Shannon – Fano Code

Shannon–Fano coding, named after Claude Elwood Shannon and Robert Fano, is a technique for constructing a prefix code based on a set of symbols and their probabilities. It is suboptimal in the sense that it does not achieve the lowest possible expected codeword length like Huffman coding; however unlike Huffman coding, it does guarantee that all codeword lengths are within one bit of their theoretical ideal $I(x) = -\log P(x)$.

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.

The algorithm works, and it produces fairly efficient variable-length encodings; when the two smaller sets produced by a partitioning are in fact of equal probability, the one bit of information used to distinguish them is used most efficiently. Unfortunately, Shannon–Fano does not always produce optimal prefix codes.

For this reason, Shannon–Fano is almost never used; Huffman coding is almost as computationally simple and produces prefix codes that always achieve the lowest expected code word length. Shannon–Fano coding is used in the IMPLODE compression method, which is part of the ZIP file format, where it is desired to apply a simple algorithm with high performance and minimum requirements for programming.

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 part being as close to the total of the right as possible.
4. The left part of the list is assigned the binary digit 0, and the right part is assigned the digit 1. This means that the codes for the symbols in the first part will all start with 0, and the codes in the second part 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.

Example 1:

The source of information A generates the symbols {A0, A1, A2, A3 and A4} with the corresponding probabilities {0.4, 0.3, 0.15, 0.1 and 0.05}. Encoding the source symbols using binary encoder and Shannon-Fano encoder gives:

Source Symbol	P_i	Binary Code	Shannon-Fano
A0	0.4	000	0
A1	0.3	001	10
A2	0.15	010	110
A3	0.1	011	1110
A4	0.05	100	1111
L_{avg}	H = 2.0087	3	2.05

The Entropy of the source is

$$H = -\sum_{i=0}^{4} Pi\ \log_2 Pi = \ 2.0087\ \text{bit/symbol}$$

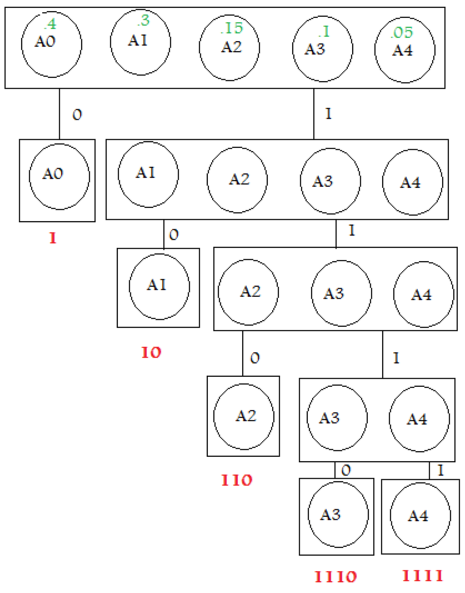
Since we have 5 symbols ($5<8=2^3$), we need 3 bits at least to represent each symbol in binary (fixed-length code). Hence the average length of the binary code is

$$\text{Lavg} = \sum_{i=0}^{4} Pi\ li = \ 3\ (0.4 + 0.3 + 0.15 + 0.1 + 0.05) = 3\ \text{bit/symbol}$$

Thus the efficiency of the binary code is

$$\eta = \frac{H}{\text{Lavg}} = \frac{2.0087}{3} = 67\%$$

Shannon-Fano code is a top-down approach. Constructing the code tree, we get



The average length of the Shannon-Fano code is

$$\mathrm{Lavg} = \sum_{i=0}^{4} \mathrm{Pi\ li} = 0.4 * 1 + 0.3 * 2 + 0.15 * 3 + 0.1 * 4 + 0.05 * 4 = 2.05\ \mathrm{bit/symbol}$$

Thus the efficiency of the Shannon-Fano code is

$$\eta = \frac{\mathrm{H}}{\mathrm{Lavg}} = \frac{2.0087}{2.05} = 98\%$$

This example demonstrates that the efficiency of the Shannon-Fano encoder is much higher than that of the binary encoder.

Example 2:

The geometric source of information A generates the symbols {A0, A1, A2 and A3} with the corresponding probabilities {0.4, 0.2, 0.2 and 0.2}. Encoding the source symbols using binary encoder and Shannon-Fano encoder gives:

Source Symbol	**P_i**	**Binary Code**	**Sh-F Code I**	**Sh-F Code II**
A0	0.4	00	0	00
A1	0.2	01	10	01
A2	0.2	10	110	10
A3	0.2	11	111	11
L_{avg}	**H = 1.922**	**2**	**2**	**2**

The Entropy of the source is

$$\mathrm{H} = -\sum_{i=0}^{3} \mathrm{Pi}\ \log_2 \mathrm{Pi} = 1.922\ \mathrm{bit/symbol}$$

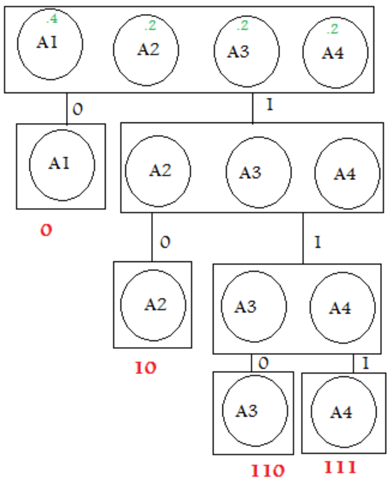
Since we have 4 symbols ($4=2^2$), we need 2 bits at least to represent each symbol in binary (fixed-length code). Hence the average length of the binary code is

$$\mathrm{Lavg} = \sum_{i=0}^{3} \mathrm{Pi\ li} = 2\ (0.4 + 0.2 + 0.2 + 0.2) = 2\ \mathrm{bit/symbol}$$

Thus the efficiency of the binary code is

$$\eta = \frac{\mathrm{H}}{\mathrm{Lavg}} = \frac{1.922}{2} = 96.1\%$$

We can construct the code tree for Shannon-Fano code in several ways, depending on the splitting of symbols. This problem appears when the two groups are not of equal probability. Two possible codes are discussed. One possible code is



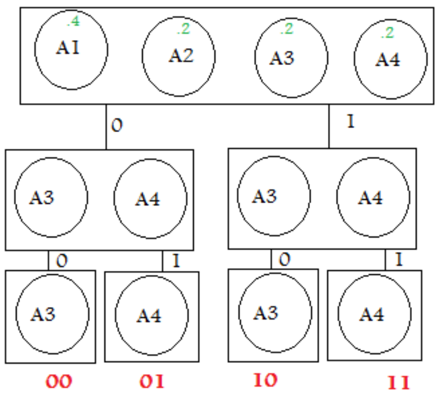
The average length of the Shannon-Fano code is

$$\mathrm{Lavg} = \sum_{i=0}^{3} \mathrm{Pi\ li} = \ 0.4 * 1 + 0.2 * 2 + 0.2 * 3 + 0.2 * 3 = 2 \text{ bit/symbol}$$

Thus the efficiency of the Shannon-Fano code is

$$\eta = \frac{H}{\mathrm{Lavg}} = \frac{1.922}{2} = 96.1\%$$

Another possible code is



The average length of the Shannon-Fano code is

$$\mathrm{Lavg} = \sum_{i=0}^{3} \mathrm{Pi\ li} = \ 0.4 * 2 + 0.2 * 2 + 0.2 * 2 + 0.2 * 2 = 2 \text{ bit/symbol}$$

Thus the efficiency of the Shannon-Fano code is

$$\eta = \frac{H}{\mathrm{Lavg}} = \frac{1.922}{2} = 96.1\%$$

All the codes in this example have the same efficiency. However, this is not the general case. As it has been demonstrated in example 1, the Shannon-Fano code has a higher efficiency than the binary code. Moreover, Shannon-Fano code can be constructed in several ways yielding different codes with different efficiencies.

Exercise 1:

The source of information A generates the symbols {A0, A1, A2, A3 and A4} with the probabilities shown in the table below. Encode the source symbols using binary encoder and Shannon-Fano encoder.

Source Symbol	P_i
A0	0.55
A1	0.25
A2	0.12
A3	0.06
A4	0.02

Compare the efficiency of both codes and comment on the results.

Exercise 2:

The source of information A generates the symbols {A0, A1, A2 and A3} with the probabilities shown in the table below. Encode the source symbols using binary encoder and Shannon-Fano encoder.

Source Symbol	P_i
A0	0.25
A1	0.25
A2	0.25
A3	0.25

Compare the efficiency of both codes and comment on the results.

Exercise 3:

The source of information A generates the symbols {A0, A1, A2, A3 and A4} with the probabilities shown in the table below. Encode the source symbols using binary encoder and Shannon-Fano encoder.

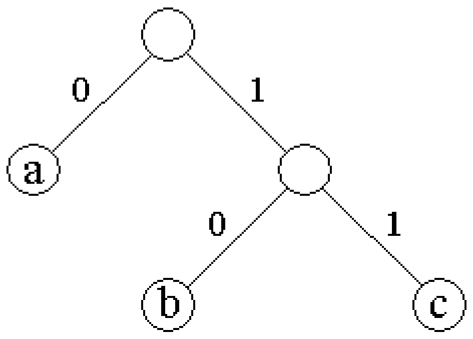
Source Symbol	P_i
A0	0.7
A1	0.15
A2	0.08
A3	0.05
A4	0.02

Compare the efficiency of both codes and comment on the results.

Code Trees:

A certain category of variable length codes can be represented by root trees. The structure of the tree defines the coding of the symbols regarded.

Code trees consist of interior nodes, leaf nodes and corresponding branches. Leaf nodes do not have a succeeding node and represent symbols, if the tree describes a prefix code. The path from the root to a leaf node defines the code word for the particular symbol assigned to this node.



Normally common compression methods use binary code trees. In that case a left branch represents a binary 0 and a right branch a binary 1.

A particular code word will be created by running from the root node to the symbol's leaf node. Any left-sided branch adds a 0 to the code word; every right-sided branch a binary 1. The required number of steps or the depth of this part of the code tree is equal to the code length.

The example mentioned above result in the following code representing the three symbols "a", "b" and "c":

0 ---> a

10 ---> b

11 ---> c