Source coding

Introduction:

- An important problem in communications is the *efficient* representation of data generated by a discrete source. The process by which this representation is accomplished is called *source encoding.* The device that performs the representation is called a *source encoder.*
- For the source encoder to be *efficient,* we require knowledge of the statistics of the source. In particular, if some source symbols are known to be more probable than others, then we may exploit this feature in the generation of a *source code* by assigning *short* code words to *frequent* source symbols, and *long* code words to *rare* source symbols.
- We refer to such a source code as a *variable-length code.* The *Morse code* is an example of a variable_ length code. In the Morse code, the letters of the alphabet and numerals are encoded into streams of *marks* and *spaces,* denoted as dots "." and dashes "-", respectively.
- Our primary interest is in the development of an efficient source encoder that satisfies two functional requirements:

  1. The code words produced by the encoder are in *binary* form.

  2. The source code is *uniquely decodable,* so that the original source sequence can be reconstructed perfectly from the encoded binary sequence.
- Source coding does not change or alter the source entropy, i.e. the average number of information bits per source symbol. In this sense source entropy is a fundamental property of the source. Source coding

does, however, alter (usually increase) the entropy of the source coded symbols. It may also reduce fluctuations in the information rate from the source and avoid symbol 'surges' which could overload the channel when the message sequence contains many high probability (i.e. frequently occurring, low entropy) symbols.

1. Fixed-Length Code Words

The entropy for a source with statistically independent symbols:

$$H = \sum_{m=1}^{M} P(m) \log_2 \frac{1}{P(m)} \quad \text{(bit/symbol)}$$

...(1)

The maximum possible entropy, H_{max}, of this source would be realized if all symbols were equiprobable, $P(m) = 1/M$, i.e.:

$$H_{\max} = \log_2 M \quad \text{(bit/symbol)}$$

....(2)

A code efficiency can therefore be defined as:

$$\eta_{code} = \frac{H}{H_{\max}} \times 100\%$$

......(3)

If source symbols are coded into binary words then there is a useful alternative interpretation of η_{code}. For a set of symbols represented by binary code words with lengths l_m (binary) digits, an overall code length, L, can be defined as the average codeword length, i.e.:

$$L = \sum_{m=1}^{M} P(m)\, l_m \quad \text{(binary digits/symbol)}$$

...(4)

The code efficiency can then be found from:

$$\eta_{code} = \frac{H}{L} \quad \text{(bit/binary digit)} \times 100\%$$

....(5)

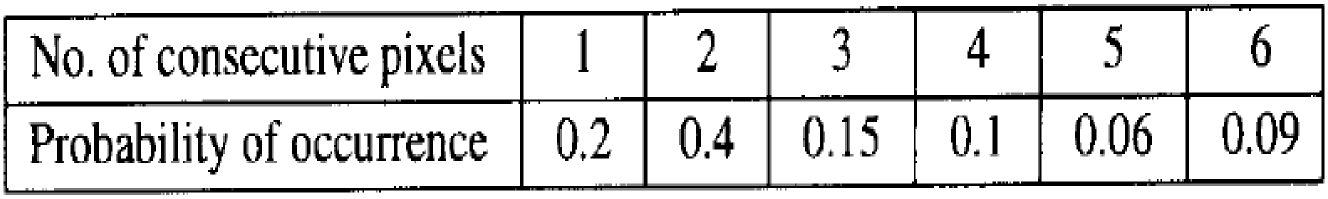
i.e.

$$H_{\max} \ \text{(bit/symbol)} = L \quad \text{(bit/codeword)}$$

....(6)

Example : A scanner converts a black and white document, line-by-line into binary data for transmission. The scanner produces source data comprising symbols representing runs of up to six similar image pixel elements with the probabilities as shown below:

-



No. of consecutive pixels	1	2	3	4	5	6
Probability of occurrence	0.2	0.4	0.15	0.1	0.06	0.09

Determine the average length of a run (in pixels) and the corresponding effective information rate for this source when the scanner is traversing 1000 pixel/s.

$$H=\sum_{m=1}^{6} P(m)\log_2\frac{1}{P(m)}$$

$$=0.2\times 2.32+0.4\times 1.32+0.15\times 2.74+0.1\times 3.32+0.06\times 4.06+0.09\times 3.47$$

$$=2.29 \quad \text{bit/symbol}$$

Average length is

$$L=\sum_{m=1}^{6} P(m)\,l_m=0.2+0.8+0.45+0.4+0.3+0.54=2.69 \quad \text{pixels}$$

At 1000 pixel/s scan rate we generate 1000/2.69 = 372 symbol/s. Thus the source information rate is 2.29 x 372 = 852 bit/s.

We are generally interested in finding a more efficient code which represents the same information using fewer digits on average. This results in different lengths of codeword being used for different symbols. The problem with such variable length codes is in recognizing the start and end of the symbols.

2. Variable-Length Code Words

When the source symbols are not equally probable, a more efficient encoding method is to use variable-length code words. An example of such encoding is the Morse code, which dates back to the nineteenth century. In the Morse code, the letters that occur more frequently are assigned short code words and those that occur infrequently are assigned long code words. Following this general philosophy, we may use the probabilities of occurrence of the different source letters in the selection of the code words. The problem is to devise a method for selecting and assigning the code words to source letters. This type of encoding is called *entropy coding*

The following properties need to be considered when attempting to decode variable length code words:

(1) Unique decoding.

This is essential if the received message is to have only a single possible meaning. Consider an M = 4 symbol alphabet with symbols represented by binary digits as follows:

$$A = 0$$
$$B = 01$$
$$C = 11$$
$$D = 00$$

If we receive the code word 0011 it is not known whether the transmission was *D, C* or A, A, C. This example is not, therefore, uniquely decodable.

2) Instantaneous decoding.

Consider now an M = 4 symbol alphabet, with the following binary representation

$$A = 0$$
$$B = 10$$
$$C = 110$$
$$D = 111$$

This code can be instantaneously decoded using the decision tree shown in Figure (1) since no complete codeword is a prefix of a larger codeword. This is in contrast to the previous example where A is a prefix of both *B* and *D.* The latter example is also a 'comma code' as the symbol zero indicates the end of a code word except for the all ones word whose length is known. Note that we are restricted in the number of available code words with small numbers of bits to ensure we achieve the desired decoding properties.

Using the representation:

$$A = 0$$
$$B = 01$$
$$C = 011$$
$$D = 111$$

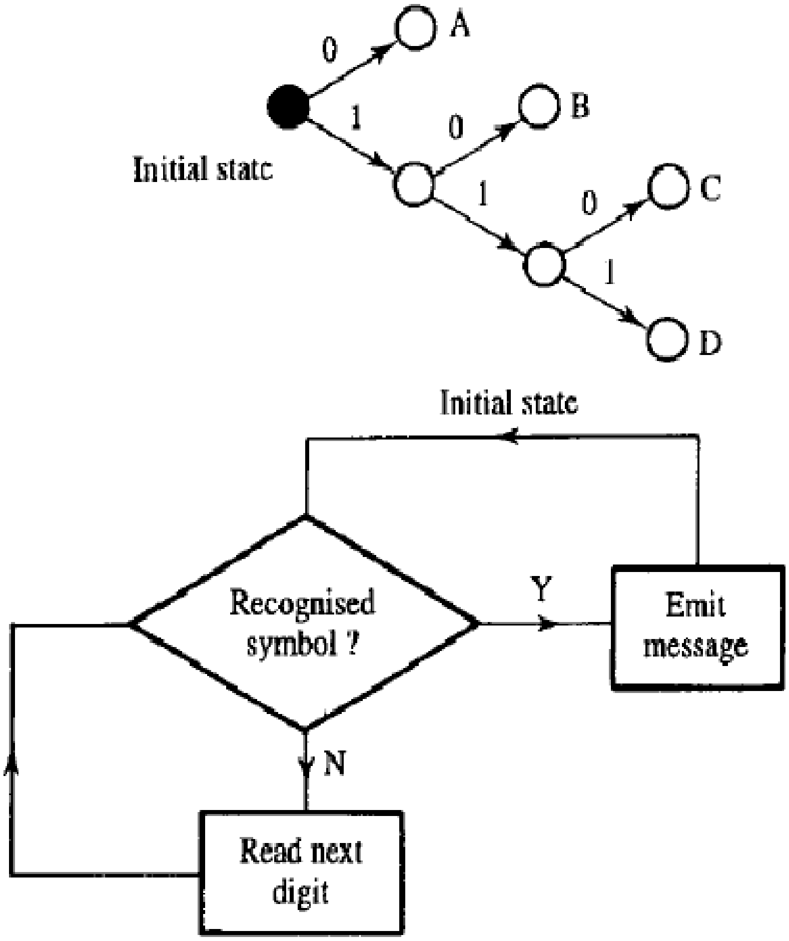
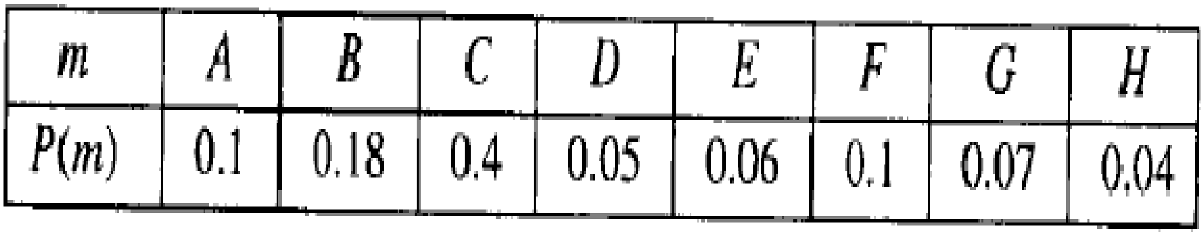


Fig (1)

the code is identical to the example just given but the bits are time reversed. It is thus still uniquely decodable but no longer instantaneous, since early code words are now prefixes of later ones.

Assume an $M = 8$ symbol source $A, \dots \dots, H$ having probabilities of symbol occurrence:



m	A	B	C	D	E	F	G	H
$P(m)$	0.1	0.18	0.4	0.05	0.06	0.1	0.07	0.04

The source entropy is given by:

$$H = \sum_{m} P(m) \log_2 \frac{1}{P(m)} = 2.55 \text{ bit/symbol}$$

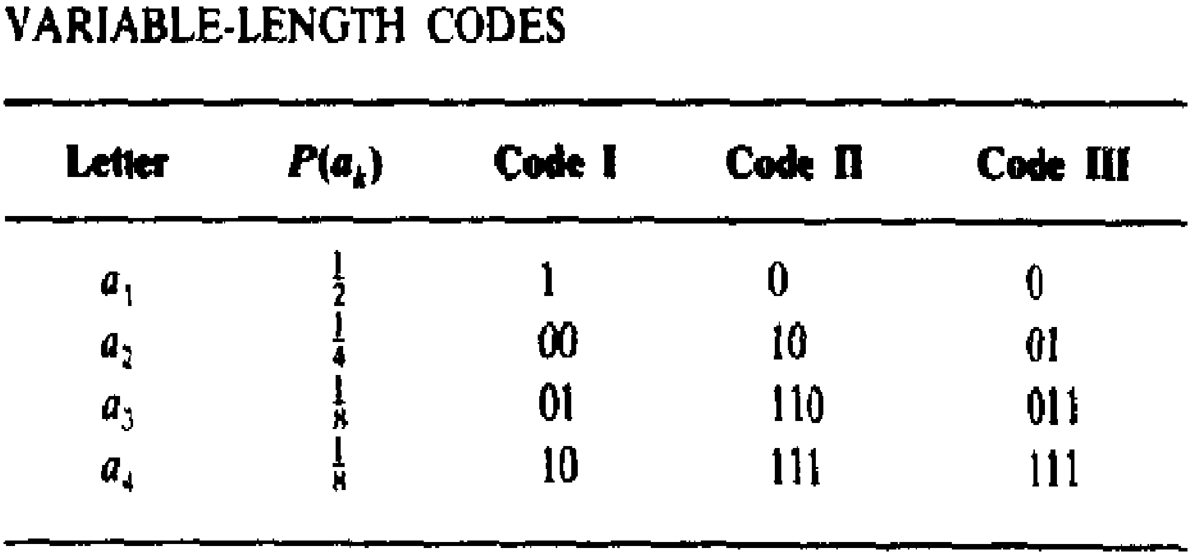
And, at a symbol rate of 1 symbol/s, the information rate is 2.55 bit/s. The maximum entropy of an 8 symbol source is $\log_2 8 = 3$ bit/symbol and the source efficiency is therefore given by:

$$\eta_{source} = \frac{2.55}{3} \times 100\% = 85\%$$

If the symbols are each allocated 3 bits, comprising all the binary patterns between 000 and 111, the coding efficiency will remain unchanged at 85%.

Another example, suppose that a DMS with output letters a_1 , a_2 , a_3 , a_4 and corresponding probabilities $P(a_1) = 1/2$, $P(a_2) = 1/4$, and $P(a_3) = P(a_4) = 1/8$ is encoded as shown in Table 1. Code I is a variable-length code that has a basic flaw. To see the flaw, suppose we are presented with the sequence 001001.Clearly, the first symbol corresponding to 00 is a_2. However, the next four bits are ambiguous (not uniquely decodable). They may be decoded either as $a_4 a_3$ or as $a_l a_2 a_i$.

TABLE 1



VARIABLE-LENGTH CODES

Letter	$P(a_k)$	Code I	Code II	Code III
a_1	$\frac{1}{2}$	1	0	0
a_2	$\frac{1}{4}$	00	10	01
a_3	$\frac{1}{8}$	01	110	011
a_4	$\frac{1}{8}$	10	111	111

Perhaps, the ambiguity can be resolved by wailing for additional bits, but such a decoding delay is highly undesirable. We shall only consider codes that are decodable *instantaneously,* that is, without any decoding delay.

Code II in Table 1 is *uniquely decodable* and *instantaneously decodable.* It is convenient to represent the code words in this code graphically as terminal nodes of a tree, as shown in Fig. 2. We observe that the digit 0 indicates the end of a code word for the first three code words. This characteristic plus the fact that no code word is longer than three binary digits makes this code instantaneously decodable. Note that no code word in this code is a prefix of any other code word. In general, the *prefix condition* requires that for a given code word C_k of length k having elements (b_1, $b_2, \dots$, b_k), there is no other code word of length $l < k$ with elements $(b_1,\ b_2, \dots,\ b_l)$ for $1 \le l \le k-1$.

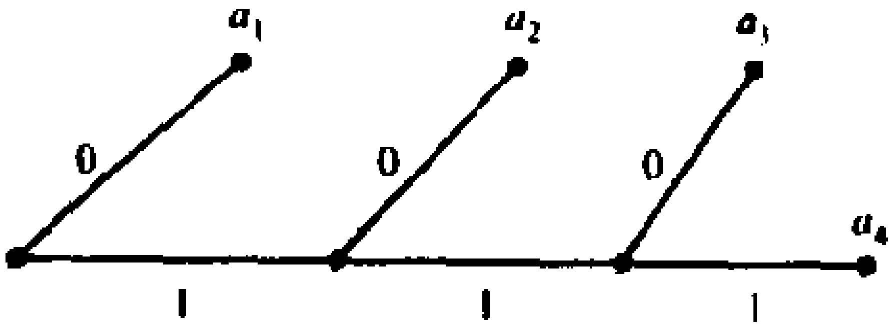


Fig 2

In other words, there is no code word of length l< *k* that is identical to the first / binary digits of another code word of length *k*>*l*. This property makes the code words instantaneously decodable. Code III given in Table 1 has the tree structure shown in Fig. 3. We note that in this case the code is uniquely decodable but *not* instantaneously decodable. Clearly, this code does *not* satisfy the prefix condition.

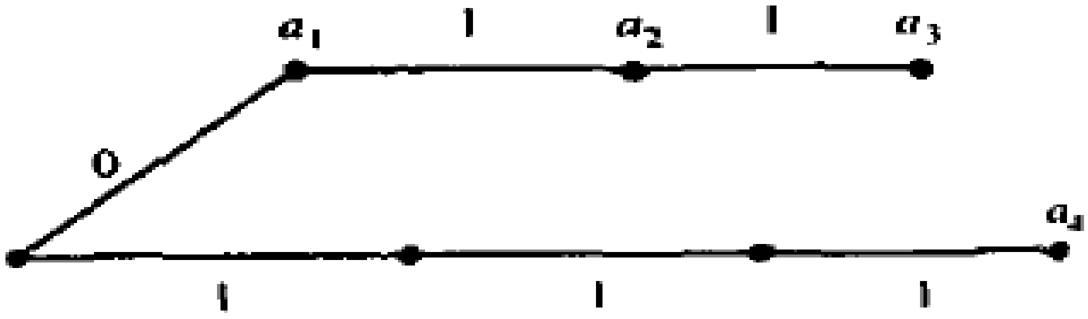
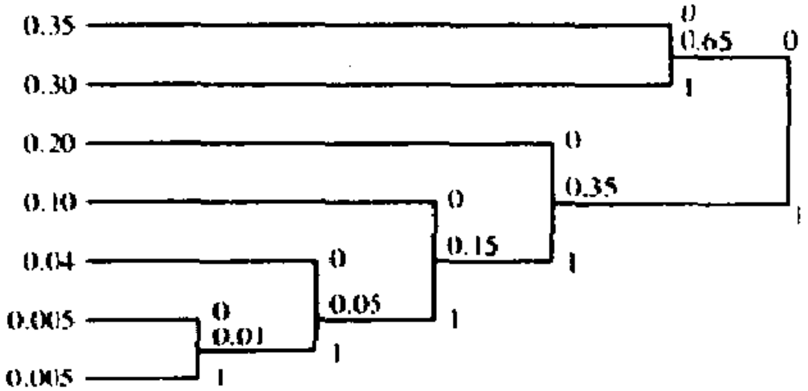


Fig 3

Our main objective is to devise a systematic procedure for constructing uniquely decodable variable-length codes that are efficient in the sense that the average number of bits per source letter.

Huffman Coding Algorithm Huffman (1952) devised a variable-length encoding algorithm, based on the source letter probabilities $P(x_i)$, i = 1, *2*, ..., *L*. This algorithm is optimum in the sense that the average number of binary digits required to represent the source symbols is a minimum, subject to the constraint that the code words satisfy the prefix condition, as defined above, which allows the received sequence to be uniquely and instantaneously decodable. We illustrate this encoding algorithm by means of two examples.

Example 1 Consider a DMS with seven possible symbols x_1 $x_2, \ldots . \; x_7$ having the probabilities of occurrence illustrated in Fig. 4.

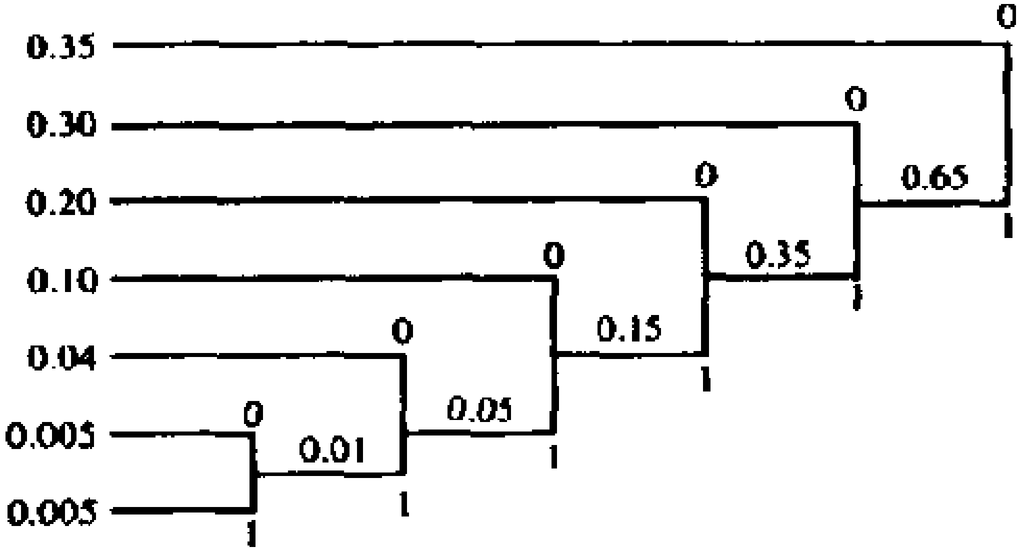


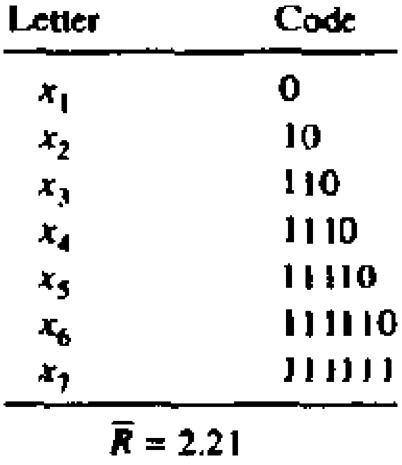
Letter	Probability	Self-information	Code
x_1	0.35	1.5146	00
x_2	0.30	1.7370	01
x_3	0.20	2.3219	10
x_4	0.10	3.3219	110
x_5	0.04	4.6439	1110
x_6	0.005	7.6439	11110
x_7	0.005	7.6439	11111
	$H(X) = 2.11$	$\bar{R} = 2.21$	

Fig 4

We have ordered the source symbols in decreasing order of the probabilities, i.e. $P(x_1) > P(x_2) > \ldots > P(x_7)$. We begin the encoding process with the two least probable symbols c_6 and x_7. These two symbols are tied together as shown in Fig. 4, with the upper branch assigned a 0 and the lower branch assigned a 1. The probabilities of these two branches are added together at the node where the two branches meet to yield the probability 0.01. Now we have the source symbols $x_1 . . . , x_5$ plus a new symbol, say $x_6{}'$, obtained by combining x_6 and x_7. The next step is to join the two least probable symbols from the set x_1, x_2, x_3, x_4, x_s, x'_6. These are x_5 and x'_6, which have a combined probability of 0.05. The branch from x_5 is assigned a 0 and the branch from x'_6 is assigned a 1. This procedure continues until we exhaust the set of possible source letters. The result is a code tree with branches that contain the desired code words. The code words are obtained by beginning at the rightmost node in the tree and proceeding to the left. The resulting code words are listed in Fig.4. The average number of binary digits per symbol for this code is $R = 2.21$ bits/symbol .The entropy of the source is 2.11 bits/symbol.

We make the observation that the code is not necessarily unique. For example, at the next to the last step in the encoding procedure, we have a tie between x_1 and x'_3 , since these symbols are equally probable. At this point, we chose to pair x_1, with x_2. An alternative is to pair x_2 with x_3. If we choose this pairing, the resulting code is illustrated in Fig. 5. The average number of bits per source symbol for this code is also 2.21. Hence, the resulting codes are equally efficient. Secondly, the assignment of a 0 to the upper branch and a 1 to the lower (less probable) branch is arbitrary. We may simply reverse the assignment of a 0 and 1 and still obtain an efficient code satisfying the prefix condition.



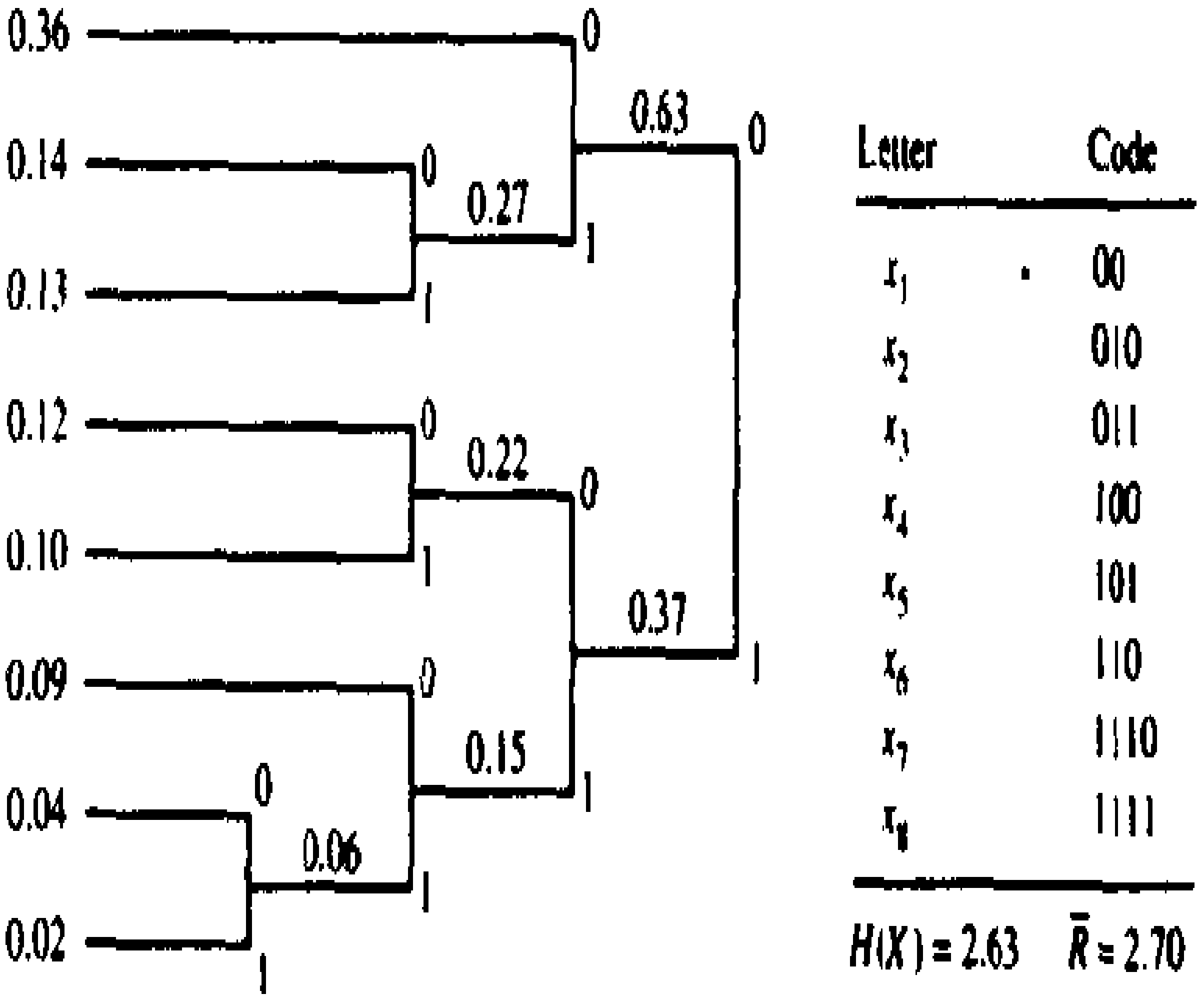


Letter	Code
x_1	0
x_2	10
x_3	110
x_4	1110
x_5	11110
x_6	111110
x_7	111111

$\bar{R} = 2.21$

Fig 5

Example 2 As a second example, let us determine the Huffman code for the output of a DMS illustrated in Fig. 6. The entropy of this source is $H(X) = 2.63$ bits/symbol. The Huffman code as illustrated in Fig. 6 has an average length of $R = 2.70$ bits/symbol. Hence, its efficiency is 0.97.



Letter	Code
x_1	00
x_2	010
x_3	011
x_4	100
x_5	101
x_6	110
x_7	1110
x_8	1111

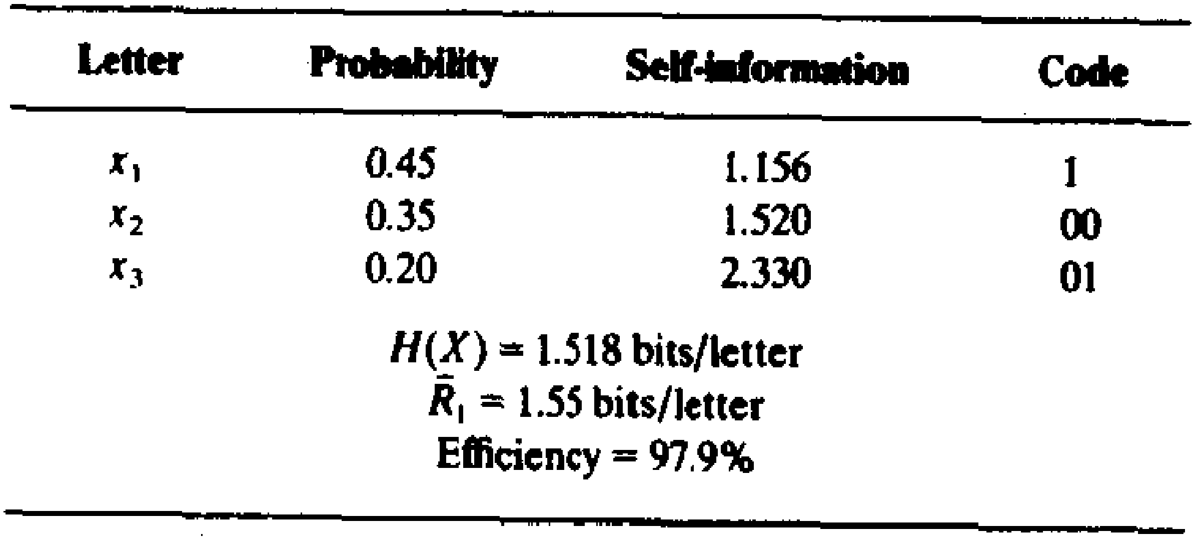
$H(X) = 2.63 \quad \bar{R} = 2.70$

Fig 6

However, instead of encoding on a symbol-by-symbol basis, a more efficient procedure is to encode blocks of L symbols at a time.

Example 3: The output of a DMS consists of *letters* x_1, x_2, and x_3 with probabilities 0.45, 0.35, and 0.20, respectively. The entropy of this source is $H(X) = 1.518$ bits/symbol. The Huffman code for this source, given in Table 2, requires $R = 1.55$bits/symbol and results in an efficiency of 97.9%. If pairs of symbols are encoded by means of the Huffman algorithm, the resulting code is as given in Table 3. The entropy of the source output for pairs of letters is $2H(X)$ = 3.036 bits/symbol pair. On the other hand, the Huffman code requires R_2 = 3.0675 bits/symbol pair. Thus, the efficiency of the encoding increases to $2H(X)/R_2$ = 0.990 or, equivalently, to 99.0%.

TABLE 2



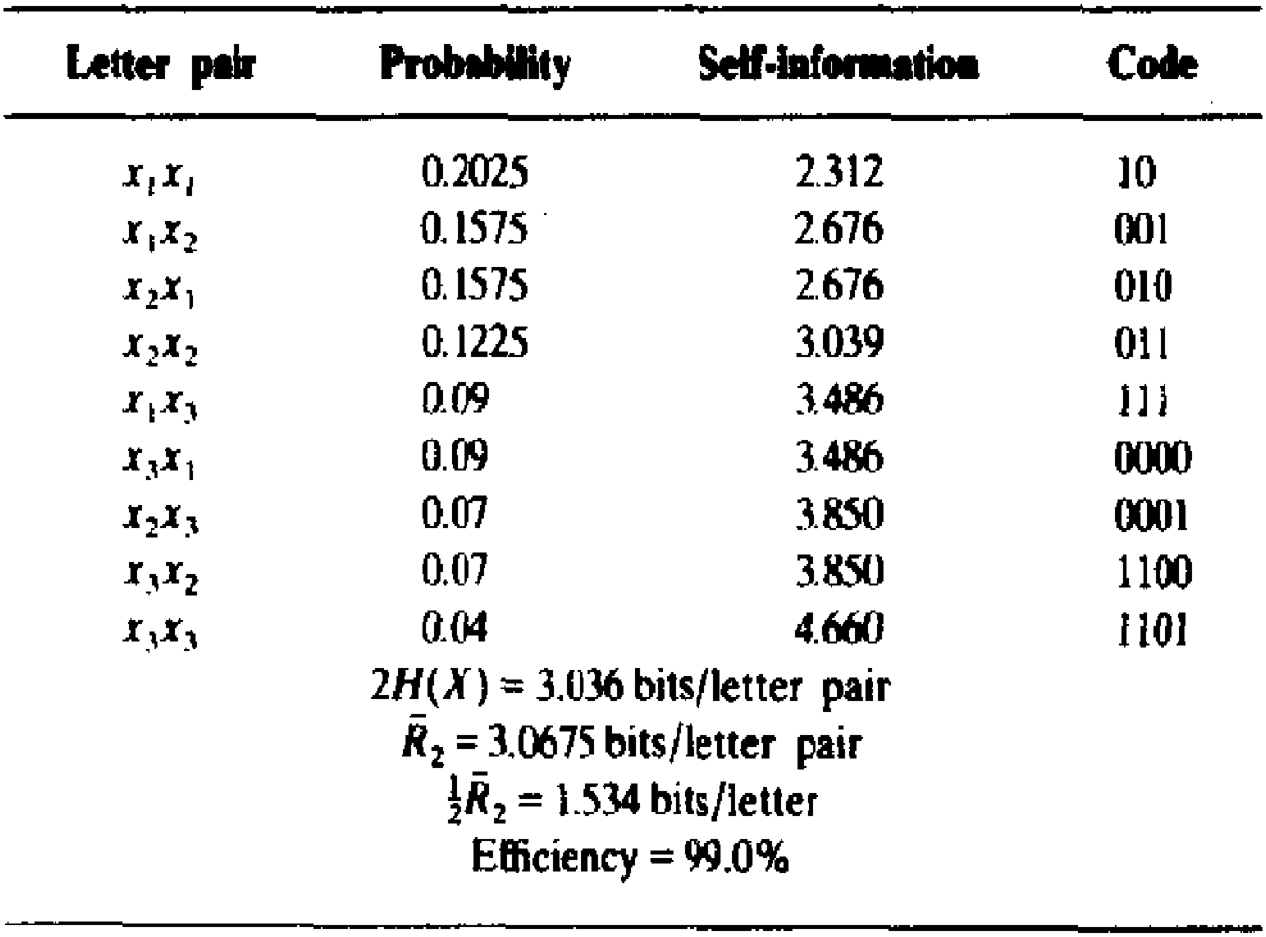
Letter	Probability	Self-information	Code
x_1	0.45	1.156	1
x_2	0.35	1.520	00
x_3	0.20	2.330	01

$H(X)$ = 1.518 bits/letter

$\bar{R}_1$ = 1.55 bits/letter

Efficiency = 97.9%

TABLE 3



Letter pair	Probability	Self-information	Code
x_1x_1	0.2025	2.312	10
x_1x_2	0.1575	2.676	001
x_2x_1	0.1575	2.676	010
x_2x_2	0.1225	3.039	011
x_1x_3	0.09	3.486	111
x_3x_1	0.09	3.486	0000
x_2x_3	0.07	3.850	0001
x_3x_2	0.07	3.850	1100
x_3x_3	0.04	4.660	1101

$2H(X) = 3.036$ bits/letter pair

$\bar{R}_2 = 3.0675$ bits/letter pair

$\frac{1}{2}\bar{R}_2 = 1.534$ bits/letter

Efficiency $= 99.0\%$

LEMPEL-ZIV CODING

A drawback of the Huffman code is that it requires knowledge of a probabilistic model of the source; unfortunately, in practice, source statistics are not always known *a Priori.* Moreover, in modeling text we find that storage requirements prevent the Huffman code from capturing the higher-order relationships between words and phrases, thereby compromising the efficiency of the code. To overcome these practical limitations, we may use the *Lempel-Ziv algorithm* which is intrinsically *adaptive* and simpler to implement than Huffman coding.

Basically, encoding in the Lempel-Ziv algorithm is accomplished by *parsing the source data stream into segments that are the shortest subsequences not encountered previously.* To illustrate this simple yet elegant idea, consider the example of an input binary
sequence specified as follows:

000101110010100101 ...

It is assumed that the binary symbols 0 and 1 are already stored in that order in the code book. We thus write
Subsequences stored: 0,1
Data to be parsed: 000101110010100101 ...

The encoding process begins at the left. With symbols 0 and 1 already stored, the *shortest subsequence* of the data stream encountered for the first time and not seen before is so we write

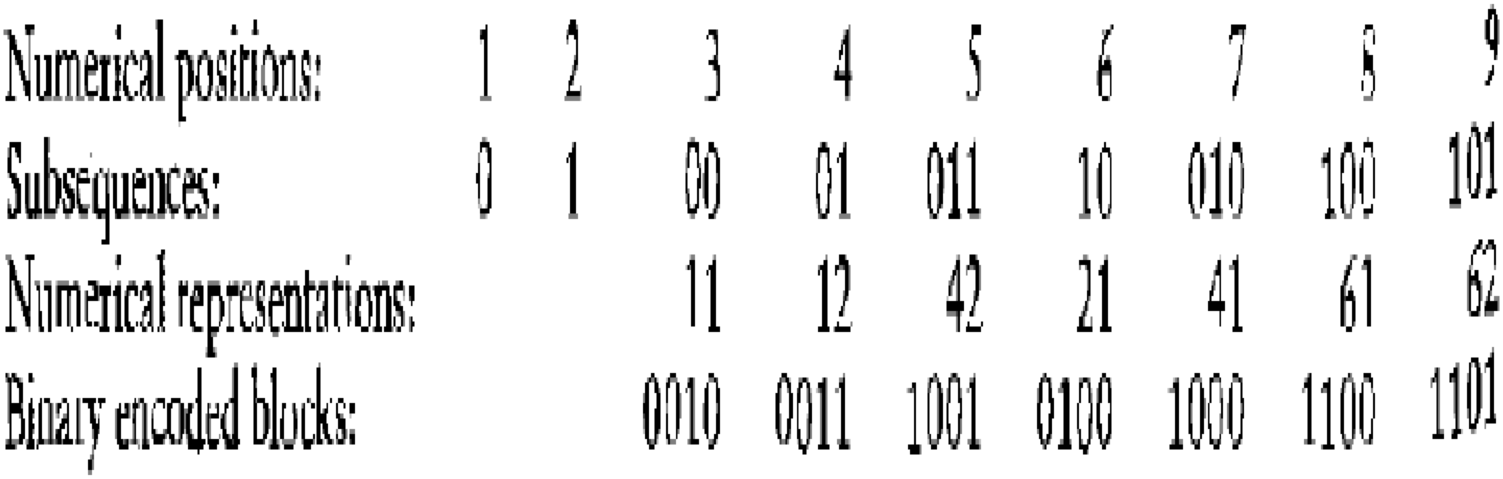
Subsequences stored: 0,1,00
Data to be parsed: 0101110010100101 ...

The second shortest subsequence not seen before is 01; accordingly, we go on to write

Subsequences stored: 0,1,00,01
Data to be parsed: 01110010100101 ...

The next shortest subsequence not encountered previously is 011; hence, we write
Subsequences stored: 0, 1, 00, 01, 011
Data to be parsed: 10010100101 ...

We continue in the manner described here until the given data stream has been completely parsed. Thus, for the example at hand, we get the *code book* of binary subsequences shown in the second row of Figure 7.



Numerical positions:	1	2	3	4	5	6	7	8	9
Subsequences:	0	1	00	01	011	10	010	100	101
Numerical representations:			11	12	42	21	41	61	62
Binary encoded blocks:			0010	0011	1001	0100	1000	1100	1101

Fig 7

The first row shown in this figure merely indicates the numerical positions of the individual subsequences in the code book. We now recognize that the first subsequence of the data stream, 00, is made up of the concatenation of the *first* code book entry, 0, with itself; it is therefore represented by the number 11. The second subsequence of the data stream, 01, consists of the *first* code book entry, 0, concatenated with the *second* code book entry, 1; it is therefore represented by the number 12. The remaining subsequences are treated in a similar fashion. The complete set of numerical representations for the various subsequences in the code book is shown in the third row of Figure 6. As a further example illustrating the composition of this row, we note that the subsequence 010 consists of the concatenation of the subsequence 01 in position 4 and symbol 0 in position 1; hence, the numerical representation 41. The last row shown in Figure 7 is the binary encoded representation of the different subsequences of the data stream.

From the example described here, we note that, in contrast to Huffman coding, the Lempel-Ziv algorithm uses fixed-length codes to represent a variable number of source symbols; this feature makes the Lempel-Ziv code suitable for synchronous transmission. In practice, fixed blocks of 12 bits long are used, which implies a code book of 4096 entries.

The Lempel-Ziv algorithm is widely used in the compression of computer files. The "compress" and "uncompress" utilities under the UNIX® operating system and numerous algorithms under the MS-DOS operating system are implementations of various versions of this algorithm.

H.W: Encode the following bit streams using Lempel-Ziv code ?

10101101001001110101000011001110101100011011

Fano-Shannon Method: This method was suggested by Shannon and Weaver in 1949 and modified by Fano in 1961. This method is used for any source and it involves writing the symbol probabilities in a table in a descending order. Then dividing them into pairs as shown in the example bellow:-

EXAMPLE: Find a code using Fano-Shannon coding for a source of five symbols of probabilities 0.5 , 0.2 , 0.1, 0.1, 0.1? Then find its efficiency?

Sol:- Dividing lines are inserted to successively divide the probabilities into halves, quarters ,...etc as shown in the Figure 8. A ‵0′ and ‵1′ are added to the code at each division and the final code obtained by reading from the right towards each symbol, writing down the appropriate sequence of 0′s and 1 ‵s.

					CODE
S_1	0.5	*0*			*0*
S_2	0.2	*1*	*0*	*0*	*100*
S_3	0.1	*1*	*0*	*1*	*101*
S_4	0.1	*1*	*1*	*0*	*110*
S_5	0.1	*1*	*1*	*1*	*111*

Fig 8

$L=0.5X1 +0.2X3 +0.1X3 +0.1X3 = 20$

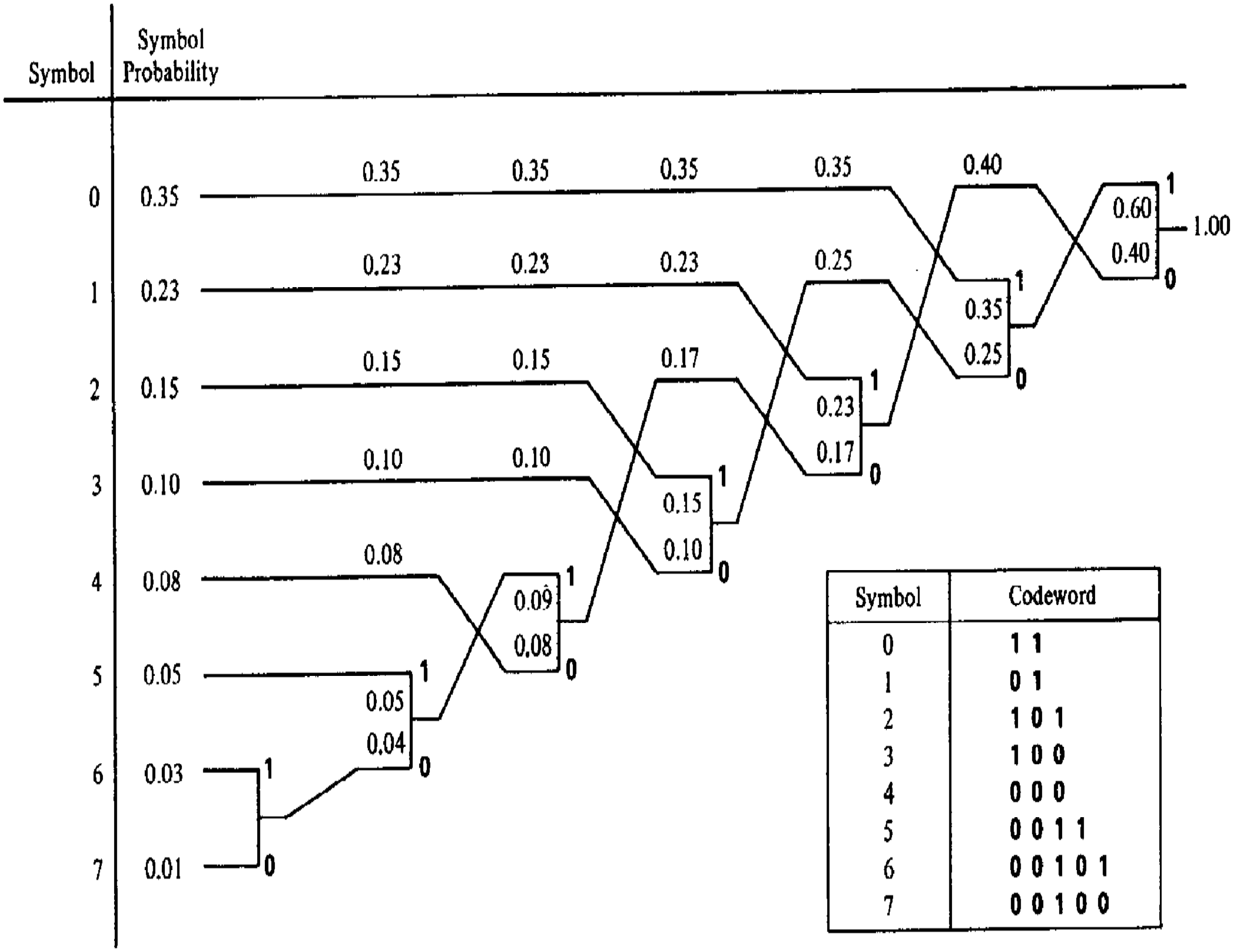
$H=1.96$

$\eta=0.98$

H.W : Repeat previous example for the following symbols and probabilities ?

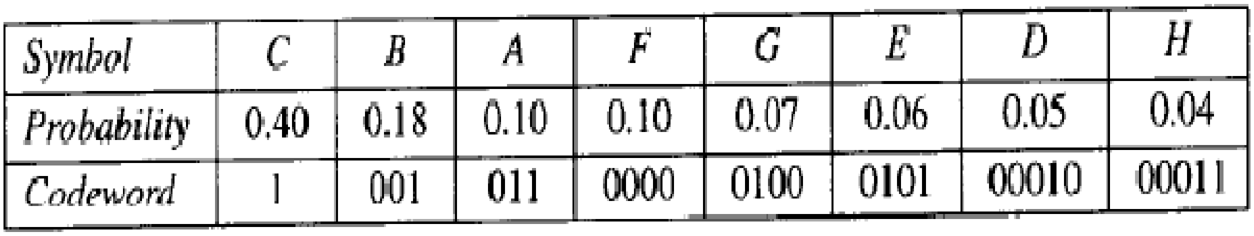
S_1 0.25
S_2 0.25
S_3 0.25
S_4 0.25

More Examples for Huffman code:-

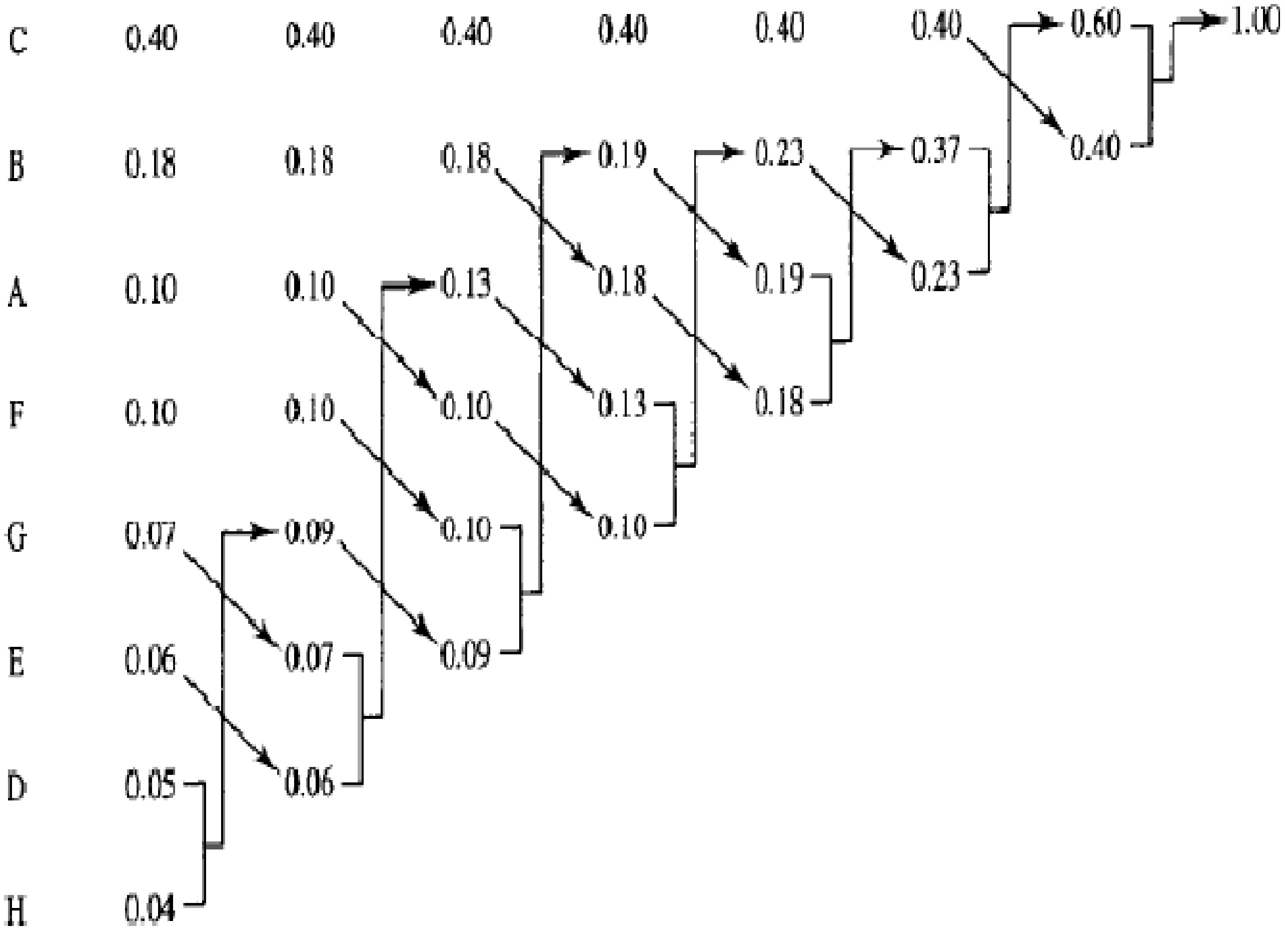


Symbol	Codeword
0	1 1
1	0 1
2	1 0 1
3	1 0 0
4	0 0 0
5	0 0 1 1
6	0 0 1 0 1
7	0 0 1 0 0

Figure 3.4-2. Huffman binary source coding of eight-symbol source.



Symbol	*C*	*B*	*A*	*F*	*G*	*E*	*D*	*H*
Probability	0.40	0.18	0.10	0.10	0.07	0.06	0.05	0.04
Codeword	1	001	011	0000	0100	0101	00010	00011


C 0.40 0.40 0.40 0.40 0.40 0.40 0.60 1.00
B 0.18 0.18 0.18 0.19 0.23 0.37 0.40
A 0.10 0.10 0.13 0.18 0.19 0.23
F 0.10 0.10 0.10 0.13 0.18
G 0.07 0.09 0.10 0.10
E 0.06 0.07 0.09
D 0.05 0.06
H 0.04