ECEN 455 Lab 2: Source Coding

Instructor: **Dr. Henry Pfister**
Email: `hpfister@tamu.edu`

Due Monday 02/27/12

Overview:

Bits are the universal unit of information in today's digital world. Any information source can be represented using bits and bits can be transmitted reliably over an unreliable physical channels. In this lab, we study how to use bits to efficiently represent a source consisting of English text. Standard English texts are composed of English letters, numbers, and special symbols. For example, many texts contain only the 26 English letters (ignoring capital letters), 10 numerical digits, and 9 special symbols including " " (space), "," (comma), "." (period), "?" (question mark), "!" (exclamation point), " ' " (apostrophe), " " " (quotation mark), "-" (hyphen), and ":" (colon). This gives a total of 45 different symbols.

A simple way to represent these 45 different symbols using bits is to assign a unique binary string of length 6 to each symbol. Text-to-bit and bit-to-text conversions can be performed using a lookup table, which records the predetermined one-to-one correspondence between the text symbols and their binary representation. This simple approach uses the same number of bits to represent each of the text symbols. This may not be very smart considering that some text symbols appear a lot more often than the others. Assigning variable-length bit strings allows one to reduce the average length. This is the basic idea behind data compression.

Exercises:

1. In this exercise, we will investigate the average length of a Huffman code.

   (a) Let's start by writing a MATLAB function that computes the average number of bits per symbol without actually constructing the code:

   $$L = \texttt{HuffmanLength}([p_1\ p_2\ \ldots\ p_m]);$$

   Let $p_1^{(j)}, \ldots, p_{m-j}^{(j)}$ be the non-increasing sequence of lumped probabilities after j steps of the code construction algorithm. At each stage, one bit is assigned to distinguish between the symbol groups associated with $p_{m-j}^{(j)}$ and $p_{m-j-1}^{(j)}$. This bit will be used with probability $p_{m-j}^{(j)} + p_{m-j-1}^{(j)}$, so the average code length is given by

   $$L = \sum_{j=0}^{m-2} \left[p_{m-j}^{(j)} + p_{m-j-1}^{(j)} \right].$$

   (b) Test this function using the source with alphabet

   $$\mathcal{X} = \{x_1, x_2, \ldots, x_9\}$$

   and corresponding probability vector

   $$p = (0.2,\ 0.15,\ 0.13,\ 0.12,\ 0.1,\ 0.09,\ 0.08,\ 0.07,\ 0.06)\,.$$

   For this example, the average code length should also be computed by hand to verify your program is correct.

(c) Discrete random variables X and Y are distributed according to:

Y / X	1	2	3
1	0.15	0.1	0.15
2	0.2	0.3	0.1

Use your MATLAB function to compare the average length of Huffman codes for coding X, Y separately and (X, Y) jointly. To code them separately, you must first compute the marginal distribution of each variable.

(d) Write a MATLAB program to read the file "constitution.txt" and count the frequency of each 7-bit ASCII value. Based on this, use your MATLAB function to compute the number of bits required to Huffman encode this file.

2. Text Compression

(a) Write a MATLAB function $[L, c] = \texttt{huffcode}(p)$ which constructs a binary Huffman code for given probability distribution $p_1, \ldots, p_m$. The function should return the average length L and a cell array c of length m which contains the codeword associated with each input symbol. Design a code for the file "constitution.txt" using the frequency counts and list the codeword and length for each ASCII value.

(b) Write a MATLAB function which encodes sequences of discrete source symbols using the Huffman code constructed in part (a). Write a matching decoder and test both on the file "constitution.txt". Does the length match your earlier predictions?

3. Extra Credit: Try building a code for "constitution.txt" based on jointly encoding two ASCII characters at a time. How many bits are required to encode the file? Is this a fair comparison?

Programming Hint:

Writing a program that designs a Huffman code can be enlightening because the algorithm is quite natural on the whiteboard but somewhat challenging to implement on a computer. The main challenge is picking the right data structure. Try using a vector $q[n]$ $(n = 1, 2, \ldots, 2m - 1)$ which keeps the probabilities of the unmerged nodes in sorted order and stores the already merged nodes at the beginning in the order they were merged.

After the k-th stage, the first $2k$ entries of the vector contain the children of the lumped probabilities and the remaining $m - k$ entries contain the lumped probabilities for k-th stage in non-decreasing order. The update for the k-th stage $(k = 1, \ldots, m - 1)$ is:

1. Notice the two smallest lumped probabilities are stored in the locations $2k - 1$ and $2k$.

2. Insert the merged probability $s = q[2k - 1] + q[2k]$ into the subvector $q[2k + 1], \ldots, q[2k + m - k - 1]$ so that it remains in non-decreasing order.

3. Keep a pointer vector r (same size as q) which points down one level in the tree. If s was inserted into location t, then $r[t] = 2k - 1$.

Finally, the codewords associated with each leaf node can be computed walking backwards through vector and concatenating zeros and ones.

MATLAB functions: Here's a short list of MATLAB commands that may be useful (use `help` to learn more): `sort, find, unique, fopen, fclose, fread, fwrite, sprintf, hist`