UNIT-II

The Huffman Coding Algorithm:

Minimum variance Huffman codes: 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 . 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.

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.

3. 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.
4. In an optimum code, the two symbols that occur least frequently will have the same length.

5. 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.

6. To see why the second observation holds true, consider the following situation. Suppose an optimum code *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 *C*. But this violates our initial contention that *C* is an optimal code. Therefore, for an optimal code the second observation also holds true.

7. 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.

Design of a Huffman Code: Let us design a Huffman code for a source that puts out letters from an alphabet $A = \{a_1, a_2, a_3, a_4, a_5\}$ 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

$$c(a_3) = \alpha_2 * 0$$

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,

$$c(a\ ) = \alpha\ * 0$$

TABLE 3.4 The reduced two-letter alphabet.

Letter	Probability	Codeword
a_3	0.6	α_3
α_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:

$$c(a\ ) = 0$$

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

$$c(a_1) = 01$$

$$c(a_3) = 000$$

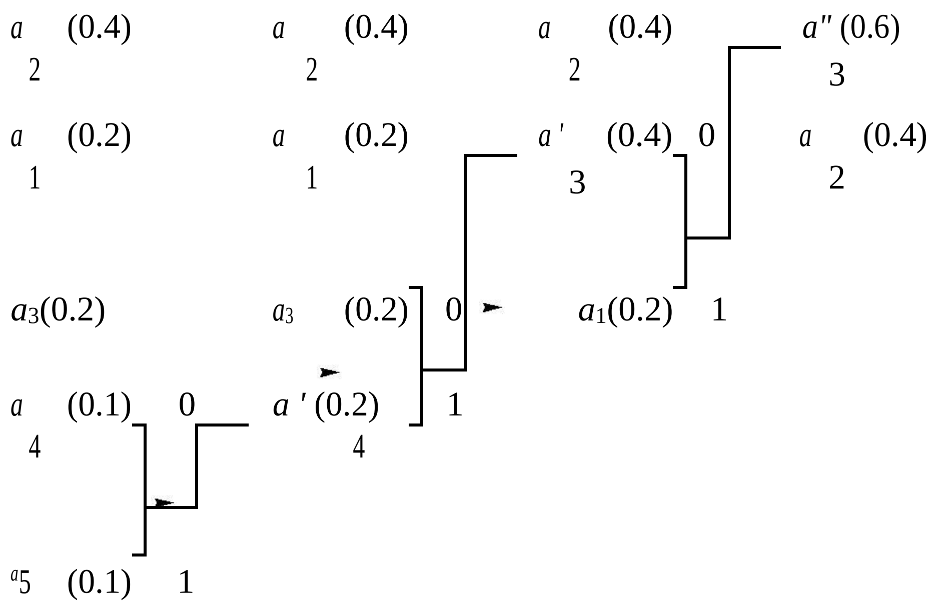
$$c(a_4) = 0010$$

$$c(a_5) = 0011$$

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

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

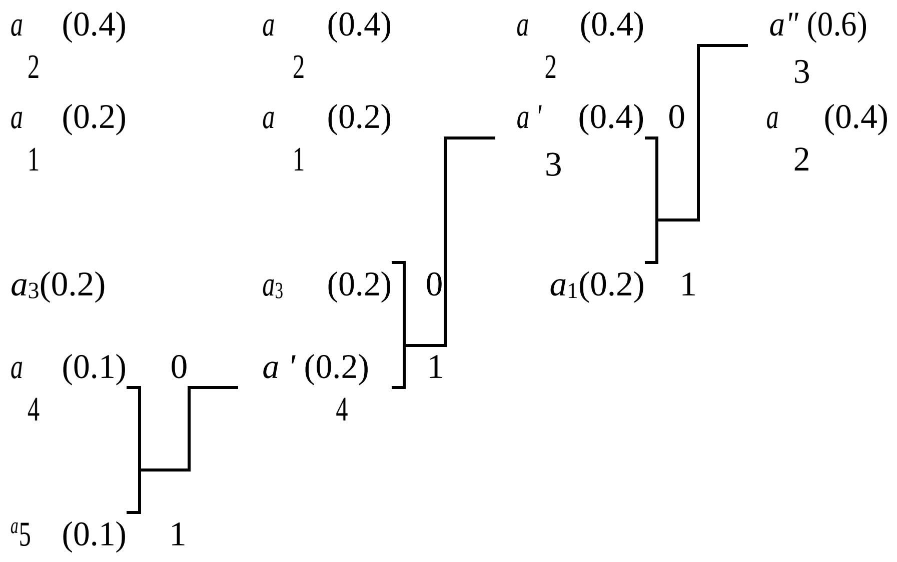


A measure of the efficiency of this code is its redundancy the difference between the entropy and the average length. In this case, the redundancy is 0.078 bits/symbol. The redundancy is zero when the probabilities are negative powers of two.

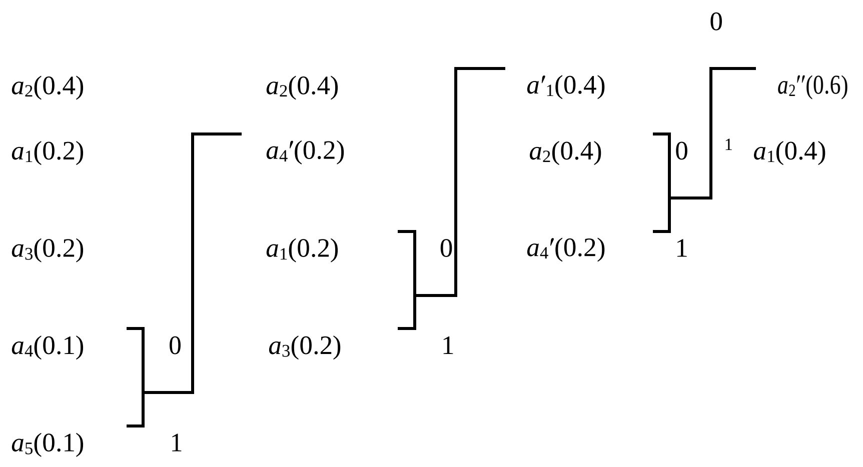
Minimum Variance Huffman Codes: By performing the sorting procedure in a slightly different manner, we could have found a different Huffman code. In the first re-sort, we could place a_4 higher in the list, as shown in Table 3.6.

T A B L E 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	$c(a_3)$



F I G U R E 3 . 2 Building the binary Huffman tree



F I G U R E 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

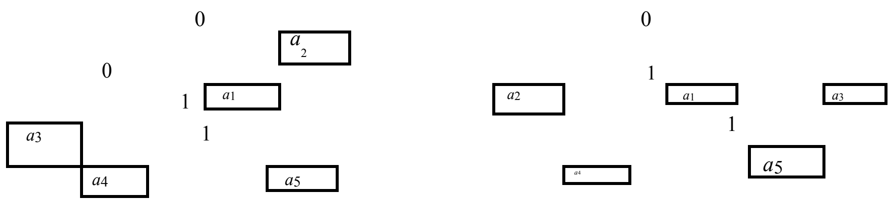
Letter	Probability	Codeword
$a1$	0.4	$\alpha 2$
$a2$	0.4	$c(a2)$
$a4$	0.2	$\alpha 1$

T A B L E 3 . 8 **Reduced two-letter alphabet.**

Letter	Probability	Codeword
$a2$	0.6	$\alpha 3$
$a1$	0.4	$\alpha 2$

T A B L E 3 . 9 **Minimum variance Huffman code.**

Letter	Probability	Codeword
$a1$	0.2	10
$a2$	0.4	00
$a3$	0.2	11
$a4$	0.1	010
a_5	0.1	011



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.

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.

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, . . . , a_m) of size m, then pick e and r such that $m = 2^e + r$ and $0\ r < 2^e$. The letter a_k is encoded as the (e + 1)-bit binary representation of k − 1, if 1 k 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.

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.

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: 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 × 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.

Encoding Procedure: 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.
Example: 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 $0\ r < 2^e$. 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.
Decoding 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.

Example of 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 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.

Golomb code: The Golomb code is described 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

$$q = \frac{n}{m}$$

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

_x 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, . . . , 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 _$\log_2$ m_ bits, where _x_ is the smallest integer greater than or equal to x. We can reduce the number of bits required if we use the _$\log_2$ m -bit binary representation of r for the first $2^{\_\log 2\ m\_}$ − m values, and the _$\log_2$ m bit binary representation of r + $2^{\_\log 2\ m,}$ m for the rest of the values.

Rice 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).

Number of All-Zero Blocks	Codeword
1	1
2	01
3	001
4	0001
5	000001
6	0000001
.	.
.	.
.	.
	630s
63	000 · · · 0 1
	00001
ROS	_ _ _

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
.

T A B L E 3 . 24 A 2-bit Tunstall code.

Sequence	Codeword
A A A	00
A AB	01
AB	10
B	11

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 *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 :

Let's encode the sequence *A A AB A AB A AB A AB A A A* using the code in Table 3.24. Starting at the left, we can see that the string *A A A* occurs in our codebook and has a code of 00. We then code *B* as 11, *A AB* 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 code-word.

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 *A A AB A AB A AB A AB A A A* as in the previous example using the code in Table 3.25. We first encode *A A A* with the code 00. We then encode *B* with 11. The next three symbols are *A AB*. 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

T A B L E 3 . 25 A 2-bit (non-Tunstall) code.

Sequence	Codeword
A A A	00
A B A	01
A B	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

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

Example of Tunstall Codes:

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

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

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 *A A*. Therefore, at the next step we remove *A A* and add all extensions of *A A* as shown in Table 3.28. The final

3-bit Tunstall code is shown in Table 3.28.

T A B L E 3 . 26 Source alphabet & associated probabilities.

Letter	Probability
A	0.60
B	0.30
C	0.10

T A B L E 3 . 27 The codebook after one iteration.

Sequence	**Probability**
B	0.30
C	0.10
A A	0.36
A B	0.18
AC	0.06

T A B L E 3 . 28 A 3-bit Tunstall code.

Sequence	**Code**
B	000
C	001
A B	010
AC	011
A A A	100
A A B	101
A AC	110

Applications of Huffman Coding: Huffman coding is often used in conjunction with other coding techniques.

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 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 com-pression 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 com-pressed 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

F I G U R E 3 . 16 **Test images.**

T A B L E 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.

T A B L E 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

T A B L E 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.

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

T A B L E 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

T A B L E 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.

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.There is also some recent work 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.

T A B L E 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

T A B L E 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.