Electronics and Electrical Communications Engineering Department

Faculty of Engineering

Cairo University

Implementation and Comparative Analysis of Huffman and Fano Source Coding Algorithms

ELC4020 “Advanced Communication Systems “

4th Year

1st Semester - Academic Year 2025/2026

Prepared by:

NAME	SECTION	ID
Mohamed Ahmed Abd El Hakam	3	9220647
Yousef Khaled Omar Mahmoud	4	9220984
Ahmed Wagdy Mohy Ibrahim	1	9220120
Abdelrahman Essa Elsayed	2	9220469

Instructor: Eng. Mohamed Khaled
Dr. Mohammed Nafie & Dr. Mohamed Khairy

1. Contents

2. Table of Figures

3. Introduction

Source Coding is a fundamental area of information theory that deals with the representation of data generated by a source. The primary goal is to reduce the average number of bits required to represent the source symbols, making data transmission and storage more efficient.[1]

Source coding is vital because it addresses the inherent redundancy in information sources. By assigning shorter codewords to frequently occurring symbols and longer codewords to less frequent ones, it achieves significant benefits:

1. **Efficiency: It minimizes the bandwidth or storage space needed to convey information, leading to faster transmission and lower costs.**
2. **Information Theory: It provides a practical way to approach the theoretical limit of compression, known as the source entropy (the minimum average number of bits needed per symbol).**
3. **Ubiquity: It's the underlying principle for almost all digital file formats and communication protocols, including ZIP files, JPEG images, MP3 audio, and wireless communication standards.**

In this project we implement and compare between two classic variable-length source coding algorithms, Huffman and Fano

4. Huffman Source Coding

4.1. Algorithm Overview [2]

Step 1: Sort all the N_s symbols based on their probability in descending order(من الكبير للصغير). ⟶

Step 2: Crate a new colume has $(N_s - 1)$ by Sum the two symbols with the smallest probabilities.

Step 3: Repeat step 1 and 2 utill only two symbols remain.

Step 4: Assing codes for: the first as 0 and the second as 1. ⟵

Step 5: go step back to Left coalum and copy all non changed proabiltes codes as right colum (exept the lowst two in the left they alrady add in step 3), the chagned ones thier codes will be the same code of their summtion but zero is added in the least significant bit for the first and one for the second .

Huffman Algorithm

4.2. Hand Analysis

Symbol	P0	C0	P1	C1	P2	C2	P3	C3	P4	C4	P5	C5
A	0.35	00	0.35	00	0.35	00	0.35	00	0.35	1	0.65	0
B	0.3	01	0.3	01	0.3	01	0.3	01	0.35	00	0.35	1
C	0.2	10	0.2	10	0.2	10	0.2	10	0.3	01		
D	0.1	110	0.1	110	0.1	110	0.15	11				
E	0.04	1110	0.04	1110	0.05	111						
F	0.005	11110	0.01	1111								
G	0.005	11111										

So, here's the output that we are aiming for.

With the Kraft's Sum, entropy, average length and efficiency:

$$\textit{The Entropy of the Source} \quad H(P(x)) = -\sum_{xi} P(xi)\log_2 P(xi)$$

$$\textit{The Length of the Codeword} \quad L(C) = -\sum_{xi} P(xi)L(xi)$$

$$\textit{Efficiency} \quad \eta = \frac{H(P(x))}{L(C)} * 100\%$$

$$\textit{Kraft's Sum} = \sum_{Li} 2^{-Li} \qquad \textit{Kraft's Inequality}: \textit{Kraft's Sum} <= 1$$

So, by calculating them we can find that:

Entropy of the code

$$H(P(x)) = 2.11\ bits/symbol$$

Average codeword length

$$L(C) = 2.21\ bits/symbol$$

So the overall efficiency is $\eta = \frac{L(C)}{H(P(x))} * 100\% = \frac{2.11}{2.21} * 100\% = 95.475\%$

And Kraft's Sum = 1

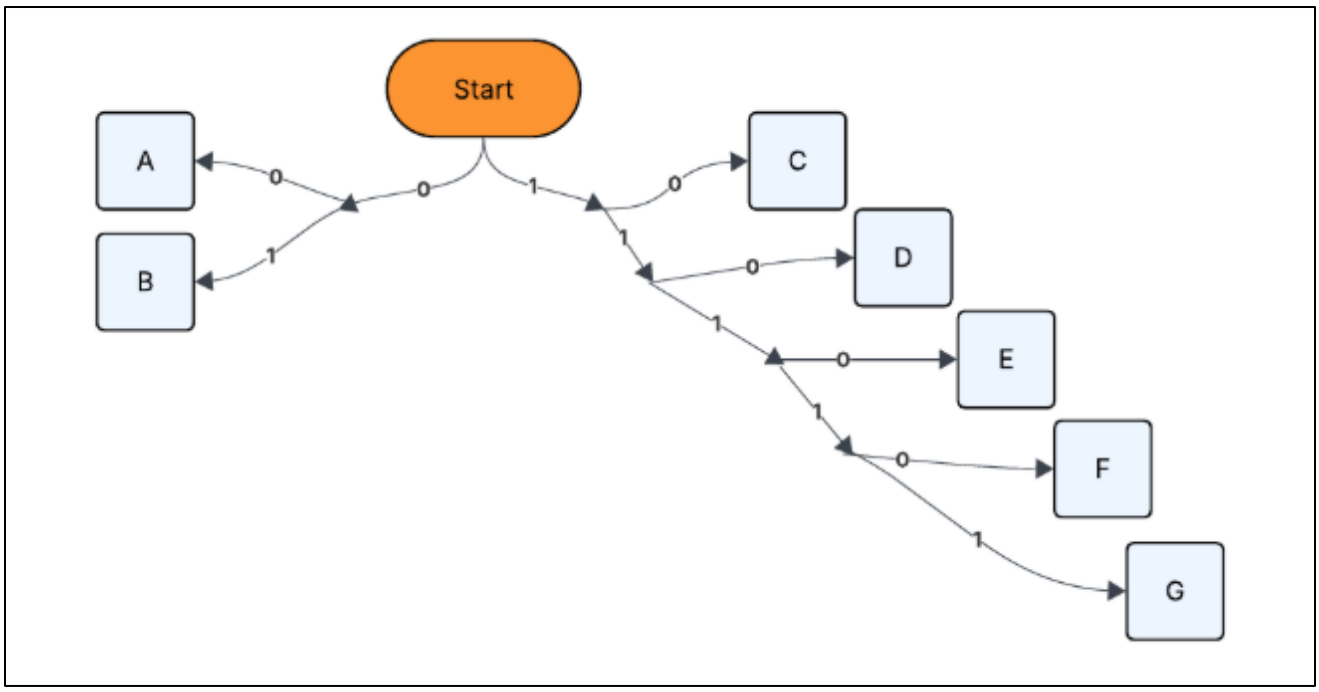

Figure 1 Huffman Output Kraft's Tree

4.3. Conclusion

The results show that **Huffman coding provides a relatively efficient prefix-free code**, with an **average code length (2.21 bits)** that is **close to but slightly higher than the entropy (2.11 bits)**. Also, since the **Kraft inequality** is satisfied with equality, the Huffman code is **optimal** among all binary prefix codes for this source.

And also from the **Huffman Kraft tree**, it can be observed that **no codeword is assigned to an internal node**, meaning that no codeword is the prefix of another. Therefore, the Huffman code is an **instantaneous (prefix-free) code**, ensuring **unique and immediate decodable** without the need for lookahead.

4.4. MATLAB Implementation [3]

4.4.1. Calculations Functions

First, we got the calculations Functions:

```
%% ------------------------------------------------------------------------
%               Entropy Calculation
% ------------------------------------------------------------------------

function H = entropy_calc(P)
%ENTROPY_CALC Compute the source entropy H(P(x))
%   H = entropy_calc(P)
%   P : vector of symbol probabilities
%   H : entropy in bits

    % Validate input
    if any(P < 0) || abs(sum(P) - 1) > 1e-6
        warning('Probabilities should sum to 1. Normalizing...');
        P = P / sum(P);
    end

    % Remove zeros (to avoid log2(0))
    P = P(P > 0);

    % Compute entropy
    H = -sum(P .* log2(P));
end
```

Figure 2 Entropy Calculation

And

```
%% ------------------------------------------------------------------------
%              Average Length Calculation
% ------------------------------------------------------------------------
function L = average_length_calc(dict)
%AVERAGE_LENGTH_CALC Compute average codeword length L(C)
%   L = average_length_calc(dict, P)
%   dict : Huffman dictionary cell array {symbol, code}
%   P : vector of symbol probabilities (same order as dict)
%
%   If P is empty, it tries to extract from dict(:,2) if present
%   L : average code length


    % --- Compute code lengths ---
    code_lengths = cellfun(@length, codes);

    % --- Normalize probabilities ---
    P = P(:) / sum(P);

    % --- Check dimensions ---
    if length(P) ~= length(code_lengths)
        error('Number of probabilities does not match number of codewords.');
    end

    % --- Compute average code length ---
    L = sum(P .* code_lengths);
end
```

Figure 4 Average length Calculation

```
%% ------------------------------------------------------------
%               Efficiency Calculation
% ------------------------------------------------------------

function eta = efficiency_calc(H, L)
%EFFICIENCY_CALC Compute Huffman coding efficiency η
%    eta = efficiency_calc(H, L)
%    H : entropy
%    L : average code length
%    eta : efficiency in percentage (%)

    if L <= 0
        error('Average length L must be positive.');
    end

    eta = (H / L) * 100;
end
```

Figure 3 Efficiency Calculation

And

```
%% ------------------------------------------------------------------------
%                Print Kraft Inequality Function
% ------------------------------------------------------------------------
function [kraft_sum, kraft_flag]=kraft_analysis(dict)
% KRAFT_ANALYSIS  Compute Kraft's inequality and visualize Kraft tree
%   kraft_analysis(dict)
%
%   Input:
%       dict : cell array {N x 3}  → {symbol, P, code}
%
%   Example:
%       dict = {'A','0'; 'B','10'; 'C','110'; 'D','111'};
%       kraft_analysis(dict);



    % --- 1. Compute Kraft's inequality ---
    code_lengths = cellfun(@length, codes);
    kraft_sum = sum(2.^(-code_lengths));

    fprintf('\n=== Kraft Inequality Check ===\n');
    fprintf('Sum(2^{-l_i}) = %.4f\n', kraft_sum);

end
```

Figure 5 Kraft's Inequality

```
% Combine into dictionary-like cell array
dict_input = [symbols(:), num2cell(P(:))];

% Assume not great until great
err_flag = 1;

% Validate using the check_symbols() function
[ok, msg] = check_symbols(dict_input);

% Display validation result
if ok
    disp('✅ Dictionary is valid!');
    err_flag =0;
else
    disp(['❌ Error: ' msg]);
    err_flag = 1;
end

% ------------------------------------------------------------------------
% Compute entropy (only if valid)
% ------------------------------------------------------------------------
H = entropy_calc(P)
```

Figure 6 Input Validation

So, the inputs are the following

--- Input Symbol Dictionary ---

	Symbol	Probability
1	A	0.3500
2	B	0.3000
3	C	0.2000
4	D	0.1000
5	E	0.0400
6	F	0.0050
7	G	0.0050

Entropy: H = 2.1100 bits/symbol

Figure 7 Input Symbols

And then we apply Huffman functions as follows, we merge the probabilities

```
% --- Iteratively merge ---
for step = 2:numCols
    % Sort ascending to pick smallest two
    curP = sort(curP, 'ascend');
    if numel(curP) >= 2
        p1 = curP(1);
        p2 = curP(2);
        mergedP = p1 + p2;
        % Remove two smallest and add merged one
        curP = [mergedP; curP(3:end)];
    end
    % Sort descending for display
    curP = sort(curP, 'descend');

    % Fill current column
    for r = 1:maxRows
        if r <= numel(curP)
            history{r,step} = curP(r);
        else
            history{r,step} = NaN;
        end
    end
end
```

Figure 8 Probability Merge

So, we got:

Step 1: Huffman Probability Merging Process

P0	P1	P2	P3	P4	P5
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	
0.1000	0.1000	0.1000	0.1500		
0.0400	0.0400	0.0500			
0.0050	0.0100				
0.0050					

Figure 9 Merged Probabilities

Then we assign the codes from last column to first

```
% === Assign child codes ===
% Last two rows in previous P column are merged into this parent
history_table_full(raw_counter, prevCcol)   = [parentCode '0'];
history_table_full(raw_counter+1, prevCcol) = [parentCode '1'];

% For each previous non-merged row (in display order top->bottom)
for ii = 1:(raw_counter-1)
    % Skip the rows that were just merged (raw_counter and raw_counter+1)

    % Get the probability value in the previous column for this row
    valPrev = history_table_full(ii, prevPcol);
    if isnan(valPrev)
        continue; % nothing to copy
    end

    % Find matching value in the current column (exclude merged parent)
    currMatches = find(abs(currPvals - valPrev) < 1e-12);

    % Remove the matchIdx (the merged parent) if it appears
    currMatches(currMatches == matchIdx) = [];

    if isempty(currMatches)
        continue; % no corresponding match found
    end

    % If there are duplicates (two identical probabilities)
    if numel(currMatches) > 1
        % take both, and copy their codes to the two rows
        history_table_full(ii,   prevCcol) = currCvals(currMatches(1));
        if (ii+1) <= numRows
            history_table_full(ii+1, prevCcol) = currCvals(currMatches(2));
        end
    else
        % single match — copy code directly
        history_table_full(ii, prevCcol) = currCvals(currMatches(1));
    end
```

Figure 10 Code Assignment

and the output after applying the steps is:

Huffman Encoding: Probability and Code Evolution (P/C Steps)

P0	C0	P1	C1	P2	C2	P3	C3	P4	C4	P5	C5
0.3500	00	0.3500	00	0.3500	00	0.3500	00	0.3500	1	0.6500	0
0.3000	01	0.3000	01	0.3000	01	0.3000	01	0.3500	00	0.3500	1
0.2000	10	0.2000	10	0.2000	10	0.2000	10	0.3000	01		
0.1000	110	0.1000	110	0.1000	110	0.1500	11				
0.0400	1110	0.0400	1110	0.0500	111						
0.0050	11110	0.0100	1111								
0.0050	11111										

Figure 11 Huffman Steps

4.4.2. Theoretical vs practical

--- Huffman Coded Dictionary ---

	Symbol	Probability	Code
1	A	0.3500	00
2	B	0.3000	01
3	C	0.2000	10
4	D	0.1000	110
5	E	0.0400	1110
6	F	0.0050	11110
7	G	0.0050	11111

H = 2.1100 | L = 2.2100 | η = 95.47 % | Kraft = 1.0000

Figure 12 Practical Results

The MATLAB simulations produced results identical to the Theoretical one

5. Fano Source Coding

Fano source coding, also known as **Shannon–Fano coding**, is an early method of **variable-length, prefix-free coding** used in lossless data compression. The algorithm works by **dividing the set of symbols into two groups** with nearly equal total probabilities and assigning binary digits (‘0’ and ‘1’) to each group recursively.

Although Fano coding produces a **prefix-free and uniquely decodable code**, it is **not always optimal**, as its average code length can be slightly higher than that of Huffman coding. Nonetheless, it provides a useful foundation for understanding how symbol probabilities influence efficient code design.

5.1. Algorithm Overview [2]

Fano Code

Step 1: Split all proabiltes into Two Groups has nearly equal prabilities.
Step 2: Assign the first one as 0 and the second one as 1.
Step 3: In Each Group Repeat step 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 on symbole.

Figure 13 Fano Algorithm

As the Steps above shows, the hand analysis of the Fano Shannon code is the following

5.2. Hand Analysis

Symbol	P0	C0	P1	C1	P2	C2	P3	C3	P4	C4	Final Code
A	0.35	0	0.35	00	0.35	00	0.35	00	0.35	00	00
B	0.3		0.3	01	0.3	01	0.3	01	0.3	01	01
C	0.2	1	0.2	10	0.2	10	0.2	10	0.2	10	10
D	0.1		0.1	11	0.1	110	0.1	110	0.1	110	110
E	0.04		0.04		0.04	111	0.04	1110	0.04	1110	1110
F	0.005		0.005		0.005		0.005	1111	0.005	11110	11110
G	0.005		0.005		0.005		0.005		0.005	11111	11111

Kraft's tree:

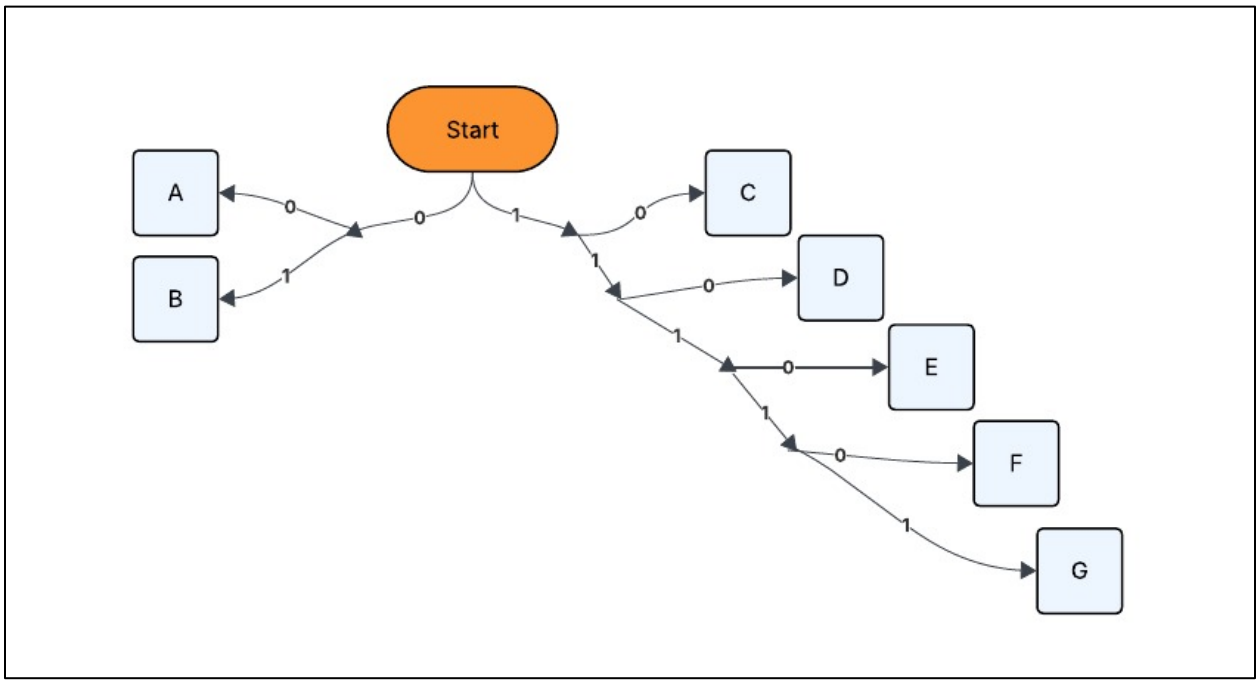

Figure 14 Fano Output Kraft's Tree

So the Output we are aiming for with the Following Entropy, Codeword Length, and Efficiency

Entropy of the code

$$H(P(x)) = 2.11\ bits/symbol$$

Average codeword length

$$L(C) = 2.21\ bits/symbol$$

So the overall efficiency is $\eta = \frac{L(C)}{H(P(x))} * 100\% = \frac{2.11}{2.21} * 100\% = 95.475\%$

And **Kraft's Sum = 1**

5.2.1. Fano Function[3]

First we splited the probabilities into groups
Then we assign the codes to each group

```
% === STEP 2: ITERATIVE FANO GROUPING ===
while ~isempty(groups)
    new_groups = {};
    for g = 1:length(groups)
        cur = groups{g};
        idxs = cur.idx;
        prefix = cur.prefix;

        % Skip if one symbol only
        if numel(idxs) <= 1
            continue;
        end

        pvals = probs(idxs);
        % --- Compute reference dynamically ---
        ref = 2^(-iteration);

        % --- Decide split method (COMBINATION-BASED) ---
        best_diff = inf;
        split_idx = [];
        n = length(pvals);

        for mask = 1:(2^n - 1)
            subset = bitget(mask, 1:n);
            subset_sum = sum(pvals(logical(subset)));
            diff = abs(subset_sum - ref);
            if diff < best_diff
                best_diff = diff;
                split_idx = find(subset);
            end
        end
```

Figure 15 Group Splitting

```
% --- Create two new groups ---
g1 = idxs(split_idx);
g2 = setdiff(idxs, idxs(split_idx));

% Assign '0' and '1' respectively
for k = g1
    codes{k} = [prefix '0'];
end
for k = g2
    codes{k} = [prefix '1'];
end
```

Figure 16 Group Code Assignment

So the output steps is

	Symbol	P0	C0	P1	C1	P2	C2	P3	C3	P4	C4	P5	C5
1	A	0.3500		0.3500	0	0.3500	00	0.3500	00	0.3500	00	0.3500	00
2	B	0.3000		0.3000	0	0.3000	01	0.3000	01	0.3000	01	0.3000	01
3	C	0.2000		0.2000	1	0.2000	10	0.2000	10	0.2000	10	0.2000	10
4	D	0.1000		0.1000	1	0.1000	11	0.1000	110	0.1000	110	0.1000	110
5	E	0.0400		0.0400	1	0.0400	11	0.0400	111	0.0400	1110	0.0400	1110
6	F	0.0050		0.0050	1	0.0050	11	0.0050	111	0.0050	1111	0.0050	11110
7	G	0.0050		0.0050	1	0.0050	11	0.0050	111	0.0050	1111	0.0050	11111

Figure 17 Fano Steps

5.2.1. Theoretical Vs Practical

The results are shown are the same as the theoretical

Fano

	Symbol	Probability	Code
1	A	0.3500	00
2	B	0.3000	01
3	C	0.2000	10
4	D	0.1000	110
5	E	0.0400	1110
6	F	0.0050	11110
7	G	0.0050	11111

H = 2.1100 | L = 2.2100 | η = 95.47 % | Kraft = 1.0000

Figure 18 Fano Output

6. Comparison and conclusion

For this particular source, **Huffman and Shannon–Fano coding produce identical results** — the same set of codeword lengths, average length (**2.21 bits**), and efficiency (≈ **95.5 %**)
However, in general:

- **Huffman coding** always yields the *optimal* prefix-free code for a given probability distribution.
- **Fano coding**, while simpler and faster to implement, may occasionally produce a slightly larger average code length.

Thus, **Huffman coding is preferred** when optimal compression efficiency is required, while **Fano coding** is mainly of educational or conceptual value for illustrating the principles of probabilistic source coding.

And also, Huffman coding generally performs better for larger symbol sets or skewed probability distributions, where precision in tree construction becomes more significant

7. References:

[1] Dr. Mohammed Nafie, "Lecture Slides in Advanced Communications," ***ELC4020 Advanced Communication Systems*****, Cairo University, 2025.**

[2] Eng. Mohamed Khaled, "Section Slides in Advanced Communications," ***ELC4020 Advanced Communication Systems*****, Cairo University, 2025.**

[3] https://github.com/youefkh05/Advanced_Communication_Coding

8. Appendix

```
%%-------------------------------------------------------------
% Problem 1: Binary Huffman Coding
%-------------------------------------------------------------

clc; clear; close all force;

% Given Symbols probabilities
symbols = {'A','B','C','D','E','F','G'};
P = [0.35 0.30 0.20 0.10 0.04 0.005 0.005];

% Create Input Dictionary
[dict_input,err_flag, H] = create_symbols_dictionary(symbols, P);

% Check Input
if err_flag ==1
    disp('? Stopping execution due to invalid dictionary.');
    return; % exits the current script or function
end

% Print the dictionary neatly
```

```
print_symbols_dic(dict_input, H);


% ------------------------------------------------------------------------
% Manual Huffman Coding (with custom output)
% ------------------------------------------------------------------------
dict_huffman = huffman_encoding_visual(dict_input);

disp('--- Manual Huffman Encoding ---');
disp(dict_huffman);

% Print the codded dictionary neatly
print_coded_dict(dict_huffman, H,"Huffman");


%%-------------------------------------------------------------
% Problem 2: Binary Fano  Coding
%-------------------------------------------------------------

% ------------------------------------------------------------------------
% Manual Fano Coding (with custom output)
% ------------------------------------------------------------------------
dict_Fano = fano_encoding_visual(dict_input)

disp('--- Manual Fano Encoding ---');
disp(dict_Fano);

% Print the codded dictionary neatly
print_coded_dict(dict_Fano, H,"Fano");



%%
% ------------------------------------------------------------------------
%              Function Definition
% ------------------------------------------------------------------------

%% ------------------------------------------------------------------------
%              Entropy Calculation
% ------------------------------------------------------------------------

function H = entropy_calc(P)
%ENTROPY_CALC Compute the source entropy H(P(x))
%   H = entropy_calc(P)
%   P : vector of symbol probabilities
%   H : entropy in bits

    % Validate input
    if any(P < 0) || abs(sum(P) - 1) > 1e-6
        warning('Probabilities should sum to 1. Normalizing...');
        P = P / sum(P);
    end

    % Remove zeros (to avoid log2(0))
    P = P(P > 0);

    % Compute entropy
```

```
    H = -sum(P .* log2(P));
end

%% ---------------------------------------------------------------------
%              Average Length Calculation
% ---------------------------------------------------------------------
function L = average_length_calc(dict)
%AVERAGE_LENGTH_CALC Compute average codeword length L(C)
%   L = average_length_calc(dict, P)
%   dict : Huffman dictionary cell array {symbol, code}
%   P : vector of symbol probabilities (same order as dict)
%
%   If P is empty, it tries to extract from dict(:,2) if present
%   L : average code length

    P = dict(:,2);
    % --- Handle inputs ---
    if nargin < 2 || isempty(P)
        % Check if dict has a probability column (3 columns)
        if size(dict, 2) >= 3 && isnumeric(dict{1,2})
            P = cell2mat(dict(:,2));
            codes = dict(:,3);
        else
            error('Probability vector P is required or must be in dict(:,2)');
        end
    else
        % Extract codes (assumed in 2nd column)
        codes = dict(:,3);
    end

    % --- Compute code lengths ---
    code_lengths = cellfun(@length, codes);

    % --- Normalize probabilities ---
    P = P(:) / sum(P);

    % --- Check dimensions ---
    if length(P) ~= length(code_lengths)
        error('Number of probabilities does not match number of codewords.');
    end

    % --- Compute average code length ---
    L = sum(P .* code_lengths);
end

%% ---------------------------------------------------------------------
%              Efficiency Calculation
% ---------------------------------------------------------------------

function eta = efficiency_calc(H, L)
%EFFICIENCY_CALC Compute Huffman coding efficiency ?
%   eta = efficiency_calc(H, L)
%   H : entropy
%   L : average code length
%   eta : efficiency in percentage (%)

    if L <= 0
```

```
        error('Average length L must be positive.');
    end

    eta = (H / L) * 100;
end


%% ---------------------------------------------------------------------
%               Print Kraft Inequality Function
% ---------------------------------------------------------------------

function [kraft_sum, kraft_flag]=kraft_analysis(dict)
% KRAFT_ANALYSIS  Compute Kraft's inequality and visualize Kraft tree
%   kraft_analysis(dict)
%
%   Input:
%       dict : cell array {N x 3}  ? {symbol, P, code}
%
%   Example:
%       dict = {'A','0'; 'B','10'; 'C','110'; 'D','111'};
%       kraft_analysis(dict);

    if ~iscell(dict) || size(dict,2) < 2
        error('Input must be a cell array {symbol, code}');
    end

    % Extract codes
    codes = dict(:,3);
    N = length(codes);

    % --- 1. Compute Kraft's inequality ---
    code_lengths = cellfun(@length, codes);
    kraft_sum = sum(2.^(-code_lengths));

    fprintf('\n=== Kraft Inequality Check ===\n');
    fprintf('Sum(2^{-l_i}) = %.4f\n', kraft_sum);



    if abs(kraft_sum - 1) < 1e-6
        fprintf('? Code satisfies equality ? Complete Prefix Code.\n');
        kraft_flag=2;
    elseif kraft_sum < 1
        fprintf('? Code satisfies inequality (valid but not complete).\n');
        kraft_flag=1;
    else
        fprintf('? Invalid prefix code (violates Kraft''s inequality).\n');
        kraft_flag=0;
    end

end


%% ---------------------------------------------------------------------
%              Create Dictionary Input Definition
% ---------------------------------------------------------------------
function [dict_input,err_flag, H] = create_symbols_dictionary(symbols, P)
```

```
%CREATE_DICTIONARY  Combines symbols and probabilities into a validated dictionary.
%
%   dict_input = create_dictionary(symbols, P)
%
%   Inputs:
%       symbols - cell array of symbols, e.g. {'A','B','C'}
%       P       - corresponding probabilities (row or column vector)
%
%   Output:
%       dict_input - cell array {symbol, probability}
%
%   Example:
%       symbols = {'A','B','C'};
%       P = [0.5 0.3 0.2];
%       dict_input = create_dictionary(symbols, P);

    % Combine into dictionary-like cell array
    dict_input = [symbols(:), num2cell(P(:))];

    % Assume not great until great
    err_flag = 1;

    % Validate using the check_symbols() function
    [ok, msg] = check_symbols(dict_input);

    % Display validation result
    if ok
        disp('? Dictionary is valid!');
        err_flag =0;
    else
        disp(['? Error: ' msg]);
        err_flag = 1;
    end

    % ---------------------------------------------------------------------
    % Compute entropy (only if valid)
    % ---------------------------------------------------------------------
    H = entropy_calc(P)

end


%% ---------------------------------------------------------------------
%               Check Input Validation Function
% ---------------------------------------------------------------------
function [isValid, errMsg] = check_symbols(dict_input)
% CHECK_SYMBOLS  Validates a symbol-probability dictionary
%
%   [isValid, errMsg] = check_symbols(dict_input)
%
%   Input:
%       dict_input : Cell array {N×2}, where first column = symbols,
%                    second column = probabilities
%
%   Output:
%       isValid : Logical true if valid, false otherwise
%       errMsg  : String describing validation error (if any)
```

```
    % Default output
    isValid = false;
    errMsg  = '';

    try
        % Extract symbols and probabilities
        symbols = dict_input(:, 1);
        P = cell2mat(dict_input(:, 2));

        % Check same length
        if numel(symbols) ~= numel(P)
            errMsg = 'Symbols and probabilities must have the same length.';
            return;
        end

        % Check probabilities sum to 1 (within tolerance)
        if abs(sum(P) - 1) > 1e-6
            errMsg = sprintf('Probabilities do not sum to 1 (sum = %.6f).', sum(P));
            return;
        end

        % Check all probabilities are positive
        if any(P <= 0)
            errMsg = 'All probabilities must be positive.';
            return;
        end

        % If all checks passed
        isValid = true;

    catch ME
        errMsg = ['Invalid dictionary input: ' ME.message];
    end
end


%% ------------------------------------------------------------------------
%               Print Dictionary Function
% -------------------------------------------------------------------------
function print_symbols_dic(dict_input, H)
% PRINT_SYMBOLS_DIC  Displays a formatted version of the symbol dictionary in a figure,
%                    and shows the calculated source entropy.
%
%   print_symbols_dic(dict_input, H)
%
%   Inputs:
%       dict_input - cell array {symbol, probability}
%       H          - source entropy (bits/symbol)

    % Validate input
    if nargin < 1 || isempty(dict_input)
        error('Input dictionary is empty or missing.');
        return;
    end

    % Convert symbols to char (uitable can't handle string objects)
```

```
    symbols = cellfun(@char, dict_input(:,1), 'UniformOutput', false);
    probs = cell2mat(dict_input(:,2));

    % Display result in Command Window
    fprintf('\nInformation Source Entropy: H = %.4f bits/symbol\n', H);
    fprintf('-----------------------------------------------------\n');

    % Create a responsive UI figure
    f = uifigure('Name', 'Symbol Dictionary', ...
                 'NumberTitle', 'off', ...
                 'Color', 'w', ...
                 'Position', [500 400 350 320]);

    % Format probabilities as strings
    probStr = arrayfun(@(p) sprintf('%.4f', p), probs, 'UniformOutput', false);

    % Combine into table data
    data = [symbols probStr];

    % Create a grid layout (auto-resizes)
    gl = uigridlayout(f, [3,1]);
    gl.RowHeight = {'fit', '1x', 'fit'};  % title, table, entropy
    gl.ColumnWidth = {'1x'};
    gl.Padding = [10 10 10 10];

    % --- Title ---
    uilabel(gl, ...
        'Text', '--- Input Symbol Dictionary ---', ...
        'FontSize', 14, ...
        'FontWeight', 'bold', ...
        'HorizontalAlignment', 'center');

    % --- Table ---
    uitable(gl, ...
        'Data', data, ...
        'ColumnName', {'Symbol', 'Probability'}, ...
        'FontSize', 12, ...
        'ColumnWidth', {'1x', '1x'}, ...
        'RowStriping', 'on');

    % --- Entropy Display ---
    uilabel(gl, ...
        'Text', sprintf('Entropy: H = %.4f bits/symbol', H), ...
        'FontSize', 12, ...
        'FontWeight', 'bold', ...
        'FontColor', [0 0.3 0.7], ...
        'HorizontalAlignment', 'center');
end

%% ------------------------------------------------------------------------
%               Print Coded Dictionary Function (General Purpose)
% ------------------------------------------------------------------------
function print_coded_dict(dict, H, title_str)
% PRINT_CODED_DICT  Display coding dictionary with entropy, avg length, efficiency, and
    Kraft analysis.
%
%   print_coded_dict(dict, H, title_str)
```

```
%
%   Inputs:
%       dict       - cell array {symbol, probability, code}
%       H          - entropy (bits/symbol)
%       title_str  - string title for the table window (e.g. 'Fano Coding Results')
%
%   This function:
%       • Calculates average codeword length L(C)
%       • Calculates efficiency ? = (H / L) * 100%
%       • Checks Kraft's inequality
%       • Displays results in MATLAB UI + command window
%
%   Example:
%       print_coded_dict(fano_dict, H, 'Fano Coding Summary');

    % === Validate input ===
    if nargin < 1 || isempty(dict)
        disp('Input dictionary is missing or empty.');
        return;
    end

    if size(dict,2) < 3
        disp('Dictionary must have 3 columns: {symbol, probability, code}.');
        return;
    end

    if nargin < 3 || isempty(title_str)
        title_str = 'Coded Dictionary Summary';
    end

    % === Extract data ===
    symbols = cellfun(@char, dict(:,1), 'UniformOutput', false);
    P       = cell2mat(dict(:,2));
    codes   = dict(:,3);

    % === Compute metrics ===
    L   = average_length_calc(dict);
    eta = efficiency_calc(H, L);
    [kraft_sum, kraft_flag] = kraft_analysis(dict);

    % === Print to Command Window ===
    fprintf('\n--- %s ---\n', title_str);
    fprintf('Symbol\tProb.\t\tCode\n');
    fprintf('----------------------------------------\n');
    for i = 1:length(symbols)
        fprintf('%s\t%.4f\t\t%s\n', symbols{i}, P(i), codes{i});
    end
    fprintf('----------------------------------------\n');
    fprintf('Entropy (H):           %.4f bits/symbol\n', H);
    fprintf('Average length (L):    %.4f bits/symbol\n', L);
    fprintf('Efficiency (?):        %.2f %%\n', eta);
    fprintf('Kraft Sum:             %.4f\n', kraft_sum);
    if kraft_flag == 2
        fprintf('Kraft Result: ? Complete Prefix Code\n');
    elseif kraft_flag == 1
        fprintf('Kraft Result: ? Valid but Not Complete\n');
    else
```

```
        fprintf('Kraft Result: ? Invalid Code\n');
    end

    % === UI Figure ===
    f = uifigure('Name', title_str, ...
                 'NumberTitle', 'off', ...
                 'Color', 'w', ...
                 'Position', [500 200 480 450]);

    gl = uigridlayout(f, [3 1]);
    gl.RowHeight = {'fit', '1x', 'fit'};
    gl.Padding = [10 10 10 10];

    % --- Title ---
    uilabel(gl, ...
        'Text', ['--- ' title_str ' ---'], ...
        'FontSize', 14, ...
        'FontWeight', 'bold', ...
        'HorizontalAlignment', 'center');

    % --- Table ---
    data = [symbols, arrayfun(@(p) sprintf('%.4f', p), P, 'UniformOutput', false),
      codes];
    uitable(gl, ...
        'Data', data, ...
        'ColumnName', {'Symbol', 'Probability', 'Code'}, ...
        'FontSize', 12, ...
        'RowStriping', 'on', ...
        'ColumnWidth', {'1x', '1x', '1x'});

    % --- Summary Line ---
    uilabel(gl, ...
        'Text', sprintf('H = %.4f | L = %.4f | ? = %.2f %% | Kraft = %.4f', H, L, eta,
      kraft_sum), ...
        'FontSize', 12, ...
        'FontWeight', 'bold', ...
        'FontColor', [0 0.3 0.7], ...
        'HorizontalAlignment', 'center');
end


%% ------------------------------------------------------------------------
%               Huffman Encoding with Visualization Function
% ------------------------------------------------------------------------

function dict = huffman_encoding_visual(dict_input)
%HUFFMAN_ENCODING_VISUAL Visual Huffman encoding with full table output (UI-based)
%
%   dict = huffman_encoding_visual(symbols, P)
%   - symbols: cell array of symbol names (e.g. {'A','B','C','D','E','F','G'})
%   - P: vector of probabilities (same length as symbols)
%
%   Creates a UI figure showing the probability & code propagation table,
%   and prints the final Huffman dictionary.

    % get the info from dictionary
    symbols = dict_input(:,1);
```

```
P = cell2mat(dict_input(:,2));
% === Input Validation ===
if numel(symbols) ~= numel(P)
    error('Symbols and probabilities must have same length.');
end

% === Normalize probabilities ===
P = P(:);
P = P / sum(P);

% === Step 1: Generate merging history ===
history_table = merge_probabilities(P);

 % === Step 1 Visualization ===
visualize_merging_process(P, history_table);

% === Step 2: Assign Huffman codes ===
history_table_full = assign_coding(history_table);

% === Step 3: Prepare data for visualization ===
% Convert numeric NaNs to empty strings for table display
final_visual_data = cell(size(history_table_full));
for r = 1:size(history_table_full,1)
    for c = 1:size(history_table_full,2)
        val = history_table_full{r,c};
        if isnumeric(val)
            if isnan(val)
                final_visual_data{r,c} = '';
            else
                final_visual_data{r,c} = num2str(val, '%.4f');
            end
        else
            final_visual_data{r,c} = val;
        end
    end
end

% Generate column headers (P1, C1, P2, C2, ...)
numCols = size(history_table_full,2);
final_visual_headers = cell(1,numCols);
for c = 1:numCols
    if mod(c,2)==1
        final_visual_headers{c} = sprintf('P%d', ceil(c/2)-1);
    else
        final_visual_headers{c} = sprintf('C%d', ceil(c/2)-1);
    end
end

% === Step 4: Build UI Visualization ===
close all;
f = uifigure('Name','Huffman Encoding Visualization', ...
             'Position',[100 100 1000 500]);
gl = uigridlayout(f,[2 1]);
gl.RowHeight = {'fit','1x'};

uilabel(gl, ...
    'Text','Huffman Encoding: Probability and Code Evolution (P/C Steps)', ...
```

```
        'FontSize',16, ...
        'FontWeight','bold', ...
        'HorizontalAlignment','center');

    % Column widths (narrow for numeric columns, wider for code columns)
    col_widths = repmat({70}, 1, numCols);
    col_widths(2:2:end) = {100}; % widen code columns

    uitable(gl, ...
        'Data',final_visual_data, ...
        'ColumnName',final_visual_headers, ...
        'RowName',{}, ...
        'FontSize',12, ...
        'ColumnWidth',col_widths, ...
        'RowStriping','on', ...
        'BackgroundColor',[1 1 1; 0.95 0.95 1]);

    % === Step 5: Extract Final Huffman Dictionary ===

    % Make a copy
    dict = dict_input;

    % Ensure dict has at least 3 columns
    if size(dict,2) < 3
        dict(:,end+1:3) = {[]};
    end
    dict(:,3)=history_table_full(:,2);

    % === Step 6: Console Output ===
    firstPcol = 1;
    firstCcol = 2;

    probs = cell2mat(history_table_full(:, firstPcol));
    codes = history_table_full(:, firstCcol);
    validIdx = ~isnan(probs);

    symbols = symbols(validIdx);
    codes = codes(validIdx);
    probs = probs(validIdx);

    fprintf('\n--- Final Huffman Codes ---\n');
    for i = 1:length(symbols)
        fprintf('Symbol %s (%.4f): %s\n', symbols{i}, probs(i), codes{i});
    end
    fprintf('=================================================================\n\n');
end

% === Probability Merge helper function  ===
function history_table = merge_probabilities(P)
%MERGE_PROBABILITIES Builds Huffman probability merging history (descending)
%
%   history_table = merge_probabilities(P)
%
%   Input:
%       P - vector of symbol probabilities
%
%   Output:
```

```
%       history_table - table of probabilities after each merge
%                       Columns: P0, P1, P2, ... (N-1 total)
%
%   Note: Probabilities are shown in descending order.

    % --- Input check ---
    if numel(P) < 2
        error('At least two probabilities are required.');
    end

    % --- Initialization ---
    P = P(:);
    P = sort(P, 'descend'); % sort descending
    N = numel(P);

    % Number of P columns = N - 1
    numCols = N - 1;
    maxRows = N;

    % Initialize history as cell
    history = cell(maxRows, numCols);

    % --- Step 0: Fill P0 (descending order) ---
    for i = 1:maxRows
        history{i,1} = P(i);
    end

    curP = P;

    % --- Iteratively merge ---
    for step = 2:numCols
        % Sort ascending to pick smallest two
        curP = sort(curP, 'ascend');
        if numel(curP) >= 2
            p1 = curP(1);
            p2 = curP(2);
            mergedP = p1 + p2;
            % Remove two smallest and add merged one
            curP = [mergedP; curP(3:end)];
        end
        % Sort descending for display
        curP = sort(curP, 'descend');

        % Fill current column
        for r = 1:maxRows
            if r <= numel(curP)
                history{r,step} = curP(r);
            else
                history{r,step} = NaN;
            end
        end
    end

    % --- Column names ---
    colNames = cell(1,numCols);
    for i = 1:numCols
        colNames{i} = sprintf('P%d', i-1);
```

```
    end

    % --- Convert to table ---
    history_table = cell2table(history, 'VariableNames', colNames);
end


% === Visualization of Probability Merging Process  ===
function visualize_merging_process(P, history_table)
% VISUALIZE_MERGING_PROCESS Display a UI table of Huffman probability merging
%
%   visualize_merging_process(P, history_table)
%
%   Displays the merging steps used in Huffman encoding as a clean table UI.
%
%   Inputs:
%       P              - Original probability vector (used for scaling)
%       history_table  - Table of merging probabilities (output of merge_probabilities)
%
%   Example:
%       history_table = merge_probabilities([0.4 0.2 0.15 0.15 0.1]);
%       visualize_merging_process([0.4 0.2 0.15 0.15 0.1], history_table);

    % --- Input validation ---
    if nargin < 2
        error('Usage: visualize_merging_process(P, history_table)');
    end
    if ~istable(history_table)
        error('history_table must be a MATLAB table.');
    end

    % === Step 1: Convert numeric values (NaN ? empty string for display) ===
    merge_visual_data = cell(size(history_table));
    for r = 1:size(history_table,1)
        for c = 1:size(history_table,2)
            val = history_table{r,c};
            if isnumeric(val)
                if isnan(val)
                    merge_visual_data{r,c} = '';
                else
                    merge_visual_data{r,c} = num2str(val, '%.4f');
                end
            else
                merge_visual_data{r,c} = val;
            end
        end
    end

    % === Step 2: Build UI Table Figure ===
    f_merge = uifigure('Name','Step 1: Probability Merging History', ...
                 'Position',[150 150 700 400]);

    gl_merge = uigridlayout(f_merge,[2 1]);
    gl_merge.RowHeight = {'fit','1x'};

    uilabel(gl_merge, ...
        'Text','Step 1: Huffman Probability Merging Process', ...
```

```
        'FontSize',16, ...
        'FontWeight','bold', ...
        'HorizontalAlignment','center');

    uitable(gl_merge, ...
        'Data',merge_visual_data, ...
        'ColumnName',history_table.Properties.VariableNames, ...
        'RowName',{}, ...
        'FontSize',12, ...
        'ColumnWidth',repmat({80}, 1, width(history_table)), ...
        'RowStriping','on', ...
        'BackgroundColor',[1 1 1; 0.95 0.95 1]);

    % === Step 3: Optional console summary ===
    fprintf('\n--- Probability Merging Steps ---\n');
    disp(history_table);
    fprintf('================================\n\n');
end


% === Assign code helper function  ===
function history_table_full = assign_coding(history_table)
    % assign_coding - expands the history table and assigns Huffman codes
    %
    % Input:
    %   history_table : numeric matrix or table (probability merging history)
    %
    % Output:
    %   history_table_full : cell array with 2N columns
    %       Odd columns: probability values
    %       Even columns: assigned codes

    % If table, convert to numeric array
    if istable(history_table)
        history_table = table2array(history_table);
    end

    % Determine size
    [numRows, numCols] = size(history_table);
    newCols = 2 * numCols;

    % Initialize
    history_table_full = cell(numRows, newCols);

    % === Fill odd columns with probabilities ===
    for col = 1:numCols
        history_table_full(:, 2 * col - 1) = num2cell(history_table(:, col));
    end

    % === Initialize last code column (start with last merge) ===
    lastPcol = 2 * numCols - 1;
    lastCcol = lastPcol + 1;
    history_table_full{1, lastCcol} = '0';
    history_table_full{2, lastCcol} = '1';
    raw_counter=1;  %for parent assignment

    % === Backward propagation of codes ===
```

```
for col = numCols:-1:2  % start from last column going backward
    currPcol = 2 * col - 1;
    currCcol = currPcol + 1;
    prevPcol = 2 * (col - 1) - 1;
    prevCcol = prevPcol + 1;
    raw_counter = raw_counter+1;

    % Get non-NaN values from P prev column
    prevPvals = cell2mat(history_table_full(:, prevPcol));
    prevPvals = prevPvals(~isnan(prevPvals));

    % Get non-NaN values from C curr column
    currCvals  = history_table_full(:, currCcol);


    % Identify merged value
    if length(prevPvals) >= 2
        mergedVal = prevPvals(end) + prevPvals(end-1);
    else
        continue;
    end

    % Find which row in current P col matches the mergedVal
    currPvals = cell2mat(history_table_full(:, currPcol));
    matchIdx = find(abs(currPvals - mergedVal) < 1e-12);
    if numel(matchIdx) > 1
        matchIdx = matchIdx(1); % take top one if duplicate
    end

    % Get parent code
    parentCode = history_table_full{matchIdx, currCcol};
    if isempty(parentCode)
        parentCode = '';
    end

    % === Assign child codes ===
    % Last two rows in previous P column are merged into this parent
    history_table_full{raw_counter, prevCcol}   = [parentCode '0'];
    history_table_full{raw_counter+1, prevCcol} = [parentCode '1'];

    % For each previous non-merged row (in display order top->bottom)
    for ii = 1:(raw_counter-1)
        % Skip the rows that were just merged (raw_counter and raw_counter+1)

        % Get the probability value in the previous column for this row
        valPrev = history_table_full{ii, prevPcol};
        if isnan(valPrev)
            continue; % nothing to copy
        end

        % Find matching value in the current column (exclude merged parent)
        currMatches = find(abs(currPvals - valPrev) < 1e-12);

        % Remove the matchIdx (the merged parent) if it appears
        currMatches(currMatches == matchIdx) = [];

        if isempty(currMatches)
```

```
                continue; % no corresponding match found
            end

            % If there are duplicates (two identical probabilities)
            if numel(currMatches) > 1
                % take both, and copy their codes to the two rows
                history_table_full{ii,   prevCcol} = currCvals{currMatches(1)};
                if (ii+1) <= numRows
                    history_table_full{ii+1, prevCcol} = currCvals{currMatches(2)};
                end
            else
                % single match — copy code directly
                history_table_full{ii, prevCcol} = currCvals{currMatches(1)};
            end
        end

    end
end
%% ------------------------------------------------------------------------
%               Fano Encoding with Visualization Function
% ------------------------------------------------------------------------

function table_out = fano_encoding_visual(dict_input)
% FANO_ENCODING_VISUAL
% Classic Fano encoding: split groups into two with nearly equal probabilities
%
% Output table columns:
% Symbol | P0 | C0 | P1 | C1 | ... until convergence

    % === STEP 1: INITIALIZE ===
    symbols = dict_input(:,1);
    probs = cell2mat(dict_input(:,2));

    % Normalize and sort descending
    probs = probs / sum(probs);
    [probs, idx] = sort(probs, 'descend');
    symbols = symbols(idx);

    n = length(symbols);

    % Initialize codes and history
    codes = repmat({''}, n, 1);
    history = cell(n, 0);

    % Add first columns P0 and C0
    history(:, end+1) = num2cell(probs); % P0
    history(:, end+1) = codes;           % C0

    % Start with one full group
    groups = {struct('indices', 1:n, 'prefix', '')};

    % === STEP 2: ITERATIVE FANO GROUPING ===
    while ~isempty(groups)
        new_groups = {};
```

```
    for g = 1:length(groups)
        cur_group = groups{g};
        group_indices = cur_group.indices;
        prefix = cur_group.prefix;

        % Skip if one symbol only
        if numel(group_indices) <= 1
            continue;
        end

        group_probs = probs(group_indices);

        % --- Split into two groups with nearly equal probabilities ---
        split_point = find_equal_split(group_probs);

        % Group 1: indices 1 to split_point
        g1_indices = group_indices(1:split_point);
        % Group 2: indices split_point+1 to end
        g2_indices = group_indices(split_point+1:end);

        % Assign '0' to first group, '1' to second group
        for k = g1_indices
            codes{k} = [prefix '0'];
        end
        for k = g2_indices
            codes{k} = [prefix '1'];
        end

        % Queue for next stage
        if ~isempty(g1_indices)
            new_groups{end+1} = struct('indices', g1_indices, 'prefix', [prefix '0']);
        end
        if ~isempty(g2_indices)
            new_groups{end+1} = struct('indices', g2_indices, 'prefix', [prefix '1']);
        end
    end

    % Stop if all groups are singletons
    if isempty(new_groups)
        break;
    end

    % Record stage snapshot
    history(:, end+1) = num2cell(probs); % Pi
    history(:, end+1) = codes;           % Ci

    groups = new_groups;
end

% === STEP 3: COMPILE OUTPUT TABLE ===
cols = {'Symbol'};
for i = 0:(size(history,2)/2 - 1)
    cols{end+1} = sprintf('P%d', i);
    cols{end+1} = sprintf('C%d', i);
end
```

```
    table_out = [symbols history];

    % Display as uitable
    figure('Name','Fano Encoding Table','Position',[300 200 1100 400]);
    uitable('Data', table_out, 'ColumnName', cols, ...
        'ColumnWidth', num2cell(repmat(80,1,numel(cols))), ...
        'FontSize', 11, 'Units','normalized', 'Position',[0 0 1 1]);

    % === STEP 4: FINAL OUTPUT (Symbol, Prob, Code) ===
    table_out = [symbols num2cell(probs) codes];
end

function split_point = find_equal_split(group_probs)
% FIND_EQUAL_SPLIT Find split point that makes two groups with nearly equal probabilities
%
% Strategy: Find the split point where |sum(top_group) - sum(bottom_group)| is minimized
%
% Input: group_probs (already sorted descending)
% Output: split_point (index where to split)

    n = length(group_probs);

    if n == 1
        split_point = 1;
        return;
    end

    total_sum = sum(group_probs);
    best_diff = inf;
    best_split = 1;

    % Try each possible split point
    for i = 1:(n-1)
        top_sum = sum(group_probs(1:i));
        bottom_sum = sum(group_probs(i+1:end));
        diff = abs(top_sum - bottom_sum);

        if diff < best_diff
            best_diff = diff;
            best_split = i;
        end
    end

    split_point = best_split;
end
```