**U19EC045 | DCOM | LAB X**

**AIM**

Write Matlab code to implement the Huffman coding, find average codeword length and efficiency

**THEORY**

a Huffman code is a particular type of optimal [prefix code](https://en.wikipedia.org/wiki/Prefix_code" \o "Prefix code) that is commonly used for [lossless data compression](https://en.wikipedia.org/wiki/Lossless_data_compression" \o "Lossless data compression). The process of finding or using such a code proceeds by means of Huffman coding

The output from Huffman's algorithm can be viewed as a [variable-length code](https://en.wikipedia.org/wiki/Variable-length_code" \o "Variable-length code) table for encoding a source symbol (such as a character in a file). The algorithm derives this table from the estimated probability or frequency of occurrence (*weight*) for each possible value of the source symbol

The technique works by creating a [binary tree](https://en.wikipedia.org/wiki/Binary_tree" \o "Binary tree) of nodes. These can be stored in a regular [array](https://en.wikipedia.org/wiki/Array_data_type" \o "Array data type), the size of which depends on the number of symbols, {\displaystyle n}. A node can be either a [leaf node](https://en.wikipedia.org/wiki/Leaf_node" \o "Leaf node) or an [internal node](https://en.wikipedia.org/wiki/Internal_node" \o "Internal node). Initially, all nodes are leaf nodes, which contain the symbol itself, the weight (frequency of appearance) of the symbol and optionally, a link to a parent node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain a weight, links to two child nodes and an optional link to a parent node. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has up to n{\displaystyle n} leaf nodes and n-1{\displaystyle n-1} internal nodes. A Huffman tree that omits unused symbols produces the most optimal code lengths.

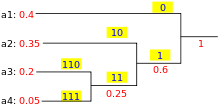
The process begins with the leaf nodes containing the probabilities of the symbol they represent. Then, the process takes the two nodes with smallest probability, and creates a new internal node having these two nodes as children. The weight of the new node is set to the sum of the weight of the children. We then apply the process again, on the new internal node and on the remaining nodes (i.e., we exclude the two leaf nodes), we repeat this process until only one node remains, which is the root of the Huffman tree.

The simplest construction algorithm uses a [priority queue](https://en.wikipedia.org/wiki/Priority_queue" \o "Priority queue) where the node with lowest probability is given highest priority:

1. Create a leaf node for each symbol and add it to the priority queue.
2. While there is more than one node in the queue:
   1. Remove the two nodes of highest priority (lowest probability) from the queue
   2. Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
   3. Add the new node to the queue.
3. The remaining node is the root node and the tree is complete.

Once the Huffman tree has been generated, it is traversed to generate a dictionary which maps the symbols to binary codes as follows:

1. Start with current node set to the root.
2. If node is not a leaf node, label the edge to the left child as *0* and the edge to the right child as *1*. Repeat the process at both the left child and the right child.

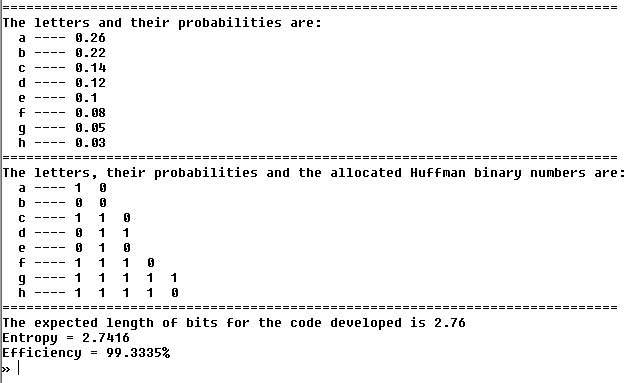


The final encoding of any symbol is then read by a concatenation of the labels on the edges along the path from the root node to the symbol.

**MATLAB CODE**

|  |
| --- |
| **clc**  **clear all;**  ***%lettes and their probabilities***  **letters=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'];**  **probabilities = [0.26, 0.22,0.14, 0.12, 0.1, 0.08, 0.05, 0.03];**  ***% cell array which includes empty vectors with index***  ***% resultCode=[][][]....***  **resultCode=cell(1,length(probabilities));**  ***%we will build the code in reverse order, at the end we will reverse it to get the actual code***  **resultCodeReversed=cell(1,length(probabilities));**  ***% [1, 2, 3, ..., n]***  **setIndices=(1:1:length(probabilities));**  ***% setContent=[1][2][3]...[n]***  **setContent=num2cell(setIndices);**  ***%coping probabilities, as it is undergoing modification***  **probablityKeeper = probabilities(:);**  ***% displying symbols and their probabilities***  **disp('=============================================================================');**  **disp('The letters and their probabilities are:')**  **for setIndex = 1:length(probablityKeeper)**  **disp(['  ',letters(setIndex), ' ---- ' ,num2str(probablityKeeper(setIndex)) ]);**  **end**  ***% loop until there is only 1 node in the tree***  **while length(setContent) > 1**  ***% sortedIndices stores the indices of the probabilities in ascending order***  **[temp, sortedIndices] = sort(probablityKeeper);**  ***% grab the node with lowest probability***  **lowestIndex = setContent{sortedIndices(1)};**  **lowestProb = probablityKeeper(sortedIndices(1));**  ***% grab the node with second lowest probability***  **secondLowest = setContent{sortedIndices(2)};**  **secondLowestProb = probablityKeeper(sortedIndices(2));**  ***% assign 0 to the left tree***  **for binaryletter\_index = 1:length(lowestIndex)**  **resultCode{lowestIndex(binaryletter\_index)} = [resultCode{lowestIndex(binaryletter\_index)}, 0];**  **end**  ***% assign 1 to the right tree***  **for binaryletter\_index = 1:length(secondLowest)**  **resultCode{secondLowest(binaryletter\_index)} = [resultCode{secondLowest(binaryletter\_index)}, 1];**  **end**    ***% remove the lowest 2 probability node from the setContent***  **setContent(sortedIndices(1:2)) = [];**  **setContent{length(setContent)+1} = [lowestIndex, secondLowest];**  ***% remove the lowest 2 probability node from the probablityKeeper***  **probablityKeeper(sortedIndices(1:2)) = [];**  ***% add the new probability node with value equal to sum of 2 lowest probability to the probablityKeeper***  **probablityKeeper(length(probablityKeeper)+1) = lowestProb + secondLowestProb;**  **end**  **disp('=============================================================================');**  **disp('The letters, their probabilities and the allocated Huffman binary numbers are:');**  ***% reverse the resultCode to get the actual code***  **for index = 1:length(resultCode)**  **resultCodeReversed{index}=fliplr(resultCode{index});**  **end**  ***% display the huffman code for each letter***  **for index = 1:length(resultCode)**  **disp(['  ', num2str(letters(index)), ' ---- ',num2str(resultCodeReversed{index})]);**  **end**  **disp('=============================================================================');**  ***% store the length of the code for each letter***  **resultCodelength=[];**  **for index = 1:length(resultCode)**  **resultCodelength=[resultCodelength,length(resultCodeReversed{index})];**  **end**  ***% display the average length of the code***  **avgLength = sum(probabilities.\*resultCodelength);**  **disp(['The expected length of bits for the code developed is ' num2str(avgLength)]);**  ***%calculate entropy***  **ent = -sum(probabilities.\*log2(probabilities));**  ***%display result***  **disp(['Entropy = ' num2str(ent)]);**  ***%calculate efficiency***  **disp(['Efficiency = ' num2str(ent\*100/avgLength) '%']);** |

**OUTPUT**



**CONCLUSION**

Hence, in this practical we have implemented Matlab code for huffman coding and also found out the average codeword length generated and also the compression efficiency.