Cairo University, Faculty of Engineering,
Electronics and Electrical Communications
Department (EECE)
Fourth Year, First Term
2025/2026

Advanced Digital Communication
Information Theory

Report1_ELC4020

Name	ID	SEC
إبراهيم رمضان إبراهيم	9220009	1
احمد خلف محمد على	9220036	1
احمد محمد إبراهيم	9220078	1
خالد محمد رمضان على	9221589	2
محمد خالد محمد شتا	9203230	3

Submitted to: **Dr. Mohamed Nafie**

& Eng. Mohamed Khaled

Submission date: **2-11-2025**

Table of Contents

Introduction

In digital communication systems, the efficient use of bandwidth and storage is essential. Source coding, also known as data compression, is a fundamental technique used to reduce the number of bits required to represent information without losing its essential meaning. The main objective of source coding is to eliminate redundancy in the original data so that it can be transmitted or stored more efficiently.

Source coding plays a vital role in modern communication and multimedia applications such as image compression (JPEG), audio compression (MP3), and video compression (H.264). By minimizing the average number of bits per symbol, source coding enables higher data transmission rates and more efficient utilization of available resources.

Huffman Code

The Huffman coding algorithm provides an optimal method for constructing prefix-free variable-length codes based on symbol probabilities. It begins by arranging the source symbols in descending order of probability and then iteratively combines the two least probable symbols into a single composite node whose probability equals their sum. This process continues until a single root node is formed, representing the entire source. Binary digits '0' and '1' are then assigned along the branches of the resulting binary tree to generate the unique codewords for each symbol. The procedure ensures that symbols with higher probabilities receive shorter codes, achieving the minimum possible average code length consistent with the source entropy.

1. MATLAB Implementation Code

```
%-------------------------- Input Section -----------------------------%
% Ask user to enter probabilities for which to generate Huffman code
% Our Example: [0.35 0.30 0.20 0.10 0.04 0.005 0.005]
% Note: The sum of probabilities must equal 1
Input_Message = input('Please Enter The Given Probabilities: ');
```

In this section, we let the user provides the set of symbol probabilities. These probabilities represent the statistical Probabilities of each symbol in the source message.
It is crucial that the sum equals 1 to ensure a valid probability distribution. Our example, [0.35 0.30 0.20 0.10 0.04 0.005 0.005] corresponds to seven symbols with descending likelihoods.

```
%------------------------ Initialize Arrays ---------------------------%
% Create a zero matrix 'Arr' with:
%   rows = number of symbols
%   columns = number of symbols - 1
% Store the input probabilities (transposed) in the first column of Arr.
% Store the length of the input vector in 'Length'.
Arr = zeros(length(Input_Message), length(Input_Message) - 1);
Arr(:, 1) = transpose(Input_Message);
Length = length(Input_Message);
```

We initialize the data structure used to construct the Huffman tree.

A matrix Arr is created to hold all the intermediate probability combinations at each iteration.

Each column represents a stage in the tree-building process, with the first column containing the original probabilities.

Length stores the number of current active symbols.

```
%------------------------ Build Probability Tree ----------------------%
% Arrange probabilities in descending order and iteratively combine the
% smallest two until one root node remains.
for i = 1:length(Input_Message)
    Arr(:, i) = sort(Arr(:, i), 'descend');

    % Stop when reaching the last column
    if i == length(Input_Message) - 1
        break;
    end

    % Combine smallest two probabilities
    Arr(1:Length - 2, i + 1) = Arr(1:Length - 2, i);
    Arr(Length - 1, i + 1) = Arr(Length, i) + Arr(Length - 1, i);
    Length = Length - 1;
end
```

This loop builds the hierarchical structure of the Huffman tree.

In each iteration, it merges the two smallest probabilities into a new node whose probability equals their sum.

The matrix is then updated, reducing the total number of nodes by one.

This process repeats until only a single root node remains — with the sum of all probabilities equal to 1 — representing the complete tree.

```
%---------------------- Initialize Huffman Code Table -------------------%
% Determine the size of Arr
N = size(Arr);
P = N(1, 2);    % Number of columns

% Initialize counters
L = 3;
M = 2;

% Create a string array for Huffman codes
Huffman_Source_Code = strings(size(Arr));

% Initialize last column codes ("0" and "1")
Huffman_Source_Code(1, P) = "0";
Huffman_Source_Code(2, P) = "1";
```

After constructing the probability tree, a table is initialized to store the binary codes.

Each row represents a symbol, while each column corresponds to a step in the code construction process.

In the final column, binary digits ‘0’ and ‘1’ are assigned to the two nodes combined last.

From this point, the codes are traced backward through the tree toward the root.

```
%----------------------- Huffman Encoding Loop -------------------------%
% Build Huffman codes column by column from right to left.
for k = P - 1:-1:1
    q = -1;

    % Search for matching combined probabilities
    for u = 1:M
        if (Arr(L, k) + Arr(L - 1, k) == Arr(u, k + 1))
            q = u;
            Huffman_Source_Code(L - 1, k) = strcat(Huffman_Source_Code(u, k + 1),
"0");
            Huffman_Source_Code(L, k)     = strcat(Huffman_Source_Code(u, k + 1),
"1");
            break;
        end
    end

    % Update codes for the remaining symbols
    j = 1;
    for y = 1:L - 2
        if (y == q)
            j = j + 1;
        end
        Huffman_Source_Code(y, k) = Huffman_Source_Code(j, k + 1);
        j = j + 1;
    end

    % Increment counters
    M = M + 1;
    L = L + 1;
end
```

This is the core of our Huffman encoding process.

It reconstructs binary codes by tracing the merged steps in reverse order — from the last combined node back to the initial probabilities.

Each time two nodes were combined, one receives a ‘0’ and the other a ‘1’.
This ensures that each final symbol obtains a unique prefix-free code.

```
%-------------------------- Display Results ----------------------------%
disp('Sorted Probabilities (Arr):');
disp(Arr);
disp('Generated Huffman Codes:');
disp(Huffman_Source_Code);
```

Here, we display the final Huffman table, showing both the sorted probability structure (Arr) and the resulting binary codewords for each symbol.

These outputs confirm that higher-probability symbols have shorter codewords, demonstrating compression efficiency.

```
%------------------ Entropy, Average Length, Efficiency ------------------%
% Entropy = Σ P(i) * log2(1/P(i))
Entropy_of_symbols = 0;
for i = 1:length(Input_Message)
    Entropy_of_symbols = Entropy_of_symbols + (-Input_Message(i) *
log2(Input_Message(i)));
end
disp(['Entropy: H(X) = ', num2str(Entropy_of_symbols), ' bits/symbol']);

% Average code length = Σ P(i) * L(i)
Avg_Len = 0;
for i = 1:length(Input_Message)
    Avg_Len = Avg_Len + length(Huffman_Source_Code{i, 1}) * Input_Message(i);
end
disp(['Average length: L(X) = ', num2str(Avg_Len), ' bits/symbol']);

% Efficiency = (Entropy / Average length) * 100%
efficiency = (Entropy_of_symbols / Avg_Len) * 100;
disp(['Efficiency = ', num2str(efficiency), '%']);
%**************************************************************************%
```

Last section, we evaluate the performance of the Huffman code:

- Entropy represents the theoretical lower bound of the average code length.
- Average length measures the actual bits per symbol in the generated code.
- Efficiency is calculated as $\eta = \frac{H(X)}{L(X)} \times 100\%$, showing how close the coding scheme is to the theoretical optimum.
- We found that Huffman coding achieves efficiency values close to 100%, reflecting its optimality as expected.

1. Theoretical Results:

We validate the algorithm by analyzing probability distributions through manual calculation.

<table>
<tr><th>Symbols</th><th colspan="2">P</th><th colspan="2">P1</th><th colspan="2">P2</th><th colspan="2">P3</th><th colspan="2">P4</th><th colspan="2">P5</th></tr>
<tr><td>A</td><td>0.35</td><td>00</td><td>0.35</td><td>00</td><td>0.35</td><td>00</td><td>0.35</td><td>00</td><td>0.35</td><td>1</td><td>0.65</td><td>0</td></tr>
<tr><td>B</td><td>0.3</td><td>01</td><td>0.3</td><td>01</td><td>0.3</td><td>01</td><td>0.3</td><td>01</td><td>0.35</td><td>00</td><td>0.35</td><td>1</td></tr>
<tr><td>C</td><td>0.2</td><td>10</td><td>0.2</td><td>10</td><td>0.2</td><td>10</td><td>0.2</td><td>10</td><td>0.3</td><td>01</td><td colspan="2" rowspan="4">Trans: F+G+E+D+C+B</td></tr>
<tr><td>D</td><td>0.1</td><td>110</td><td>0.1</td><td>110</td><td>0.1</td><td>110</td><td>0.15</td><td>11</td><td colspan="2" rowspan="3">Trans: F+G+E+D +C</td></tr>
<tr><td>E</td><td>0.04</td><td>1110</td><td>0.04</td><td>1110</td><td>0.05</td><td>111</td><td colspan="2" rowspan="3">Trans: F+G+E+D</td></tr>
<tr><td>F</td><td>0.005</td><td>11110</td><td>0.01</td><td>1111</td><td colspan="2" rowspan="2">Trans: F+G+E</td></tr>
<tr><td>G</td><td>0.005</td><td>11111</td><td colspan="2">Trans: F+G</td></tr>
</table>

$$H(x) = -\sum P(X_i)\ \log_2 P(X_i)$$

$$H(x) = -(0.35 \times \log_2 0.35 + 0.30 \times \log_2 0.30 + 0.2 \times \log_2 0.2 + 0.1 \times \log_2 0.1 + 0.04 \times \log_2 0.04 + 0.005 \times \log_2 0.005 + 0.005 \times \log_2 0.005) = 2.109962$$

$$L(C) = \sum P(X_i)\ l_i$$

$$L(C) = 0.35 \times 2 + 0.30 \times 2 + 0.2 \times 2 + 0.1 \times 3 + 0.04 \times 4 + 0.005 \times 5 + 0.005 \times 5 = 2.21$$

$$\eta = \frac{H(x)}{L(C)} = \frac{2.109962}{2.21} = 95.47\%$$

3. Simulation Results:

Please Enter The Given Probabilities: [0.35 0.30 0.20 0.10 0.04 0.005 0.005]
Sorted Probabilities (Arr):

0.3500	0.3500	0.3500	0.3500	0.3500	0.6500
0.3000	0.3000	0.3000	0.3000	0.3500	0.3500
0.2000	0.2000	0.2000	0.2000	0.3000	0
0.1000	0.1000	0.1000	0.1500	0	0
0.0400	0.0400	0.0500	0	0	0
0.0050	0.0100	0	0	0	0
0.0050	0	0	0	0	0

Generated Huffman Codes:

"00"	"00"	"00"	"00"	"1"	"0"
"01"	"01"	"01"	"01"	"00"	"1"
"10"	"10"	"10"	"10"	"01"	""
"110"	"110"	"110"	"11"	""	""
"1110"	"1110"	"111"	""	""	""
"11110"	"1111"	""	""	""	""
"11111"	""	""	""	""	""

Entropy: H(X) = 2.11 bits/symbol
Average length: L(X) = 2.21 bits/symbol
Efficiency = 95.4734%

- As shown in the results, we can see that they are identical to the results from the hand analysis. However, this result is not unique, because in the above case we assigned the maximum probability (0.65) to zero and the minimum probability (0.35) to one. If we instead assigned the maximum probability to one and the minimum to zero, we could obtain another valid solution.

Fano Code

The Fano coding algorithm provides a systematic method for constructing uniquely decodable codes by recursively dividing a probability-ordered symbol set into two subsets of approximately equal cumulative probability. At each partition, symbols in one subset are assigned a leading bit of '0' while the other subset receives '1', with the process continuing recursively until each symbol is uniquely encoded.

1. MATLAB Implementation Code

```
n = input('Enter number of symbols: ');
symbols = cell(1, n);
for i = 1:n
    symbols{i} = input(sprintf('Enter symbol %d: ', i), 's');
end

p = zeros(1, n);
for i = 1:n
    p(i) = input(sprintf('Enter probability of %s: ', symbols{i}));
end

%% step1: Sort  in descending order
[p, idx] = sort(p, 'descend');
symbols = symbols(idx);
```

The algorithm begins by collecting symbols and their probabilities via user input, normalizing them, and sorting them in descending order of probability

```
%% Our Example
symbols = {'a', 'b', 'c', 'd', 'e', 'f', 'g'};
p   = [0.35, 0.3, 0.2, 0.1, 0.04, 0.005, 0.005];
p = p / sum(p);
[p, idx] = sort(p, 'descend');
symbols = symbols(idx);
```

For a quick test, we provide a ready-made version of the code with predefined symbols and probabilities as an example.

```
function codes = recursive_fano(symbols, p)
    n = length(symbols);
    codes = cell(1, n);
    if n == 1
        codes{1} = '';
        return;
    end
    % step2: split to 2 groups nearly equal prabilities.
    total = sum(p);
    diff = abs(cumsum(p) - total/2);
    [~, split] = min(diff);
    % Recursively apply the Fano algorithm to the right and left subset
    % Step 3: Assign the first one as 0 and the second one as 1.
    % Step 4:Repeat step 2 but in Assigning use the code be for last grouping
    % and assign 0 or 1 in the least SB.
    % Step 5: Repeat 3&4 until each group contain one symbole.
    left = recursive_fano(symbols(1:split), p(1:split));
    right = recursive_fano(symbols(split+1:end), p(split+1:end));
    for i = 1:length(left)
        codes{i} = ['0' left{i}];
    end
    for i = 1:length(right)
        codes{split+i} = ['1' right{i}];
    end
end
```

The recursive Fano function begins by determining the optimal partition point that divides the symbol set into two subsets of approximately equal cumulative probability. It then recursively applies the same procedure to each subset, with left partition codes prepended by '0' and right partition codes prepended by '1'. This process continues until the base case is reached, where a single symbol receives an empty string, after which the binary prefixes accumulate through the recursive call stack to form the final codewords.

```
fprintf('\nSymbol   Probability   Fano Code\n');
for i = 1:length(symbols)
    fprintf('  %-6s    %.3f        %s\n', symbols{i}, p(i), codes{i});
end
%% calculate Entropy and Efficiency
H = -sum(p .* log2(p));    % entropy H = -sum(p_i * log2(p_i))
L = 0;
for i = 1:length(codes)
    L = L + p(i) * length(codes{i});  % average code length L = sum( p_i * l_i)
end
eta = H / L;
fprintf('\nEntropy (H): %.4f bits/symbol\n', H);
fprintf('Average Code Length (L): %.4f bits/symbol\n', L);
fprintf('Coding Efficiency (η = H/L): %.2f%%\n', eta * 100);
```

The results are displayed in a formatted table showing symbols, probabilities, and assigned codes. Entropy H, average code length L, and coding efficiency η (H/L ratio) are computed to evaluate compression performance.

2. Theoretical Results:

We validate the algorithm by analyzing probability distributions through manual calculation.

$$H(x) = -\sum P(X_i)\ \log_2 P(X_i)$$

Symbol [S]	P	S	P	C	P	C	P	C	P	C	P	C
a	0.35	a	0.35		0.35	00						
b	0.30	b	0.30	0	0.30	01						
c	0.20	c	0.20		0.20	10						
d	0.10	d	0.10		0.10		0.10	110				
e	0.04	e	0.04	1	0.04	11	0.04		0.04	1110		
f	0.005	f	0.005		0.005		0.005	111	0.005	1111	0.005	11110
g	0.005	g	0.005		0.005		0.005		0.005		0.005	11111

$$H(x) = -(0.35 \times \log_2 0.35 + 0.30 \times \log_2 0.30 + 0.2 \times \log_2 0.2 + 0.1 \times \log_2 0.1 + 0.04 \times \log_2 0.04 + 0.005 \times \log_2 0.005 + 0.005 \times \log_2 0.005) = 2.109962$$

$$L(C) = \sum P(X_i)\ l_i$$

$$L(C) = 0.35 \times 2 + 0.30 \times 2 + 0.2 \times 2 + 0.1 \times 3 + 0.04 \times 4 + 0.005 \times 5 + 0.005 \times 5 = 2.21$$

$$\eta = \frac{H(x)}{L(C)} = \frac{2.109962}{2.21} = 95.47\%$$

3. Simulation Results:

```
Symbol   Probability   Fano Code
  a        0.350        00
  b        0.300        01
  c        0.200        10
  d        0.100        110
  e        0.040        1110
  f        0.005        11110
  g        0.005        11111
Entropy (H): 2.1100 bits/symbol
Average Code Length (L): 2.2100 bits/symbol
Coding Efficiency (η = H/L): 95.47%
```

- The obtained result matches the hand analysis, however, the Fano algorithm can generate different valid codes.

Full MATLAB Code:

Full Huffman Code

```
%--------------------------- Huffman Algorithm -------------------------%
clear;
clc;
close all;

%-------------------------- Input Section ------------------------------%
% Ask user to enter probabilities for which to generate Huffman code
% Our Example: [0.35 0.30 0.20 0.10 0.04 0.005 0.005]
% Note: The sum of probabilities must equal 1
Input_Message = input('Please Enter The Given Probabilities: ');

%------------------------ Initialize Arrays ---------------------------%
% Create a zero matrix 'Arr' with:
%   rows = number of symbols
%   columns = number of symbols - 1
% Store the input probabilities (transposed) in the first column of Arr.
% Store the length of the input vector in 'Length'.
Arr = zeros(length(Input_Message), length(Input_Message) - 1);
Arr(:, 1) = transpose(Input_Message);
Length = length(Input_Message);

%------------------------ Build Probability Tree ----------------------%
% Arrange probabilities in descending order and iteratively combine the
% smallest two until one root node remains.
for i = 1:length(Input_Message)
    Arr(:, i) = sort(Arr(:, i), 'descend');

    % Stop when reaching the last column
    if i == length(Input_Message) - 1
        break;
    end

    % Combine smallest two probabilities
    Arr(1:Length - 2, i + 1) = Arr(1:Length - 2, i);
    Arr(Length - 1, i + 1) = Arr(Length, i) + Arr(Length - 1, i);
    Length = Length - 1;
end

%---------------------- Initialize Huffman Code Table ------------------%
% Determine the size of Arr
N = size(Arr);
P = N(1, 2);    % Number of columns

% Initialize counters
L = 3;
M = 2;

% Create a string array for Huffman codes
Huffman_Source_Code = strings(size(Arr));
```

```
% Initialize last column codes ("0" and "1")
Huffman_Source_Code(1, P) = "0";
Huffman_Source_Code(2, P) = "1";

%----------------------- Huffman Encoding Loop -------------------------%
% Build Huffman codes column by column from right to left.
for k = P - 1:-1:1
    q = -1;

    % Search for matching combined probabilities
    for u = 1:M
        if (Arr(L, k) + Arr(L - 1, k) == Arr(u, k + 1))
            q = u;
            Huffman_Source_Code(L - 1, k) = strcat(Huffman_Source_Code(u, k + 1),
"0");
            Huffman_Source_Code(L, k)     = strcat(Huffman_Source_Code(u, k + 1),
"1");
            break;
        end
    end

    % Update codes for the remaining symbols
    j = 1;
    for y = 1:L - 2
        if (y == q)
            j = j + 1;
        end
        Huffman_Source_Code(y, k) = Huffman_Source_Code(j, k + 1);
        j = j + 1;
    end

    % Increment counters
    M = M + 1;
    L = L + 1;
end

%-------------------------- Display Results ----------------------------%
disp('Sorted Probabilities (Arr):');
disp(Arr);
disp('Generated Huffman Codes:');
disp(Huffman_Source_Code);

%------------------ Entropy, Average Length, Efficiency ------------------%
% Entropy = Σ P(i) * log2(1/P(i))
Entropy_of_symbols = 0;
for i = 1:length(Input_Message)
    Entropy_of_symbols = Entropy_of_symbols + (-Input_Message(i) *
log2(Input_Message(i)));
end
disp(['Entropy: H(X) = ', num2str(Entropy_of_symbols), ' bits/symbol']);

% Average code length = Σ P(i) * L(i)
Avg_Len = 0;
for i = 1:length(Input_Message)
    Avg_Len = Avg_Len + length(Huffman_Source_Code{i, 1}) * Input_Message(i);
```

```
end
disp(['Average length: L(X) = ', num2str(Avg_Len), ' bits/symbol']);

% Efficiency = (Entropy / Average length) * 100%
efficiency = (Entropy_of_symbols / Avg_Len) * 100;
disp(['Efficiency = ', num2str(efficiency), '%']);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Full Fano Code

```
clc;
clear;
close all;

%%
%n = input('Enter number of symbols: ');
%symbols = cell(1, n);
%for i = 1:n
%    symbols{i} = input(sprintf('Enter symbol %d: ', i), 's');
%end

%p = zeros(1, n);
%for i = 1:n
%    p(i) = input(sprintf('Enter probability of %s: ', symbols{i}));
%end
%%%

%% Our Example
symbols = {'a', 'b', 'c', 'd', 'e', 'f', 'g'};
p   = [0.35, 0.3, 0.2, 0.1, 0.04, 0.005, 0.005];


p = p / sum(p);
%% step1: Sort  in descending order
[p, idx] = sort(p, 'descend');
symbols = symbols(idx);

% Call recursive_fano function
fano_encode = @(sym, prob) recursive_fano(sym, prob);
codes = fano_encode(symbols, p);

fprintf('\nSymbol   Probability   Fano Code\n');
for i = 1:length(symbols)
    fprintf('  %-6s    %.3f        %s\n', symbols{i}, p(i), codes{i});
end
%% calculate Entropy and Efficiency
H = -sum(p .* log2(p));    % entropy H = -sum(p_i * log2(p_i))
L = 0;
for i = 1:length(codes)
    L = L + p(i) * length(codes{i});  % average code length L = sum( p_i * l_i)
end
```

```
eta = H / L;

fprintf('\nEntropy (H): %.4f bits/symbol\n', H);
fprintf('Average Code Length (L): %.4f bits/symbol\n', L);
fprintf('Coding Efficiency (η = H/L): %.2f%%\n', eta * 100);


%% ******************** Fano Code Generation Function *****************
function codes = recursive_fano(symbols, p)
    n = length(symbols);
    codes = cell(1, n);
    if n == 1
        codes{1} = '';
        return;
    end
    % step2: split to 2 groups nearly equal prabilities.
    total = sum(p);
    diff = abs(cumsum(p) - total/2);
    [~, split] = min(diff);
    % Recursively apply the Fano algorithm to the right and left subset
    % Step 3: Assign the first one as 0 and the second one as 1.
    % Step 4:Repeat step 2 but in Assigning use the code be for last grouping
    % and assign 0 or 1 in the least SB.
    % Step 5: Repeat 3&4 until each group contain one symbole.
    left = recursive_fano(symbols(1:split), p(1:split));
    right = recursive_fano(symbols(split+1:end), p(split+1:end));
    for i = 1:length(left)
        codes{i} = ['0' left{i}];
    end
    for i = 1:length(right)
        codes{split+i} = ['1' right{i}];
    end
end
```