Coding for Efficiency

Let's suppose that, over some channel, we want to transmit text containing only 4 symbols, a, b, c, and d. Further, let's suppose they have a probability of occurrence in any block of text we send as follows:

\[
\begin{align*}
    a &= 0.4 \\
    b &= 0.3 \\
    c &= 0.175 \\
    d &= 0.125
\end{align*}
\]

We will also assume that these are encoded in binary using a fixed-length code similar to ASCII, although in this case 4 symbols requires only 2 bits for encoding any symbol to be sent across the channel. For a block of text containing n characters total, the total length of the message is (from [C1]):

\[
0.4 \times n \times 2 + 0.3 \times n \times 2 + 0.175 \times n \times 2 + 0.125 \times n \times 2 = 2n
\]

Clearly, this is independent of the letter frequencies. This is because no use was made of the frequency characteristics of the symbols in the encoding. It should be clear that if the more frequently occurring symbols were transmitted using shorter binary strings, the total length of the message would be shorter. Suppose that we encoded the symbols a, b, c, and d with non-ASCII codes such that:

- a is encoded using 1 binary digit
- b is encoded using 1 binary digit
- c is encoded using 2 binary digits and
- d is encoded using 2 binary digits

For instance, a = 0, b = 1, c = 10, d = 11

Now the total message length is:

\[
0.4 \times n \times 1 + 0.3 \times n \times 1 + 0.175 \times n \times 2 + 0.125 \times n \times 2 = 1.3n
\]

for a performance improvement of \(0.7n/2n = 35\%\) in information rate. A common code using this philosophy is the well-known Morse code. An examination of this code shows that letters such as E, T, and A, are encoded using much fewer dots and dashes than letters such as Q, J, and Z. (See Figure)
One measure of the efficiency of a code is the Average Symbol Length (ASL). That is, on the average, how many binary bits does it take to encode a single symbol. For the example above the result was 1.3 binary bits/symbol (recall that the total number of bits was 1.3n for n symbols in the message.) The general formula for this calculation is given by the well-known formula for a weighted average:

\[
\text{ASL} = \sum f_i l_i \quad \text{[C1]}
\]

where the \( f_i \) are the frequency ratios of each symbol and the \( l_i \) are the binary bit lengths of the corresponding symbols

**Huffman code**

The Huffman algorithm is a procedure which produces optimum codes of the sort we have been describing, having the number of digits used to encode a symbol be inversely proportional to the frequency of occurrence of that symbol. This is a tree encoding scheme, and can be implemented as follows:

1. Order all of the symbols in decreasing order of frequency of occurrence. If more than one letter have the same frequency, their order with respect to each other is irrelevant.

2. Draw a tree, starting with the leaves as follows:
   a. The two symbols with the smallest frequencies are written next to each other, and joined to a ‘node’ drawn above them with straight lines. The sum of the two frequencies is written next to this node.
b. The new node created above is added to the set of frequencies and the two symbols used so far are removed.

3. Repeat step 2, including the frequency for the new node and ignoring the frequencies of the two symbols used to create it. Note that the frequency for this node might be greater than the frequencies of the next to lowest-frequency symbols, and would not then be included in the next branch of the tree.

4. Repeat step 3, until all symbols have been incorporated into the tree, and there is only one node at the top of the tree.

5. Label the branches of the tree by labeling each of the two branches leaving a node, one with a 1 and the other with a 0. It is common to label all branches descending to the left with a 0 and those descending to the right with a 1, but this is not necessary.

6. Each symbol is now assigned a code by tracing along the branches from the symbol’s leaf to the topmost node and writing the 0’s and 1’s in order, from right to left as you travel up the tree. Equivalently, trace a path from the top single node of the tree to the symbol of interest, writing the 0’s an 1’s in order from left to right.

Example. Consider 5 symbols with frequencies as follows

\[
\begin{align*}
  a &= 0.3 \\
  b &= 0.25 \\
  c &= 0.15 \\
  d &= 0.15 \\
  e &= 0.15
\end{align*}
\]

1. The symbols are already ordered correctly

2. d and e are the first two leaves (although we could have chosen c and d or c and e), creating a new node (let’s call it x) with frequency 0.3. We now have a new frequency table

\[
\begin{align*}
  a &= 0.3 \\
  x &= 0.3 \\
  b &= 0.25 \\
  c &= 0.15
\end{align*}
\]

3. Repeating step 2 creates a new node (y) from b and c with a frequency of 0.4, resulting in the following table
y = .4
a = .3
x = .3

4. Continuing in this way

a and x combine to form w with a frequency of .6 giving
w = .6
y = .4

Combining w and y gives us our last node. The tree looks as follows

![Tree Diagram]

The coding of the five symbols, scanning up the tree and writing the binary digits from right to left, is

a = 01
b = 10
c = 11
d = 000
e = 001

Equivalently, start at the top node of the tree and trace a path to each leaf (symbol), writing down the bits from left to right as you go.
Any code generated by the Huffman algorithm has the following properties

1. It has the Prefix Property. A code has the prefix property if no codeword is the prefix of any other codeword. A code having this property is called *instantly decodable*, or an *instantaneous code*, since we do not have to wait the maximum codeword length to decide which symbol was transmitted. Suppose, in the code just generated, a was just encoded as 0 and b was encoded as 1. This would give us a more efficient code in terms of message length, but would take longer to decode, since we need to wait for enough of the message to be received to be able to decide if 0 was an a, or just the beginning of d or e. Furthermore, without additional cues, it might be difficult to tell whether 000 was d or three a’s.

2. It is not unique. Since we have certain discretion regarding which symbols to combine at each stage of the process, and how to label the branches, there are a number of equivalently efficient codes that could be generated by this process.

3. It gives the most efficient code possible within the constraints of property 1. It has the property previously discussed regarding the relationship between frequencies of the symbols and the number of bits in their encoding.

### Practice Problems - Huffman Codes

1. Each of the following is a group of codewords for a certain code (four different codes). Which of them have the prefix property? Indicate ‘yes’ or ‘no’ as appropriate.
   a. 01, 101, 000, 111
   b. 00, 011, 0111, 001
   c. 0, 01, 001
   d. 1, 01, 001
Practice Problems - Huffman Codes (cont’t)

2. Construct a binary Huffman code for the following set of symbols and their frequencies. Show the tree, labelling the branches with 1’s and 0’s as appropriate, label every node its frequency, and label all the leaves with the symbol. Write the codeword for each symbol in the space provided.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Frequency</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>.25</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>.22</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>.12</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>.12</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>.10</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>.09</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>.07</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>.03</td>
<td></td>
</tr>
</tbody>
</table>

3. What is the average codeword length for the code generated in part 3?

4. What is the theoretical minimum average codeword length for the symbol frequencies in problem 3 assuming the code does not need to have the prefix property?

5. What would be the minimum codeword length if the symbols in problem 3 were encoded using a fixed length code?