WEEK 4

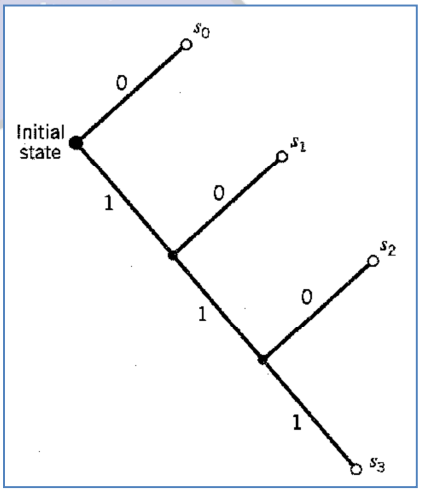
Prefix Coding

Consider a discrete memoryless source of alphabet $\{s_0, s_1, \ldots, s_{k-1}\}$ and statistics $\{P_0, P_1, \ldots, P_{k-1}\}$. For a source code representing the output of this source to be of practical use, the code has to be uniquely decodable. This restriction ensures that for each finite sequence of symbols emitted by the source, the corresponding sequence of code words is different from the sequence of code words corresponding to any other source sequence. We are specifically interested in a special class of codes satisfying a restriction known as the *prefix condition*. To define the prefix condition, let the code word assigned to source symbol s_k be denoted by $(m_{k1}\ m_{k2}, \ldots, m_{kn})$, where the individual elements $m_{k1}\ m_{k2}, \ldots, m_{kn}$ are 0s and 1s, and n is the code-word length. Any sequence made up of the initial part of the code word is called a *prefix* of the code word. A *prefix code* is defined as a code in which no code word is the prefix of any other code word.

To illustrate the meaning of a prefix code, consider the three source codes described in the table. Code I is **NOT** a prefix code since the bit 0, the code word for s_0, is a prefix of 00, the code word for s_2. Likewise, the bit 1, the code word for s_1 is a prefix of 11, the code word for s_3. Similarly, we may show that code III is not a prefix code, but code II is.

Source Symbol	*Probability of Occurrence*	*Code I*	*Code II*	*Code III*
s_0	0.5	0	0	0
s_1	0.25	1	10	01
s_2	0.125	00	110	011
s_3	0.125	11	111	0111

To decode a sequence of code words generated from a prefix source code, the *source decoder* simply starts at the beginning of the sequence and decodes one code word at a time. Specifically, it sets up what is equivalent to a *decision tree*, which is a graphical portrayal of the code words in the particular source code. For example, Figure below depicts the decision tree corresponding to code II in the above table. The tree has an initial state and four terminal states corresponding to source symbols s_0, s_1, s_2, and s_3. The decoder always starts at the initial state. The first received bit moves the decoder to the terminal state s_0 if it is 0, or else to a second decision point if it



is 1. In the latter case, the second bit moves the decoder one step further down the tree, either to terminal state s_1 if it is 0, or else to a third decision point if it is 1, and so on. Once each terminal state emits its symbol, the decoder is reset to its initial state.

Moreover, if a prefix code has been constructed for a discrete memory less source with source alphabet (s_0, s_1…, s_{K-1}) and source statistics (p_0, p_1,…, P_{k-1}) and the code word for symbol s_k has length l_k , k = 0, 1, . . ., K - 1, then the code-word lengths of the code always satisfy a certain inequality known as the *Kraft-McMillan Inequality,* as shown by

$$\sum_{k=0}^{k-1} 2^{-l_k} \leq 1$$

With prefix code, symbole s_k emitted with probability $p_k = 2^{-l_k}$

Then we have :

$$\sum_{k=0}^{k-1} 2^{-l_k} = \sum_{k=0}^{k-1} p_k = 1$$

And the average code word length is

$$L = \sum_{k=0}^{k-1} \frac{l_k}{2^{l_k}}$$

And the corresponding entropy of the source is:

$$H(\delta) = \sum_{k=0}^{k-1} \left(\frac{1}{2^{l_k}}\right) \log(2^{l_k})$$

$$= \sum_{k=0}^{k-1} \frac{l_k}{2^{l_k}}$$

- If Kraft's inequality holds with strict inequality, the code has some redundancy.
- If Kraft's inequality holds with strict equality, the code in question is a complete code.
- If Kraft's inequality does not hold, the code is not uniquely decodable.

Shannon–Fano Coding

It is a technique for constructing a prefix code based on a set of symbols and their probabilities (estimated or measured). It is suboptimal in the sense that it does not achieve the lowest possible expected code word length like Huffman coding; however unlike Huffman coding, it does guarantee that all code word lengths are within one bit of their theoretical ideal – log P(x).

A Shannon–Fano tree is built according to a specification designed to define an effective code table. The actual algorithm is simple:

In Shannon–Fano coding, the symbols are arranged in order from most probable to least probable, and then divided into two sets whose total probabilities are as close as possible to being equal. All symbols then have the first digits of their codes assigned; symbols in the first set receive "0" and symbols in the second set receive "1". As long as any sets with more than one member remain, the same process is repeated on those sets, to determine successive digits of their codes. When a set has been reduced to one symbol, of course, this means the symbol's code is complete and will not form the prefix of any other symbol's code.

Example,

Construct the Shannon-Fano code for the following letters with their probability:

A=1/2 , B=1/4, C=1/8, D=1/16, E=1/32, F=1/32

Sol:

Symbols	Probability	bit assignment					Generated Code
A	1/2	0					0
B	1/4	1	0				10
C	1/8	1	1	0			110
D	1/16	1	1	1	0		1110
E	1/32	1	1	1	1	0	11110
F	1/32	1	1	1	1	1	11111

The entropy of the generated code is :

$$\mathrm{H(m)} = \sum_{i=1}^{6} P_i \, log \frac{1}{P_i} = 1.9375 \; bits$$

The generated code length is :

$$L = \sum_{i=1}^{6} P_i L_i = \frac{1}{2}(1) + \frac{1}{4}(2) + \frac{1}{8}(3) + \frac{1}{16}(4) + 2 * \left(\frac{1}{32}\right)(5) = 1.9375 \; bits$$

$$efficiency \; (\eta) = \frac{H(m)}{L} = \frac{1.9375}{1.9375} = 1 \; (perfect \; code) \; with \; zero \; redundancy$$

Unfortunately, Shannon–Fano does not always produce optimal prefix codes !

Shannon-Fano Algorithm

A Shannon–Fano tree is built according to a specification designed to define an effective code table. The actual algorithm is simple:

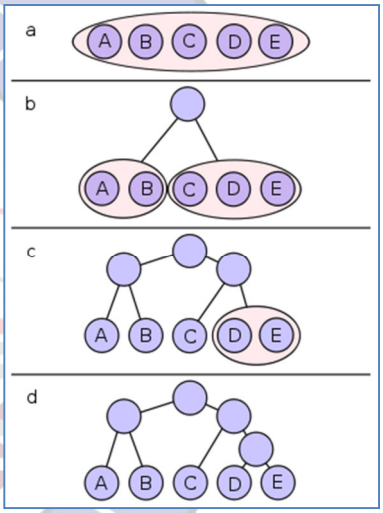
1. For a given list of symbols, develop a corresponding list of probabilities or frequency counts so that each symbol's relative frequency of occurrence is known.
2. Sort the lists of symbols according to frequency, with the most frequently occurring symbols at the left and the least common at the right.
3. Divide the list into two parts, with the total frequency counts of the left half being as close to the total of the right as possible.
4. The left half of the list is assigned the binary digit 0, and the right half is assigned the digit 1. This means that the codes for the symbols in the first half will all start with 0, and the codes in the second half will all start with 1.
5. Recursively apply the steps 3 and 4 to each of the two halves, subdividing groups and adding bits to the codes until each symbol has become a corresponding code leaf on the tree.

The example shows the construction of the Shannon code for a small alphabet. The five symbols which can be coded have the following frequency:

Symbol	**A**	**B**	**C**	**D**	**E**
Count	15	7	6	6	5
Probabilities	0.38461538	0.17948718	0.15384615	0.15384615	0.12820513

All symbols are sorted by frequency, from left to right (shown in Figure a). Putting the dividing line between symbols B and C results in a total of 22 in the left group and a total of 17 in the right group. This minimizes the difference in totals between the two groups.

With this division, A and B will each have a code that starts with a 0 bit, and the C, D, and E codes will all start with a 1, as shown in Figure b. Subsequently, the left half of the tree gets a new division between A and B, which puts A on a leaf with code 00 and B on a leaf with code 01.



After four division procedures, a tree of codes results. In the final tree, the three symbols with the highest frequencies have all been assigned 2-bit codes, and two symbols with lower counts have 3-bit codes as shown table below:

Symbol	**A**	**B**	**C**	**D**	**E**
Code	00	01	10	110	111

Results in 2 bits for A, B and C and per 3 bits for D and E an average bit number of:

$$\frac{[2bits.(15+7+6)+3bits(6+5)]}{39Symbols} = 2.28\ bitsper\ symbol$$

H.W : A source generates six messages with probabilities 0.30, 0.25, 0.15, 0.12, 0.10, and 0.08, respectively, construct Shannon-Fano code for this source & find its efficiency & redundancy:

H.W : Construct Shannon-Fano code for the set of probabilities {0.35, 0.17, 0.17, 0.16, 0.15}