}$ is the largest probability in the probability model, then for $p_{\max} \geqslant 0.5$, the upper bound for the Huffman code is $H(\mathcal{S}) + p_{\max}$, while for $p_{\max} < 0.5$, the upper bound is $H(\mathcal{S}) + p_{\max} + 0.086$. Obviously, this is a much tighter bound than the one we derived above. The derivation of this bound takes some time (see [21] for details).

3.2.6 Extended Huffman Codes ★

In applications where the alphabet size is large, $p_{\max}$ is generally quite small, and the amount of deviation from the entropy, especially in terms of a percentage of the rate, is quite small. However, in cases where the alphabet is small and the probability of occurrence of the different letters is skewed, the value of $p_{\max}$ can be quite large; and the Huffman code can become rather inefficient when compared to the entropy.

TABLE 3.12 Huffman code for the alphabet $\mathcal{A}$.

Letter	Codeword
a_1	00
a_2	11
a_3	10

Example 3.2.4:

Consider a source that puts out *iid* letters from the alphabet $\mathcal{A} = \{a_1, a_2, a_3\}$ with the probability model $P(a_1) = 0.8$, $P(a_2) = 0.02$, and $P(a_3) = 0.18$. The entropy for this source is 0.816 bits/symbol. A Huffman code for this source is shown in Table 3.12.

The average length for this code is 1.2 bits/symbol. The difference between the average code length and the entropy, or the redundancy, for this code is 0.384 bits/symbol, which is 47% of the entropy. This means that to code this sequence, we would need 47% more bits than the minimum required. ♦

We can sometimes reduce the coding rate by blocking more than one symbol together. To see how this can happen, consider a source S that emits a sequence of letters from an alphabet $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$. Each element of the sequence is generated independently of the other elements in the sequence. The entropy for this source is given by

$$H(S) = -\sum_{i=1}^{m} P(a_i) \log_2 P(a_i)$$

We know that we can generate a Huffman code for this source with rate R such that

$$H(S) \leqslant R < H(S) + 1 \tag{5}$$

We have used the looser bound here; the same argument can be made with the tighter bound. Notice that we have used "rate R" to denote the number of bits per symbol. This is a standard convention in the data compression literature. However, in the communication literature, the word "rate" often refers to the number of bits per second.

Suppose we now encode the sequence by generating one codeword for every n symbols. As there are m^n combinations of n symbols, we will need m^n codewords in our Huffman code. We could generate this code by viewing the m^n symbols as letters of an *extended alphabet*

$$\mathcal{A}^{(n)} = \{\overbrace{a_1 a_1 \ldots a_1}^{\text{n times}}, a_1 a_1 \ldots a_2, \ldots, a_1 a_1 \ldots a_m, a_1 a_1 \ldots a_2 a_1, \ldots, a_m a_m \ldots a_m\}$$

from a source $S^{(n)}$. Let us denote the rate for the new source as $R^{(n)}$. Then we know that

$$H(S^{(n)}) \leqslant R^{(n)} < H(S^{(n)}) + 1 \tag{6}$$

$R^{(n)}$ is the number of bits required to code n symbols. Therefore, the number of bits required per symbol, R, is given by

$$R = \frac{1}{n} R^{(n)}$$

The number of bits per symbol can be bounded as

$$\frac{H(S^{(n)})}{n} \leqslant R < \frac{H(S^{(n)})}{n} + \frac{1}{n}$$

In order to compare this to (5), and see the advantage we get from encoding symbols in blocks instead of one at a time, we need to express $H(S^{(n)})$ in terms of $H(S)$. This turns out to be a relatively easy (although somewhat messy) thing to do.

$$\begin{aligned}
H(S^{(n)}) &= -\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\dots\sum_{i_n=1}^{m} P(a_{i_1}, a_{i_2}, \dots a_{i_n}) \log[P(a_{i_1}, a_{i_2}, \dots a_{i_n})] \\
&= -\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\dots\sum_{i_n=1}^{m} P(a_{i_1})P(a_{i_2})\dots P(a_{i_n}) \log[P(a_{i_1})P(a_{i_2})\dots P(a_{i_n})] \\
&= -\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\dots\sum_{i_n=1}^{m} P(a_{i_1})P(a_{i_2})\dots P(a_{i_n}) \sum_{j=1}^{n}\log[P(a_{i_j})] \\
&= -\sum_{i_1=1}^{m} P(a_{i_1})\log[P(a_{i_1})]\left\{\sum_{i_2=1}^{m}\dots\sum_{i_n=1}^{m} P(a_{i_2})\dots P(a_{i_n})\right\} \\
&\quad -\sum_{i_2=1}^{m} P(a_{i_2})\log[P(a_{i_2})]\left\{\sum_{i_1=1}^{m}\sum_{i_3=1}^{m}\dots\sum_{i_n=1}^{m} P(a_{i_1})P(a_{i_3})\dots P(a_{i_n})\right\} \\
&\quad \cdot \\
&\quad \cdot \\
&\quad \cdot \\
&\quad -\sum_{i_n=1}^{m} P(a_{i_n})\log[P(a_{i_n})]\left\{\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\dots\sum_{i_{n-1}=1}^{m} P(a_{i_1})P(a_{i_2})\dots P(a_{i_{n-1}})\right\}
\end{aligned}$$

The $n-1$ summations in braces in each term sum to one. Therefore,

$$\begin{aligned}
H(S^{(n)}) &= -\sum_{i_1=1}^{m} P(a_{i_1})\log[P(a_{i_1})] - \sum_{i_2=1}^{m} P(a_{i_2})\log[P(a_{i_2})] - \dots - \sum_{i_n=1}^{m} P(a_{i_n})\log[P(a_{i_n})] \\
&= nH(S)
\end{aligned}$$

and we can write (6) as

$$H(S) \leqslant R \leqslant H(S) + \frac{1}{n} \tag{7}$$

By comparing this to (5), we can see that encoding the output of the source in longer blocks of symbols *guarantees* a rate closer to the entropy. Note that all we are talking about here is a bound or guarantee about the rate. As we have seen in the previous chapter, there are a number of situations in which we can achieve a rate *equal* to the entropy with a block length of one!

Example 3.2.5:

For the source described in the previous example, we will generate a codeword for every *two* symbols, instead of generating a codeword for every symbol. If we look at the source sequence two at a time, the number of possible symbol pairs, or size of the extended alphabet, is $3^2 = 9$. The extended alphabet, probability model, and Huffman code for this example are shown in Table 3.13.

TABLE 3.13 The extended alphabet and corresponding Huffman code

Letter	Probability	Code
a_1a_1	0.64	0
a_1a_2	0.016	10101
a_1a_3	0.144	11
a_2a_1	0.016	101000
a_2a_2	0.0004	10100101
a_2a_3	0.0036	1010011
a_3a_1	0.1440	100
a_3a_2	0.0036	10100100
a_3a_3	0.0324	1011

The average codeword length for this extended code is 1.7228 bits/symbol. However, each symbol in the extended alphabet corresponds to two symbols from the original alphabet. Therefore, in terms of the original alphabet, the average codeword length is $1.7228/2 = 0.8614$ bits/symbol. This redundancy is about 0.045 bits/symbol, which is only about 5.5% of the entropy. ♦

We see that by coding blocks of symbols together, we can reduce the redundancy of Huffman codes. In the previous example, two symbols were blocked together to obtain a rate reasonably close to the entropy. Blocking two symbols together means the alphabet size goes from m to m^2, where m was the size of the initial alphabet. In this case, m was three, so the size of the extended alphabet was nine. This size is not an excessive burden for most applications. However, if the probabilities of the symbols were more unbalanced, then it would require blocking many more symbols together before the redundancy was lowered to acceptable levels. As we block more and more symbols together, the size of the alphabet grows exponentially, and the Huffman coding scheme becomes impractical. Under these conditions, we need to look at techniques other than Huffman coding. One approach that is very useful in these conditions is *arithmetic coding*. We will discuss this technique in some detail in the next chapter.

3.2.7 Implementation of Huffman Codes

Huffman encoding is relatively simple to implement using a table lookup. Decoding is another matter. If speed is the only factor, we can implement the decoder using table lookup as well. However, for the decoder, the table has to be of size 2^N where N is the length of the longest

TABLE 3.14 **A nine-letter alphabet and corresponding Huffman code.**

Letter	Code
A	0
B	10101
C	11
D	101000
E	10100101
F	1010011
G	100
H	10100100
I	1011

code. Consider the code in Table 3.13. The longest codeword is eight bits long. This means we need a table of size 256, whereas the code consists of only nine codewords. This is a significant amount of wasted storage. And the waste grows exponentially with the length of the longest codeword. Another option is to store the code as a binary tree. This is certainly memory efficient, however, decoding each symbol requires traversing the tree, which may be computationally inefficient.

Most implementations of Huffman decoding find a middle ground by combining aspects of a table-based decoder and a tree-based decoder. The decoder used in the *inflate* algorithm, which is part of both gzip and zlib, uses such a hybrid method. One way of thinking about the decoder is to view it as a state machine. The state machine takes as its input a specified number of bits, where the number of input bits may be different in each state. Having received these bits, the state machine moves to a new state, and in the process may output a decoded letter. Let's describe the algorithm using an example. Suppose we have a source with the alphabet and code shown in Table 3.14. A tabular implementation of the decoder would require a table of size 256. Instead, we will implement a hierarchy of three tables with the first table being of size 16 and the others being of size 4. Each of the 16 values points to either a decoded letter and the number of bits used to decode the letter (number of bits "to be gobbled" in the inimitable words of Gailly and Adler [22]), or a subsidiary table. The decoding operation can be visualized in terms of the state machine shown in Figure 3.9. In this implementation, the decoding process can be represented using a machine with three states. In state I, the decoder uses a four-bit input as an index into a lookup table; while in state II and state III, the decoder uses a two-bit index into the lookup table. The decoder moves from state I to state II when it receives an input of 1010 and from state II to state III when it receives an input of 01. In the figure, the outcome of other patterns of input are shown as $nnnn/(L, m)$ where $nnnn$ represents the binary index, L denotes the letter decoded, and m indicates the number of bits used in the decoding process. In the figure, x denotes a "don't care," that is, a bit that can be either a 0 or a 1.

The lookup tables corresponding to these states are shown in Tables 3.15–3.17. Notice that in most cases, the letter A is decoded and one bit is used up. This is to be expected as A is the most probable letter with the single bit codeword 0. As eight of the entries in the table begin with 0, it makes sense that eight of the entries will decode to A. After each of these

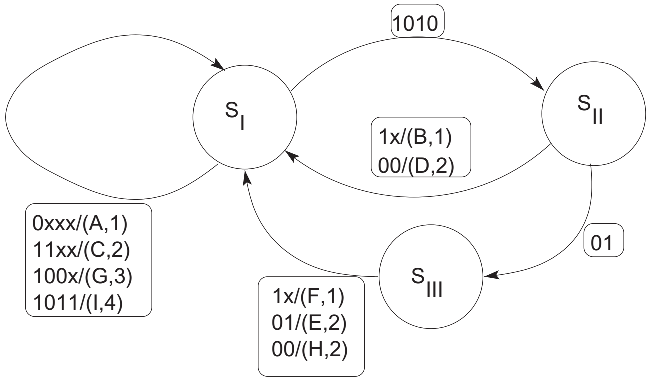

FIGURE 3.9 **A state machine description of Huffman decoding.**

TABLE 3.15 **Table I.**

Letter	Code
0000	$A, 1$
0001	$A, 1$
0010	$A, 1$
0011	$A, 1$
0100	$A, 1$
0101	$A, 1$
0110	$A, 1$
0111	$A, 1$
1000	$G, 3$
1001	$G, 3$
1010	Table II
1011	$I, 4$
1100	$C, 2$
1101	$C, 2$
1110	$C, 2$
1111	$C, 2$

entries is decoded, the decoder shifts out (gobbles up) the leftmost bit and shifts in another bit from the received bitstream before decoding the next letter using the same table. One of the entries, 1010, points to a second table referred to as Table II. In this case, the decoder moves to state II, shifts out all four bits, reads in two new bits, and uses them as a lookup index for Table II shown in Table 3.16. If these two bits are 01 the decoder moves to state III and uses the following two bits from the received bitstream as an index into the lookup table shown in Table 3.17. If these bits are not 01, the decoder uses them as an index into Table 3.16, decodes the corresponding letter, shifts out the number of bits indicated by the table and returns to state I. In state III the decoder decodes the letter indicated by the lookup table, shifts out the corresponding number of bits, and returns to state I.

TABLE 3.16 Table II.

Letter	Code
00	$D, 2$
01	Table III
10	$B, 1$
11	$B, 1$

TABLE 3.17 Table III.

Letter	Code
00	$H, 2$
01	$E, 2$
10	$F, 1$
11	$F, 1$

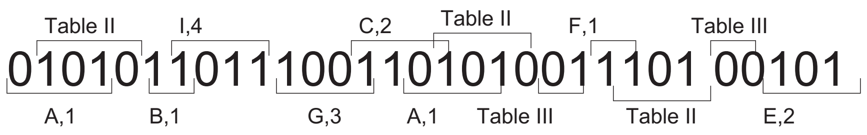

FIGURE 3.10 Decoding of the example binary string.

To see how the decoding process works let's decode the following bitstream:

$$0101011011100110101001110100101$$

We begin in state I with the first four bits 0101. Using this as an index to Table 3.15, we decode the letter A, shift out one bit, and shift in another bit to get a new index of 1010. According to Table 3.15, this is a pointer to Table 3.16, and the decoder moves to state II. Therefore, we read the next two bits from the bitstream and use them as an index to Table 3.16. The next two bits are 11 so we decode a B, shift out a bit, and move back to state I. We then read three more bits to make up the index into Table 3.15. The next three bits are 011 giving us an index of 1011 into Table 3.15. We decode this as an I and read the next four bits, 1001. From Table 3.15, this is decoded as a G for an expenditure of three bits. We read three more bits to get an index of 1101. This decodes as a C with an expenditure of two bits. The process continues as shown in Figure 3.10 resulting in the decoded sequence ABIGCAFE.

We can see in this example that we saved some memory by going from a 256 element lookup table to three lookup tables with 24 entries. In order to obtain these savings, we paid in terms of computation. In the case of the single lookup table, one lookup is sufficient to decode a letter. In this case, when we encounter the letters A, C, G, or I, we use one table lookup. For the letters D and B, we use two table lookups, and for the letters H, E, and F, we need three lookups. Thus, on the average, we need $1 \times (0.64 + 0.144 + 0.144 + 0.0324) + 2 \times (0.016 + 0.016) + 3 \times (0.0036 + 0.0036 + 0.0004) = 1.0472$ lookups per letter. This is only slightly more lookups than we would have needed had we used a single lookup table.

3.3 Nonbinary Huffman Codes ★

The binary Huffman coding procedure can be easily extended to the nonbinary case where the code elements come from an m-ary alphabet, and m is not equal to two. Recall that we obtained the Huffman algorithm based on the following observations about an optimum binary prefix code:

1. Symbols that occur more frequently (have a higher probability of occurrence) will have shorter codewords than symbols that occur less frequently.
2. The two symbols that occur least frequently will have the same length.

Also recall that an additional requirement is that the two symbols with the lowest probability differ only in the last position.

We can obtain a nonbinary Huffman code in almost exactly the same way. The obvious thing to do would be to modify the second observation to read "The m symbols that occur least frequently will have the same length," and also modify the additional requirement to read "The m symbols with the lowest probability differ only in the last position."

However, we run into a small problem with this approach. Consider the design of a ternary Huffman code for a source with a six-letter alphabet. Using the rules described above, we would first combine the three letters with the lowest probability into a composite letter. This would give us a reduced alphabet with four letters. However, combining the three letters with lowest probability from this alphabet would result in a further reduced alphabet consisting of only two letters. We have three values to assign and only two letters. Instead of combining three letters at the beginning, we could have combined two letters. This would result in a reduced alphabet of size five. If we combined three letters from this alphabet, we would end up with a final reduced alphabet size of three. Finally, we could combine two letters in the second step, which would again result in a final reduced alphabet of size three. Which alternative should we choose?

Recall that the symbols with the lowest probability will have the longest codeword. Furthermore, all of the symbols that we combine together into a composite symbol will have codewords of the same length. This means that all letters we combine together at the very first stage will have codewords that have the same length, and these codewords will be the longest of all the codewords. If at some stage we are allowed to combine less than m symbols, the logical place to do this would be in the very first stage.

In the general case of an m-ary code and an M-letter alphabet, how many letters should we combine in the first phase? Let m' be the number of letters that are combined in the first phase. Then m' is the number between two and m, such that m' modulo $(m-1)$ is equal to M modulo $(m-1)$.

Example 3.3.1:

Generate a ternary Huffman code for a source with a six-letter alphabet and a probability model $P(a_1) = P(a_3) = P(a_4) = 0.2$, $P(a_5) = 0.25$, $P(a_6) = 0.1$, and $P(a_2) = 0.05$. In this case $m = 3$, therefore m' is either 2 or 3.

$$6 \pmod 2 = 0, \qquad 2 \pmod 2 = 0, \qquad 3 \pmod 2 = 1$$

TABLE 3.18 Sorted six-letter alphabet.

Letter	Probability	Codeword
a_5	0.25	$c(a_5)$
a_1	0.20	$c(a_1)$
a_3	0.20	$c(a_3)$
a_4	0.20	$c(a_4)$
a_6	0.10	$c(a_6)$
a_2	0.05	$c(a_2)$

TABLE 3.19 Reduced five-letter alphabet.

Letter	Probability	Codeword
a_5	0.25	$c(a_5)$
a_1	0.20	$c(a_1)$
a_3	0.20	$c(a_3)$
a_4	0.20	$c(a_4)$
a'_6	0.15	α_1

Since $6 \pmod 2 = 2 \pmod 2$, $m' = 2$. Sorting the symbols in probability order results in Table 3.18.

As m' is 2, we can assign the codewords of the two symbols with the lowest probability as

$$c(a_6) = \alpha_1 * 0$$
$$c(a_2) = \alpha_1 * 1$$

where α_1 is a ternary string and * denotes concatenation. The reduced alphabet is shown in Table 3.19.

Now we combine the three letters with the lowest probability into a composite letter a'_3 and assign their codewords as

$$c(a_3) = \alpha_2 * 0$$
$$c(a_4) = \alpha_2 * 1$$
$$c(a'_6) = \alpha_2 * 2$$

But $c(a'_6) = \alpha_1$. Therefore,

$$\alpha_1 = \alpha_2 * 2$$

which means that

$$c(a_6) = \alpha_2 * 20$$
$$c(a_2) = \alpha_2 * 21$$

Sorting the reduced alphabet, we have Table 3.20. Thus, $\alpha_2 = 0$, $c(a_5) = 1$, and $c(a_1) = 2$. Substituting for α_2, we get the codeword assignments in Table 3.21.

The tree corresponding to this code is shown in Figure 3.11. Notice that at the lowest level of the tree, we have only two codewords. If we had combined three letters at the first step and combined two letters at a later step, the lowest level would have contained three codewords, and a longer average code length would result (see Problem 7 at the end of this chapter). ♦

TABLE 3.20 Reduced three-letter alphabet.

Letter	Probability	Codeword
a_3'	0.55	α_2
a_5	0.25	$c(a_5)$
a_1	0.20	$c(a_1)$

TABLE 3.21 Ternary code for six-letter alphabet.

Letter	Probability	Codeword
a_1	0.20	2
a_2	0.05	021
a_3	0.20	00
a_4	0.20	01
a_5	0.25	1
a_6	0.10	020

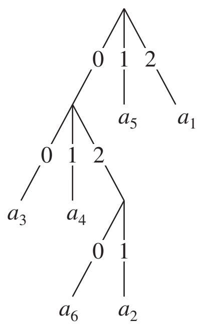

FIGURE 3.11 Code tree for the nonbinary Huffman code.

3.4 Adaptive Huffman Coding

Huffman coding requires knowledge of the probabilities of the source sequence. If this knowledge is not available, Huffman coding becomes a two-pass procedure: the statistics are collected in the first pass, and the source is encoded in the second pass. In order to convert this algorithm into a one-pass procedure, Faller [23] and Gallagher [21] independently developed adaptive algorithms to construct the Huffman code based on the statistics of the symbols already encountered. These were later improved by Knuth [24] and Vitter [25].

Theoretically, if we wanted to encode the (k+1)th symbol using the statistics of the first k symbols, we could recompute the code using the Huffman coding procedure each time a symbol is transmitted. However, this would not be a very practical approach due to the large amount of computation involved—hence, the adaptive Huffman coding procedures.

The Huffman code can be described in terms of a binary tree similar to the ones shown in Figure 3.4. The squares denote the external nodes or leaves and correspond to the symbols in the source alphabet. The codeword for a symbol can be obtained by traversing the tree from the root to the leaf corresponding to the symbol, where 0 corresponds to a left branch and 1 corresponds to a right branch. In order to describe how the adaptive Huffman code works, we add two other parameters to the binary tree: the *weight* of each leaf, which is written as a number inside the node, and a *node number*. The weight of each external node is simply the number of times the symbol corresponding to the leaf has been encountered. The weight of each internal node is the sum of the weights of its offspring. The node number y_i is a unique number assigned to each internal and external node. If we have an alphabet of size n, then the $2n-1$ internal and external nodes can be numbered as $y_1, \ldots, y_{2n-1}$ such that if x_j is the weight of node y_j, we have $x_1 \leqslant x_2 \leqslant \cdots \leqslant x_{2n-1}$. Furthermore, the nodes y_{2j-1} and y_{2j} are offspring of the same parent node, or siblings, for $1 \leqslant j < n$, and the node number for the parent node is greater than y_{2j-1} and y_{2j}. These last two characteristics are called the *sibling property*, and any tree that possesses this property is a Huffman tree [21].

In the adaptive Huffman coding procedure, neither transmitter nor receiver knows anything about the statistics of the source sequence at the start of transmission. The tree at both the transmitter and the receiver consists of a single node that corresponds to all symbols not yet transmitted (NYT) and has a weight of zero. As transmission progresses, nodes corresponding to symbols transmitted are added to the tree, and the tree is reconfigured using an update procedure. Before the beginning of transmission, a fixed code for each symbol is agreed upon between transmitter and receiver. A simple (short) code is as follows:

If the source has an alphabet $(a_1, a_2, \ldots, a_m)$ of size m, then pick e and r such that $m = 2^e + r$ and $0 \leqslant r < 2^e$. The letter a_k is encoded as the $(e+1)$-bit binary representation of $k-1$, if $1 \leqslant k \leqslant 2r$; else, a_k is encoded as the e-bit binary representation of $k-r-1$. For example, suppose $m = 26$, then $e = 4$, and $r = 10$. The symbol a_1 is encoded as 00000, the symbol a_2 is encoded as 00001, and the symbol a_{22} is encoded as 1011.

When a symbol is encountered for the first time, the code for the NYT node is transmitted, followed by the fixed code for the symbol. A node for the symbol is then created, and the symbol is taken out of the NYT list.

Both transmitter and receiver start with the same tree structure. The updating procedure used by both transmitter and receiver is identical. Therefore, the encoding and decoding processes remain synchronized.

3.4.1 Update Procedure

The update procedure requires that the nodes be in a fixed order. This ordering is preserved by numbering the nodes. The largest node number is given to the root of the tree, and the smallest number is assigned to the NYT node. The numbers from the NYT node to the root of the tree are assigned in increasing order from left to right and from lower level to upper level. The set of nodes with the same weight makes up a *block*. Figure 3.12 is a flowchart of the updating procedure.

The function of the update procedure is to preserve the sibling property. In order that the update procedures at the transmitter and receiver both operate with the same information, the

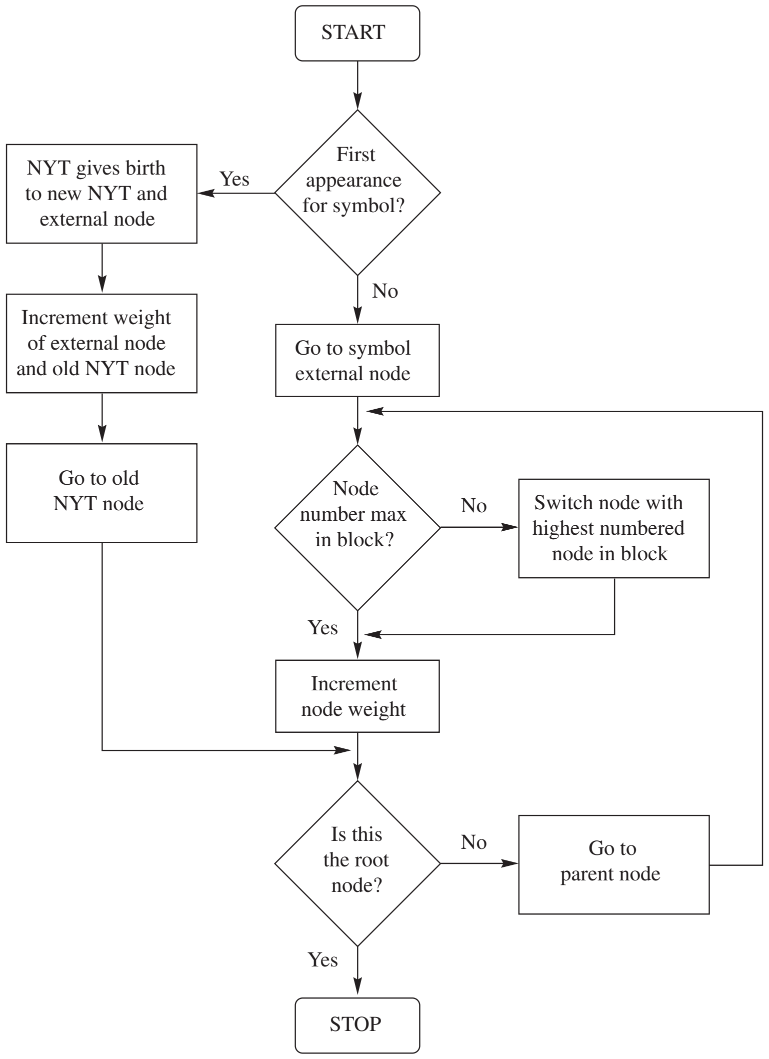
START
First appearance for symbol?
Yes
NYT gives birth to new NYT and external node
No
Increment weight of external node and old NYT node
Go to symbol external node
Go to old NYT node
Node number max in block?
No
Switch node with highest numbered node in block
Yes
Increment node weight
Is this the root node?
No
Go to parent node
Yes
STOP

FIGURE 3.12 Update procedure for the adaptive Huffman coding algorithm.

tree at the transmitter is updated after each symbol is encoded, and the tree at the receiver is updated after each symbol is decoded. The procedure operates as follows.

After a symbol has been encoded or decoded, the external node corresponding to the symbol is examined to see if it has the largest node number in its block. If the external node does not have the largest node number, it is exchanged with the node that has the largest node number in the block, as long as the node with the higher number is not the parent of the node being updated. The weight of the external node is then incremented. If we did not exchange the nodes before the weight of the node was incremented, it is very likely that the ordering required by the sibling property would be destroyed. Once we have incremented the weight of the node, we have adapted the Huffman tree at that level. We then turn our attention to the next level by examining the parent node of the node whose weight was incremented to see if it has the largest number in its block. If it does not, it is exchanged with the node with the largest number in the block. Again, an exception to this is when the node with the higher node number is the parent of the node under consideration. Once an exchange has taken place (or it has been determined that there is no need for an exchange), the weight of the parent node is incremented. We then proceed to a new parent node and the process is repeated. This process continues until the root of the tree is reached.

If the symbol to be encoded or decoded has occurred for the first time, a new external node with a weight of zero is assigned to the symbol and a new NYT node is appended to the tree. Both the new external node and the new NYT node are offsprings of the old NYT node. We increment the weight of the new external node by one. As the old NYT node is the parent of the new external node, we increment its weight by one and then go on to update all the other nodes until we reach the root of the tree.

Example 3.4.1: Update Procedure

Assume we are encoding the message [a a r d v a r k], where our alphabet consists of the 26 lowercase letters of the English alphabet.

The updating process is shown in Figure 3.13. We begin with only the NYT node. The total number of nodes in this tree will be $2 \times 26 - 1 = 51$, so we start numbering backwards from 51 with the number of the root node being 51. The first letter to be transmitted is a. As a does not yet exist in the tree, we send a binary code 00000 for a and then add a to the tree. The NYT node gives birth to a new NYT node and a terminal node corresponding to a. The weight of the terminal node will be higher than the NYT node, so we assign the number 49 to the NYT node and 50 to the terminal node corresponding to the letter a. The second letter to be transmitted is also a. This time the transmitted code is 1. The node corresponding to a has the highest number (if we do not consider its parent), so we do not need to swap nodes. The next letter to be transmitted is r. This letter does not have a corresponding node on the tree, so we send the codeword for the NYT node, which is 0 followed by the index of r, which is 10001. The NYT node gives birth to a new NYT node and an external node corresponding to r. Again, no update is required. The next letter to be transmitted is d, which is also being sent for the first time. We again send the code for the NYT node, which is now 00 followed by the index for d, which is 00011. The NYT node again gives birth to two new nodes. However, an update is still not required. This changes with the transmission of the next letter, v, which has also not yet been encountered. Nodes 43 and 44 are added to the tree, with 44 as the terminal node corresponding to v. We examine the grandparent node of v (node 47) to see if it has the largest number in its block. As it does not, we swap it with node 48, which has the largest number in its block. We then increment node 48 and move to its parent, which is node 49. In the block containing node 49, the largest number belongs to node 50. Therefore, we swap nodes 49 and 50 and then increment node 50. We then move to the parent node of node 50, which is node 51. As this is the root node, all we do is increment node 51. ♦

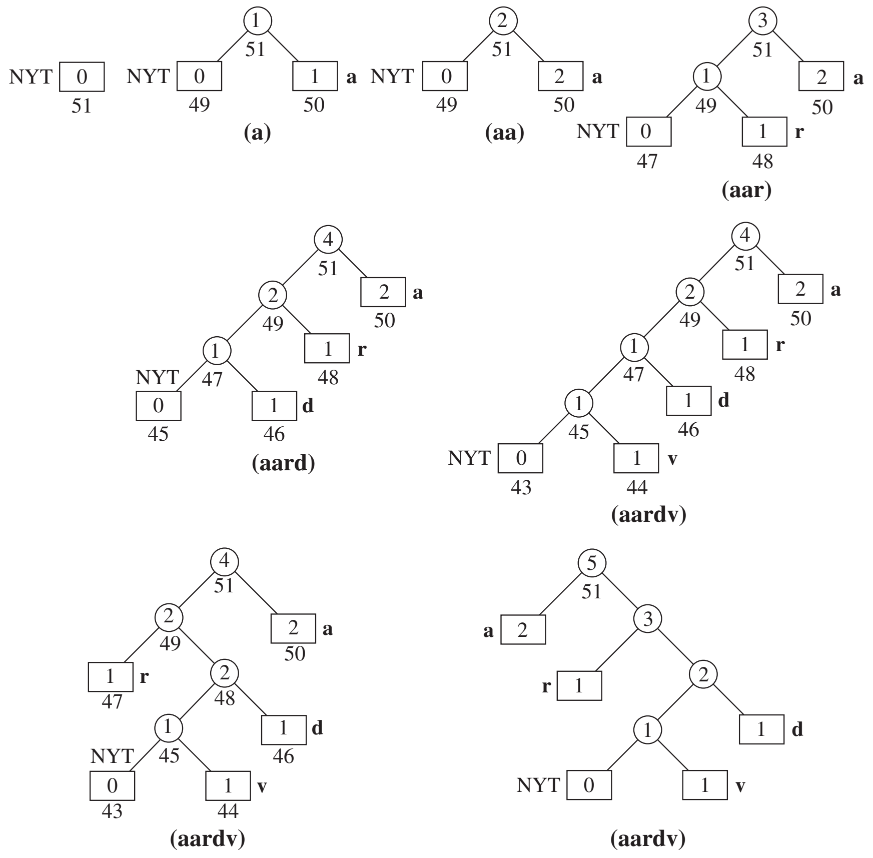

FIGURE 3.13 Adaptive Huffman tree after [a a r d v] is processed.

3.4.2 Encoding Procedure

The flowchart for the encoding procedure is shown in Figure 3.14. Initially, the tree at both the encoder and decoder consists of a single node, the NYT node. Therefore, the codeword for the very first symbol that appears is a previously agreed-upon fixed code. After the very first symbol, whenever we have to encode a symbol that is being encountered for the first time, we send the code for the NYT node, followed by the previously agreed-upon fixed code for the symbol. The code for the NYT node is obtained by traversing the Huffman tree from the root to the NYT node. This alerts the receiver to the fact that the symbol whose code follows does not as yet have a node in the Huffman tree. If a symbol to be encoded has a corresponding node in the tree, then the code for the symbol is generated by traversing the tree from the root to the external node corresponding to the symbol.

To see how the coding operation functions, we use the same example that was used to demonstrate the update procedure.

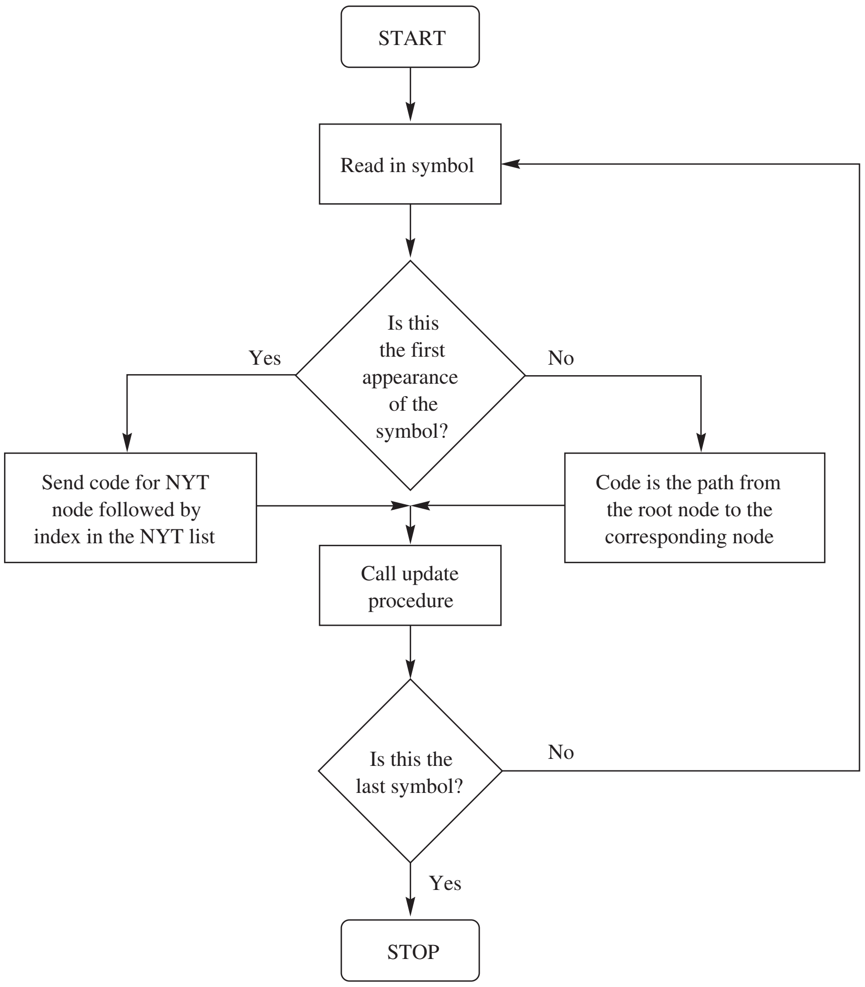
START
Read in symbol
Is this the first appearance of the symbol?
Yes
No
Send code for NYT node followed by index in the NYT list
Code is the path from the root node to the corresponding node
Call update procedure
Is this the last symbol?
No
Yes
STOP

FIGURE 3.14 Flowchart of the encoding procedure.

Example 3.4.2: Encoding Procedure

In Example 3.4.1 we used an alphabet consisting of 26 letters. In order to obtain our prearranged code, we have to find m and e such that $2^e + r = 26$, where $0 \leqslant r < 2^e$. It is easy to see that the values of $e = 4$ and $r = 10$ satisfy this requirement.

The first symbol encoded is the letter a. As a is the first letter of the alphabet, $k = 1$. As 1 is less than 20, a is encoded as the 5-bit binary representation of $k - 1$, or 0, which is 00000. The Huffman tree is then updated as shown in the figure. The NYT node gives birth to an external node corresponding to the element a and a new NYT node. As a has occurred once, the external node corresponding to a has a weight of one. The weight of the NYT node is zero. The internal node also has a weight of one, as its weight is the sum of the weights of its offspring. The next symbol is again a. As we have an external node corresponding to symbol a, we simply traverse the tree from the root node to the external node corresponding to a in order to find the codeword. This traversal consists of a single right branch. Therefore, the Huffman code for the symbol a is 1.

After the code for a has been transmitted, the weight of the external node corresponding to a is incremented, as is the weight of its parent. The third symbol to be transmitted is r. As this is the first appearance of this symbol, we send the code for the NYT node followed by the previously arranged binary representation for r. If we traverse the tree from the root to the NYT node, we get a code of 0 for the NYT node. The letter r is the 18th letter of the alphabet; therefore, the binary representation of r is 10001. The code for the symbol r becomes 010001. The tree is again updated as shown in the figure, and the coding process continues with symbol d. Using the same procedure for d, the code for the NYT node, which is now 00, is sent, followed by the index for d, resulting in the codeword 0000011. The next symbol v is the 22nd symbol in the alphabet. As this is greater than 20, we send the code for the NYT node followed by the 4-bit binary representation of $22 - 10 - 1 = 11$. The code for the NYT node at this stage is 000, and the 4-bit binary representation of 11 is 1011; therefore, v is encoded as 0001011. The next symbol is a, for which the code is 0, and the encoding proceeds. ♦

3.4.3 Decoding Procedure

The flowchart for the decoding procedure is shown in Figure 3.15. As we read in the received binary string, we traverse the tree in a manner identical to that used in the encoding procedure. Once a leaf is encountered, the symbol corresponding to that leaf is decoded. If the leaf is the NYT node, then we check the next e bits to see if the resulting number is less than r. If it is less than r, we read in another bit to complete the code for the symbol. The index for the symbol is obtained by adding one to the decimal number corresponding to the e- or $e + 1$-bit binary string. Once the symbol has been decoded, the tree is updated and the next received bit is used to start another traversal down the tree. To see how this procedure works, let us decode the binary string generated in the previous example.

Example 3.4.3: Decoding Procedure

The binary string generated by the encoding procedure is

$$000001010001000001100010110$$

Initially, the decoder tree consists only of the NYT node. Therefore, the first symbol to be decoded must be obtained from the NYT list. We read in the first 4 bits, 0000, as the value of e is four. The 4 bits 0000 correspond to the decimal value of 0. As this is less than the value of r, which is 10, we read in one more bit for the entire code of 00000. Adding one to the decimal value corresponding to this binary string, we get the index of the received symbol as 1. This is the index for a; therefore, the first letter is decoded as a. The tree is now updated as shown in Figure 3.13. The next bit in the string is 1. This traces a path from the root node to the external node corresponding to a. We decode the symbol a and update the tree. In this case, the update consists only of incrementing the weight of the external node corresponding

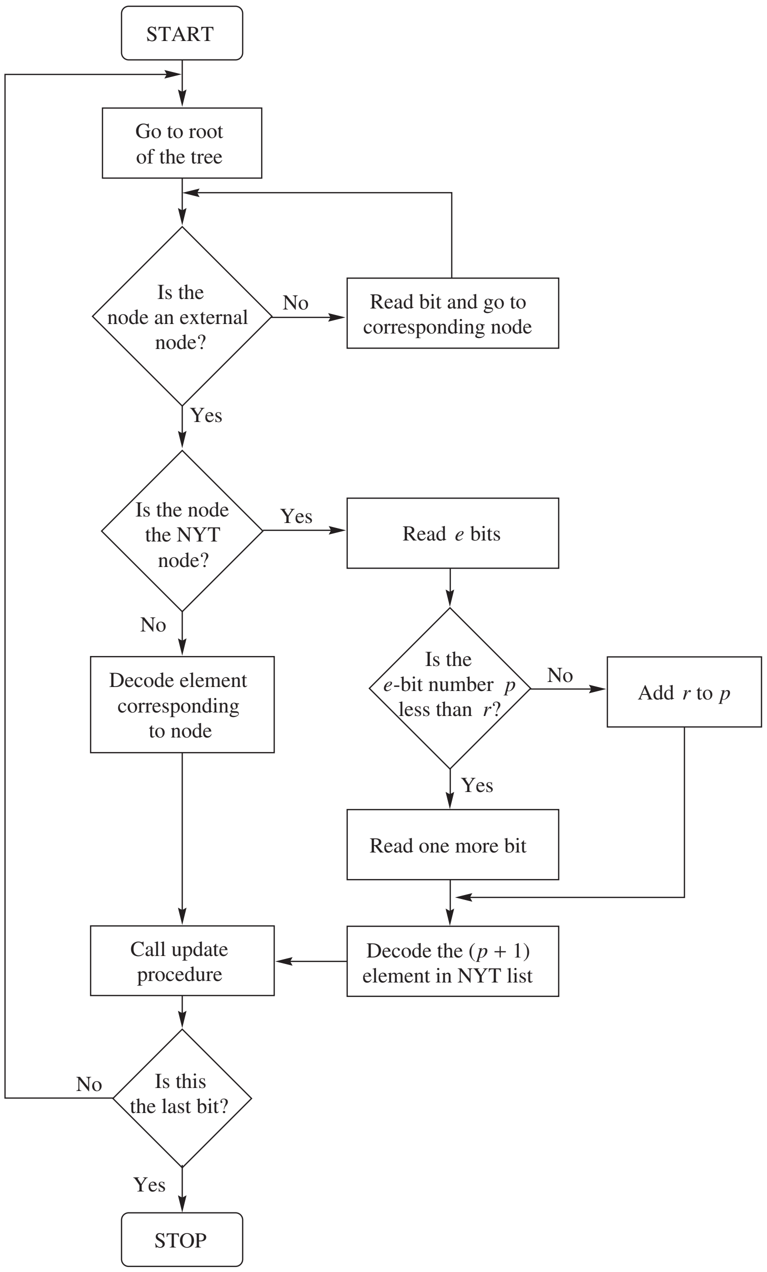
START
Go to root of the tree
Is the node an external node?
No
Read bit and go to corresponding node
Yes
Is the node the NYT node?
Yes
Read e bits
No
Decode element corresponding to node
Is the e-bit number p less than r?
No
Add r to p
Yes
Read one more bit
Call update procedure
Decode the $(p + 1)$ element in NYT list
No
Is this the last bit?
Yes
STOP

FIGURE 3.15 **Flowchart of the decoding procedure.**

to a. The next bit is a 0, which traces a path from the root to the NYT node. The next 4 bits, 1000, correspond to the decimal number 8, which is less than 10, so we read in one more bit to get the 5-bit word 10001. The decimal equivalent of this 5-bit word plus one is 18, which is the index for r. We decode the symbol r and then update the tree. The next 2 bits, 00, again trace a path to the NYT node. We read the next 4 bits, 0001. Since this corresponds to the decimal number 1, which is less than 10, we read another bit to get the 5-bit word 00011. To get the index of the received symbol in the NYT list, we add one to the decimal value of this 5-bit word. The value of the index is 4, which corresponds to the symbol d. Continuing in this fashion, we decode the sequence $aardva$. ♦

Although the Huffman coding algorithm is one of the best-known variable-length coding algorithms, there are some other lesser-known algorithms that can be very useful in certain situations. In particular, the Golomb-Rice codes and the Tunstall codes are becoming increasingly popular. We describe these codes in the following sections.

3.5 Golomb Codes

The Golomb-Rice codes belong to a family of codes designed to encode integers with the assumption that the larger an integer, the lower its probability of occurrence. The simplest code for this situation is the *unary* code. The unary code for a positive integer n is simply n 1s followed by a 0. Thus, the code for 4 is 11110, and the code for 7 is 11111110. The unary code is the same as the Huffman code for the semi-infinite alphabet $\{1, 2, 3, \ldots\}$ with probability model

$$P[k] = \frac{1}{2^k}$$

Because the Huffman code is optimal, the unary code is also optimal for this probability model.

Although the unary code is optimal in very restricted conditions, we can see that it is certainly very simple to implement. One step higher in complexity are a number of coding schemes that split the integer into two parts, representing one part with a unary code and the other part with a different code. An example of such a code is the Golomb code. Other examples can be found in [26].

The Golomb code is described in a succinct paper [27] by Solomon Golomb, which begins “Secret Agent 00111 is back at the Casino again, playing a game of chance, while the fate of mankind hangs in the balance.” Agent 00111 requires a code to represent runs of success in a roulette game, and Golomb provides it! The Golomb code is actually a family of codes parameterized by an integer $m > 0$. In the Golomb code with parameter m, we represent an integer $n > 0$ using two numbers q and r, where

$$q = \left\lfloor \frac{n}{m} \right\rfloor$$

and

$$r = n - qm$$

TABLE 3.22 Golomb code for $m = 5$.

n	q	r	Codeword	n	q	r	Codeword
0	0	0	0000	08	1	3	101100
1	0	1	0010	09	1	4	101110
2	0	2	0100	10	2	0	110000
3	0	3	0110	11	2	1	110010
4	0	4	0111	12	2	2	110100
5	1	0	1000	13	2	3	110110
6	1	1	1001	14	2	4	110111
7	1	2	1010	15	3	0	111000

$\lfloor x \rfloor$ is the integer part of x. In other words, q is the quotient and r is the remainder when n is divided by m. The quotient q can take on values 0, 1, 2, ... and is represented by the unary code of q. The remainder r can take on the values $0, 1, 2, \ldots, m-1$. If m is a power of two, we use the $\log_2 m$-bit binary representation of r. If m is not a power of two, we could still use $\lceil \log_2 m \rceil$ bits, where $\lceil x \rceil$ is the smallest integer greater than or equal to x. We can reduce the number of bits required if we use the $\lfloor \log_2 m \rfloor$-bit binary representation of r for the first $2^{\lceil \log_2 m \rceil} - m$ values, and the $\lceil \log_2 m \rceil$-bit binary representation of $r + 2^{\lceil \log_2 m \rceil} - m$ for the rest of the values.

Example 3.5.1: Golomb Code

Let's design a Golomb code for $m = 5$. As

$$\lceil \log_2 5 \rceil = 3 \quad \text{and} \quad \lfloor \log_2 5 \rfloor = 2$$

the first $8 - 5 = 3$ values of r (that is, $r = 0, 1, 2$) will be represented by the 2-bit binary representation of r, and the next two values (that is, $r = 3, 4$) will be represented by the 3-bit representation of $r + 3$. The quotient q is always represented by the unary code for q. Thus, the codeword for 3 is 0110, and the codeword for 21 is 1111001. The codewords for $n = 0$, ..., 15 are shown in Table 3.22. ♦

It can be shown that the Golomb code is optimal for the probability model

$$P(n) = p^{n-1}q, \qquad q = 1 - p$$

when

$$m = \left\lceil -\frac{1}{\log_2 p} \right\rceil .$$

3.6 Rice Codes

The Rice code was originally developed by Robert F. Rice (he called it the Rice machine) [28,29] and later extended by Pen-Shu Yeh and Warner Miller [30]. The Rice code can be viewed as an adaptive Golomb code. In the Rice code, a sequence of nonnegative integers (which might have been obtained from the preprocessing of other data) is divided into blocks of J integers apiece. Each block is then coded using one of several options, most of which are a form of Golomb codes. Each block is encoded with each of these options, and the option resulting in the least number of coded bits is selected. The particular option used is indicated by an identifier attached to the code for each block.

The easiest way to understand the Rice code is to examine one of its implementations. We will study the implementation of the Rice code in the recommendation for lossless compression from the Consultative Committee on Space Data Standards (CCSDS).

3.6.1 CCSDS Recommendation for Lossless Compression

As an application of the Rice algorithm, let's briefly look at the algorithm for lossless data compression recommended by CCSDS. The algorithm consists of a preprocessor (modeling step) and a binary coder (coding step). The preprocessor removes correlation from the input and generates a sequence of nonnegative integers. This sequence has the property that smaller values are more probable than larger values. The binary coder generates a bitstream to represent the integer sequence. The binary coder is our main focus at this point.

The preprocessor functions as follows: given a sequence $\{y_i\}$, for each y_i we generate a prediction $\hat{y}_i$. A simple way to generate a prediction would be to take the previous value of the sequence to be a prediction of the current value of the sequence:

$$\hat{y}_i = y_{i-1}$$

We will look at more sophisticated ways of generating a prediction in Chapter 7. We then generate a sequence whose elements are the difference between y_i and its predicted value $\hat{y}_i$:

$$d_i = y_i - \hat{y}_i$$

The d_i value will have a small magnitude when our prediction is good and a large value when it is not. Assuming an accurate modeling of the data, the former situation is more likely than the latter. Let $y_{\max}$ and $y_{\min}$ be the largest and smallest values that the sequence $\{y_i\}$ takes on. It is reasonable to assume that the value of $\hat{y}_i$ will be confined to the range $[y_{\min}, y_{\max}]$. Define

$$T_i = \min\{y_{\max} - \hat{y}_i, \hat{y}_i - y_{\min}\} \tag{8}$$

The sequence $\{d_i\}$ can be converted into a sequence of nonnegative integers $\{x_i\}$ using the following mapping:

$$x_i = \begin{cases} 2d_i & 0 \leqslant d_i \leqslant T_i \\ 2|d_i| - 1 & -T_i \leqslant d_i < 0 \\ T_i + |d_i| & otherwise \end{cases} \tag{9}$$

The value of x_i will be small whenever the magnitude of d_i is small. Therefore, the value of x_i will be small with higher probability. The sequence $\{x_i\}$ is divided into segments with each segment being further divided into blocks of size J. It is recommended by CCSDS that J have a value of 16. Each block is then coded using one of the following options. The coded block is transmitted along with an identifier that indicates which particular option was used.

TABLE 3.23 Code used for zero block option.

Number of All-Zero Blocks	Codeword
1	1
2	01
3	001
4	0001
5	000001
6	0000001
$\vdots$	$\vdots$
63	$\overbrace{000\cdots0}^{63\ 0\text{s}}1$
ROS	00001

- **Fundamental sequence:** This is a unary code. A number n is represented by a sequence of n 0s followed by a 1 (or a sequence of n 1s followed by a 0).
- **Split sample options:** These options consist of a set of codes indexed by a parameter m. The code for a k-bit number n using the mth split sample option consists of the m least significant bits of k followed by a unary code representing the $k - m$ most significant bits. For example, suppose we wanted to encode the 8-bit number 23 using the third split sample option. The 8-bit representation of 23 is 00010111. The three least significant bits are 111. The remaining bits (00010) correspond to the number 2, which has a unary code 001. Therefore, the code for 23 using the third split sample option is 111011. Notice that different values of m will be preferable for different values of x_i, with higher values of m used for higher-entropy sequences.
- **Second extension option:** The second extension option is useful for sequences with low entropy—when, in general, many of the values of x_i will be zero. In the second extension option, the sequence is divided into consecutive pairs of samples. Each pair is used to obtain an index γ using the following transformation:

$$\gamma = \frac{1}{2}(x_i + x_{i+1})(x_i + x_{i+1} + 1) + x_{i+1} \tag{10}$$

  and the value of γ is encoded using a unary code. The value of γ is an index to a lookup table with each value of γ corresponding to a pair of values x_i, x_{i+1}.
- **Zero block option:** The zero block option is used when one or more of the blocks of x_i are zero—generally when we have long sequences of y_i that have the same value. In this case the number of zero blocks are transmitted using the code shown in Table 3.23. The ROS code is used when the last five or more blocks in a segment are all zero.

The Rice code has been used in several space applications, and variations of the Rice code have been proposed for a number of different applications.

TABLE 3.24 A 2-bit Tunstall code.

Sequence	Codeword
AAA	00
AAB	01
AB	10
B	11

3.7 Tunstall Codes

Most of the variable-length codes that we look at in this book encode letters from the source alphabet using codewords with varying numbers of bits: codewords with fewer bits for letters that occur more frequently and codewords with more bits for letters that occur less frequently. The Tunstall code is an important exception. In the Tunstall code, all codewords are of equal length. However, each codeword represents a different number of letters. An example of a 2-bit Tunstall code for an alphabet $\mathcal{A} = \{A, B\}$ is shown in Table 3.24. The main advantage of a Tunstall code is that errors in codewords do not propagate, unlike other variable-length codes, such as Huffman codes, in which an error in one codeword will cause a series of errors to occur.

Example 3.7.1:

Let's encode the sequence $AAABAABAABAABAAA$ using the code in Table 3.24. Starting at the left, we can see that the string AAA occurs in our codebook and has a code of 00. We then code B as 11, AAB as 01, and so on. We finally end up with the code 001101010100 for the sequence. ♦

The design of a code that has a fixed codeword length but a variable number of symbols per codeword should satisfy the following conditions:

1. We should be able to parse a source output sequence into sequences of symbols that appear in the codebook.
2. We should maximize the average number of source symbols represented by each codeword.

In order to understand what we mean by the first condition, consider the code shown in Table 3.25. Let's encode the same sequence $AAABAABAABAABAAA$ as in the previous example using the code in Table 3.25. We first encode AAA with the code 00. We then encode B with 11. The next three symbols are AAB. However, there are no codewords corresponding to this sequence of symbols. Thus, this sequence is unencodable using this particular code—not a desirable situation.

Tunstall [31] gives a simple algorithm that fulfills these conditions. The algorithm is as follows:

Suppose we want an n-bit Tunstall code for a source that generates *iid* letters from an alphabet of size N. The number of codewords is 2^n. We start with the N letters of the source

TABLE 3.25 A 2-bit (non-Tunstall) code.

Sequence	Codeword
AAA	00
ABA	01
AB	10
B	11

alphabet in our codebook. Remove the entry in the codebook that has the highest probability and add the N strings obtained by concatenating this letter with every letter in the alphabet (including itself). This will increase the size of the codebook from N to $N(N-1)$. The probabilities of the new entries will be the product of the probabilities of the letters concatenated to form the new entry. Now look through the $N+(N-1)$ entries in the codebook and find the entry that has the highest probability, keeping in mind that the entry with the highest probability may be a concatenation of symbols. Each time we perform this operation we increase the size of the codebook by $N-1$. Therefore, this operation can be performed K times, where

$$N + K(N-1) \leqslant 2^n$$

Example 3.7.2: Tunstall Codes

Let us design a 3-bit Tunstall code for a memoryless source with the following alphabet:

$$\mathcal{A} = \{A, B, C\}$$
$$P(A) = 0.6, \quad P(B) = 0.3, \quad P(C) = 0.1$$

We start out with the codebook and associated probabilities shown in Table 3.26. Since the letter A has the highest probability, we remove it from the list and add all two-letter strings beginning with A as shown in Table 3.27. After one iteration, we have five entries in our codebook. Going through one more iteration will increase the size of the codebook by two, and we will have seven entries, which is still less than the final codebook size. Going through another iteration after that would bring the codebook size to nine, which is greater than the maximum size of eight. Therefore, we will go through just one more iteration. Looking through the entries in Table 3.27, the entry with the highest probability is AA. Therefore, at the next step we remove AA and add all extensions of AA as shown in Table 3.28. The final 3-bit Tunstall code is shown in Table 3.28. ♦

TABLE 3.26 Source alphabet and associated probabilities.

Letter	Probability
A	0.60
B	0.30
C	0.10

TABLE 3.27 The codebook after one iteration.

Sequence	Probability
B	0.30
C	0.10
AA	0.36
AB	0.18
AC	0.06

TABLE 3.28 A 3-bit Tunstall code.

Sequence	Code
B	000
C	001
AB	010
AC	011
AAA	100
AAB	101
AAC	110

3.8 Applications of Huffman Coding

In this section, we describe some applications of Huffman coding. As we progress through the book, we will describe more applications, since Huffman coding is often used in conjunction with other coding techniques.

3.8.1 Lossless Image Compression

A simple application of Huffman coding to image compression would be to generate a Huffman code for the set of values that any pixel may take. For monochrome images, this set usually consists of integers from 0 to 255. Examples of such images are contained in the accompanying data sets. The four that we will use in the examples in this book are shown in Figure 3.16.

We will make use of one of the programs from the accompanying software (see Preface) to generate a Huffman code for each image and then encode the image using the Huffman code. The results for the four images in Figure 3.16 are shown in Table 3.29. The Huffman code is stored along with the compressed image as the code will be required by the decoder to reconstruct the image.

The original (uncompressed) image representation uses 8 bits/pixel. The image consists of 256 rows of 256 pixels, so the uncompressed representation uses 65,536 bytes. The compression ratio is simply the ratio of the number of bytes in the uncompressed representation to the number of bytes in the compressed representation. The number of bytes in the compressed representation includes the number of bytes needed to store the Huffman code. Notice that the compression ratio is different for different images. This can cause some problems in

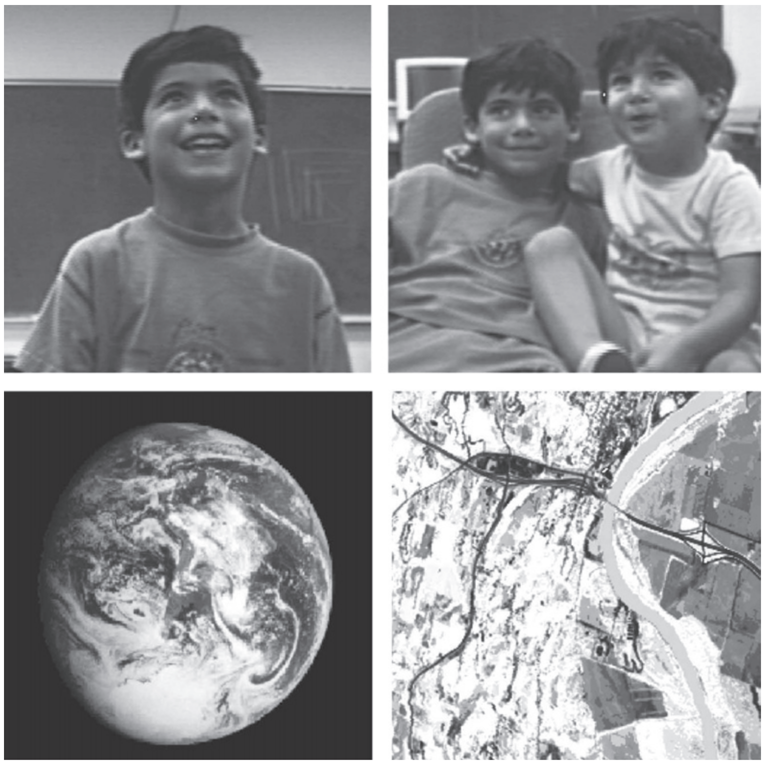

FIGURE 3.16 Test images.

TABLE 3.29 Compression using Huffman codes on pixel values.

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio
Sena	7.01	57,504	1.14
Sensin	7.49	61,430	1.07
Earth	4.94	40,534	1.62
Omaha	7.12	58,374	1.12

certain applications where it is necessary to know in advance how many bytes will be needed to represent a particular data set.

The results in Table 3.29 are somewhat disappointing because we get a reduction of only about $\frac{1}{2}$ to 1 bit/pixel after compression. For some applications, this reduction is acceptable. For example, if we were storing hundreds of thousands of images in an archive, a reduction of 1 bit/pixel saves many gigabytes in disk space. However, we can do better. Recall that when we first talked about compression, we said that the first step for any compression algorithm was to model the data so as to make use of the structure in the data. In this case, we have made absolutely no use of the structure in the data.

TABLE 3.30 Compression using Huffman codes on pixel difference values.

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio
Sena	4.02	32,968	1.99
Sensin	4.70	38,541	1.70
Earth	4.13	33,880	1.93
Omaha	6.42	52,643	1.24

TABLE 3.31 Compression using adaptive Huffman codes on pixel difference values.

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio
Sena	3.93	32,261	2.03
Sensin	4.63	37,896	1.73
Earth	4.82	39,504	1.66
Omaha	6.39	52,321	1.25

From a visual inspection of the test images, we can clearly see that the pixels in an image are heavily correlated with their neighbors. We could represent this structure with the crude model $\hat{x}_n = x_{n-1}$. The residual would be the difference between neighboring pixels. If we carry out this differencing operation and use the Huffman coder on the residuals, the results are as shown in Table 3.30. As we can see, using the structure in the data resulted in substantial improvement.

The results in Tables 3.29 and 3.30 were obtained using a two-pass system, in which the statistics were collected in the first pass and a Huffman table was generated using these statistics. The image was then encoded in the second pass. Instead of using a two-pass system, we could have used a one-pass adaptive Huffman coder. The results for this are given in Table 3.31.

Notice that there is little difference between the performance of the adaptive Huffman code and the two-pass Huffman coder. In addition, the fact that the adaptive Huffman coder can be used as an online or real-time coder makes the adaptive Huffman coder a more attractive option in many applications. However, the adaptive Huffman coder is more vulnerable to errors and may also be more difficult to implement. In the end, the particular application will determine which approach is more suitable.

3.8.2 Text Compression

Text compression seems natural for Huffman coding. In text, we have a discrete alphabet that, in a given class, has relatively stationary probabilities. For example, the probability model for a particular novel will not differ significantly from the probability model for another novel. Similarly, the probability model for a set of C programs is not going to be much different than

TABLE 3.32 **Probabilities of occurrence of the letters in the English alphabet in the U.S. Constitution.**

Letter	Probability	Letter	Probability
A	0.057305	N	0.056035
B	0.014876	O	0.058215
C	0.025775	P	0.021034
D	0.026811	Q	0.000973
E	0.112578	R	0.048819
F	0.022875	S	0.060289
G	0.009523	T	0.078085
H	0.042915	U	0.018474
I	0.053475	V	0.009882
J	0.002031	W	0.007576
K	0.001016	X	0.002264
L	0.031403	Y	0.011702
M	0.015892	Z	0.001502

TABLE 3.33 **Probabilities of occurrence of the letters in the English alphabet in this chapter.**

Letter	Probability	Letter	Probability
A	0.049855	N	0.048039
B	0.016100	O	0.050642
C	0.025835	P	0.015007
D	0.030232	Q	0.001509
E	0.097434	R	0.040492
F	0.019754	S	0.042657
G	0.012053	T	0.061142
H	0.035723	U	0.015794
I	0.048783	V	0.004988
J	0.000394	W	0.012207
K	0.002450	X	0.003413
L	0.025835	Y	0.008466
M	0.016494	Z	0.001050

the probability model for a different set of C programs. The probabilities in Table 3.32 are the probabilities of the 26 letters (upper- and lowercase) obtained for the U.S. Constitution and are representative of English text. The probabilities in Table 3.33 were obtained by counting the frequency of occurrences of letters in an earlier version of this chapter. While the two documents are substantially different, the two sets of probabilities are very much alike.

We encoded the earlier version of this chapter using Huffman codes that were created using the probabilities of occurrence obtained from the chapter. The file size dropped from about 70,000 bytes to about 43,000 bytes with Huffman coding.

While this reduction in file size is useful, we could have obtained better compression if we had first removed the structure existing in the form of correlation between the symbols in the file. Obviously, there is a substantial amount of correlation in this text. For example, *Huf* is always followed by *fman*! Unfortunately, this correlation is not amenable to simple numerical models, as was the case for the image files. However, there are other somewhat more complex techniques that can be used to remove the correlation in text files. We will look more closely at these in Chapters 5 and 6.

3.8.3 Audio Compression

Another class of data that is very suitable for compression is CD-quality audio data. The audio signal for each stereo channel is sampled at 44.1 kHz, and each sample is represented by 16 bits. This means that the amount of data stored on one CD is enormous. If we want to transmit this data, the amount of channel capacity required would be significant. Compression is definitely useful in this case. In Table 3.34 we show, for a variety of audio material, the file size, the entropy, the estimated compressed file size if a Huffman coder is used, and the resulting compression ratio.

The three segments used in this example represent a wide variety of audio material, from a symphonic piece by Mozart to a folk rock piece by Cohn. Even though the material is varied, Huffman coding can lead to some reduction in the capacity required to transmit this material.

Note that we have only provided the *estimated* compressed file sizes. The estimated file size in bits was obtained by multiplying the entropy by the number of samples in the file. We used this approach because the samples of 16-bit audio can take on 65,536 distinct values, and, therefore, the Huffman coder would require 65,536 distinct (variable-length) codewords. In most applications, a codebook of this size would not be practical. There is a way of handling large alphabets, called recursive indexing, that we will describe in Chapter 9. There is also some recent work [11] on using a Huffman tree in which leaves represent sets of symbols with the same probability. The codeword consists of a prefix that specifies the set followed by a suffix that specifies the symbol within the set. This approach can accommodate relatively large alphabets.

As with the other applications, we can obtain an increase in compression if we first remove the structure from the data. Audio data can be modeled numerically. In later chapters we will examine more sophisticated modeling approaches. For now, let us use the very simple model that was used in the image-coding example; that is, each sample has the same value as the previous sample. Using this model, we obtain the difference sequence. The entropy of the difference sequence is shown in Table 3.35.

TABLE 3.34 Huffman coding of 16-bit CD-quality audio.

File Name	Original File Size (bytes)	Entropy (bits)	Estimated Compressed File Size (bytes)	Compression Ratio
Mozart	939,862	12.8	725,420	1.30
Cohn	402,442	13.8	349,300	1.15
Mir	884,020	13.7	759,540	1.16

TABLE 3.35 Huffman coding of differences of 16-bit CD-quality audio.

File Name	Original File Size (bytes)	Entropy of Differences (bits)	Estimated Compressed File Size (bytes)	Compression Ratio
Mozart	939,862	09.7	569,792	1.65
Cohn	402,442	10.4	261,590	1.54
Mir	884,020	10.9	602,240	1.47

Note that there is a further reduction in the file size: the compressed file sizes are about 60% of the original files. Further reductions can be obtained by using more sophisticated models.

Many of the lossless audio compression schemes, including FLAC (Free Lossless Audio Codec), Apple's ALAC or ALE, *Shorten* [32], *Monkey's Audio*, and the MPEG-4 ALS [33] algorithms, use a linear predictive model to remove some of the structure from the audio sequence and then use Rice coding to encode the residuals. Most others, such as *AudioPak* [34] and *OggSquish*, use Huffman coding to encode the residuals.

3.9 Summary

In this chapter we began our exploration of data compression techniques with a description of the Huffman coding technique and several other related techniques. The Huffman coding technique and its variants are some of the most commonly used coding approaches. We will encounter modified versions of Huffman codes when we look at compression techniques for text, image, and video. In this chapter we described how to design Huffman codes and discussed some of the issues related to Huffman codes. We also described how adaptive Huffman codes work and looked briefly at some of the places where Huffman codes are used. We will see more of these in future chapters.

To explore further applications of Huffman coding, you can use the programs `huff_enc`, `huff_dec`, and `adap_huff` to generate your own Huffman codes for your favorite applications.

Further Reading

1. A detailed and very accessible overview of Huffman codes is provided in "Huffman Coding," by S. Pigeon [35], in *Lossless Compression Handbook*.
2. Details about nonbinary Huffman codes and a much more theoretical and rigorous description of variable-length codes can be found in *The Theory of Information and Coding*, Volume 3 of *Encyclopedia of Mathematics and Its Application*, by R.J. McEliece [9].
3. The tutorial article "Data Compression" in the September 1987 issue of *ACM Computing Surveys*, by D.A. Lelewer and D.S. Hirschberg [36], along with other material, provides a very nice brief coverage of the material in this chapter.

4. A somewhat different approach to describing Huffman codes can be found in *Data Compression—Methods and Theory*, by J.A. Storer [37].
5. A more theoretical but very readable account of variable-length coding can be found in *Elements of Information Theory*, by T.M. Cover and J.A. Thomas [38].
6. Although the book *Coding and Information Theory*, by R.W. Hamming [5], is mostly about channel coding, Huffman codes are described in some detail in Chapter 4.

3.10 Projects and Problems

1. The probabilities in Tables 3.32 and 3.33 were obtained using the program `countalpha` from the accompanying software. Use this program to compare probabilities for different types of text, C programs, messages on Internet forums, and so on. Comment on any differences you might see and describe how you would tailor your compression strategy for each type of text.
2. Use the programs `huff_enc` and `huff_dec` to do the following (in each case use the codebook generated by the image being compressed):
   (a) Code the Sena, Sensin, and Omaha images.
   (b) Write a program to take the difference between adjoining pixels, and then use `huffman` to code the difference images.
   (c) Repeat (a) and (b) using `adap_huff`.

   Report the resulting file sizes for each of these experiments and comment on the differences.
3. Using the programs `huff_enc` and `huff_dec`, code the Bookshelf1 and Sena images using the codebook generated by the Sensin image. Compare the results with the case where the codebook was generated by the image being compressed.
4. A source emits letters from an alphabet $\mathcal{A} = \{a_1, a_2, a_3, a_4, a_5\}$ with probabilities $P(a_1) = 0.15$, $P(a_2) = 0.04$, $P(a_3) = 0.26$, $P(a_4) = 0.05$, and $P(a_5) = 0.50$.
   (a) Calculate the entropy of this source.
   (b) Find a Huffman code for this source. Item(c)Find the average length of the code in (b) and its redundancy.
5. For an alphabet $\mathcal{A} = \{a_1, a_2, a_3, a_4\}$ with probabilities $P(a_1) = 0.1$, $P(a_2) = 0.3$, $P(a_3) = 0.25$, and $P(a_4) = 0.35$, find a Huffman code using the following:
   (a) The first procedure outlined in this chapter
   (b) The minimum variance procedure

   Comment on the difference in the Huffman codes.
6. In many communication applications, it is desirable that the number of 1s and 0s transmitted over the channel be about the same. However, if we look at Huffman codes,

many of them seem to have many more 1s than 0s or vice versa. Does this mean that Huffman coding will lead to inefficient channel usage? For the Huffman code obtained in Problem 3, find the probability that a 0 will be transmitted over the channel. What does this probability say about the question posed above?

7. For the source in Example 3.3.1, generate a ternary code by combining three letters in the first and second steps and two letters in the third step. Compare with the ternary code obtained in the example.

8. In Example 3.4.1, we showed how the tree develops when the sequence $aardv$ is transmitted. Continue this example with the next letters in the sequence, ark.

9. The Monte Carlo approach is often used for studying problems that are difficult to solve analytically. Let's use this approach to study the problem of buffering when using variable-length codes. We will simulate the situation in Example 3.2.1 and study the time to overflow and underflow as a function of the buffer size. In our program, we will need a random number generator, a set of seeds to initialize the random number generator, a counter B to simulate the buffer occupancy, a counter T to keep track of the time, and a value N, which is the size of the buffer. Input to the buffer is simulated by using the random number generator to select a letter from our alphabet. The counter B is then incremented by the length of the codeword for the letter. The output to the buffer is simulated by decrementing B by 2 except when T is divisible by 5. For values of T divisible by 5, decrement B by 3 instead of 2 (why?). Keep incrementing T, each time simulating an input and an output, until either $B \geqslant N$, corresponding to a buffer overflow, or $B < 0$, corresponding to a buffer underflow. When either of these events happens, record what happened and when, and restart the simulation with a new seed. Do this with at least 100 seeds.
Perform this simulation for a number of buffer sizes ($N = 100, 1000, 10{,}000$), and the two Huffman codes obtained for the source in Example 3.2.1. Describe your results in a report.

10. While the variance of lengths is an important consideration when choosing between two Huffman codes that have the same average lengths, it is not the only consideration. Another consideration is the ability to recover from errors in the channel. In this problem we will explore the effect of error on two equivalent Huffman codes.

(a) For the source and Huffman code of Example 3.2.1 (Table 3.5), encode the sequence

$$a_2a_1a_3a_2a_1a_2$$

Suppose there was an error in the channel and the first bit was received as a 0 instead of a 1. Decode the received sequence of bits. How many characters are received in error before the first correctly decoded character?

(b) Repeat using the code in Table 3.9.

(c) Repeat parts (a) and (b) with the error in the third bit.

11. (This problem was suggested by P.F. Swaszek.)

(a) For a binary source with probabilities $P(0) = 0.9$, $P(1) = 0.1$, design a Huffman code for the source obtained by blocking m bits together, $m = 1, 2, \ldots, 8$. Plot the average lengths versus m. Comment on your result.

(b) Repeat for $P(0) = 0.99$, $P(1) = 0.01$.

You can use the program `huff_enc` to generate the Huffman codes.

12. Encode the following sequence of 16 values using the Rice code with $J = 8$ and one split sample option:

$$32, 33, 35, 39, 37, 38, 39, 40, 40, 40, 40, 39, 40, 40, 41, 40$$

For prediction use the previous value in the sequence

$$\hat{y}_i = y_{i-1}$$

and assume a prediction of zero for the first element of the sequence.

13. For an alphabet $\mathcal{A} = \{a_1, a_2, a_3\}$ with probabilities $P(a_1) = 0.7$, $P(a_2) = 0.2$, $P(a_3) = 0.1$, design a 3-bit Tunstall code.

14. Write a program for encoding images using the Rice algorithm. Use eight options, including the fundamental sequence, five split sample options, and the two low-entropy options. Use $J = 16$. For prediction use either the pixel to the left or the pixel above. Encode the Sena image using your program. Compare your results with the results obtained by Huffman coding the differences between pixels.