```
clc
clear

% addpath("EE4740_Miniproject_1/")
sound = audioread("SA1.wav");
```

Question 1

Preliminary research, this part aims to study the selection of L suitable for Huffman code

```
num = 40;
t = zeros(1,num);

h=waitbar(0,'please wait');

for i = 1:num
    for j = 1:3
        L = i*5;
        tStart = tic;
        [quantized_sound, level, encoded_audio_string, codeMap, efficiency]
= HuffmanCoding(sound, L);
        decoded_audio = HuffmanDecoding(encoded_audio_string, codeMap);

        tolerance = 1e-5;
        Check(quantized_sound, decoded_audio, tolerance);
        time_buffer(j) = toc(tStart);
        eff_buffer(j) = efficiency;
        waitbar(i/num, h)
    end
    t(i) = mean(time_buffer);
    eff(i) = mean(eff_buffer);
    entries(i) = size(codeMap,1);
end
delete(h);

subplot(1,2,1)
plot(entries, t)
xlabel('Number of Huffman Code Entries')
ylabel('Time consuming')
title('Relation of Huffman code entries and consumed time')

subplot(1,2,2)
plot(entries, eff)
xlabel('Number of Huffman Code Entries')
ylabel('Efficiency')
title('Relation of Huffman code entries and Efficiency')
```

Question 2

2.1

This section aims to show the differences between different audio data

```
L = 50;

audio_files = {'person1.wav', 'person2.wav', 'person3.wav', 'person4.wav',
'person5.wav', 'person6.wav', 'person7.wav', 'person8.wav'};

quant_freqs = zeros(L, length(audio_files));

for i = 1:length(audio_files)
    [sound, ~] = audioread(audio_files{i});

    max_ = max(sound);
    min_ = min(sound);
    step = (max_ - min_) / (L-1);
    level = min_ : step : max_;
    differences = abs(sound - reshape(level, [1, 1, length(level)]));
    [~, indices] = min(differences, [], 3);
    quantized_sound = level(indices)';

    for j = 1:L
        quant_freqs(j, i) = sum(quantized_sound == level(j)) /
length(quantized_sound);
    end
end

figure
bar(quant_freqs, 'grouped')
title('Quantization Level Frequencies for Different People')
xlabel('Quantization Level')
ylabel('Frequency')
legend('Person 1', 'Person 2', 'Person 3', 'Person 4', 'Person 5', 'Person
6', 'Person 7', 'Person 8', 'Location', 'bestoutside')
```

2.2

This part is the required Huffman algorithm implementation. Note that robustness will be considered separately in Question 3, reason shown the same place.

```
clc
clear

training_files = {'person2.wav', 'person3.wav', 'person4.wav', ...
                  'person6.wav', 'person7.wav', 'person8.wav', ...
                  'person12.wav', 'person22.wav', 'person32.wav', ...
                  'person42.wav', 'person52.wav', 'person62.wav', ...
                  'person72.wav', 'person82.wav'};
```

```
test_files = {'person1.wav', 'person5.wav'};

training_data = [];
for i = 1:length(training_files)
    [data, Fs] = audioread(training_files{i});
    if i == 1
        training_Fs = Fs;
    else
        if Fs ~= training_Fs
            error('Sampling rate mismatch in training data.');
        end
    end
    training_data = [training_data; data];
end

test_data = [];
for i = 1:length(test_files)
    [data, Fs] = audioread(test_files{i});
    if i == 1
        test_Fs = Fs;
    else
        if Fs ~= test_Fs
            error('Sampling rate mismatch in test data.');
        end
    end
    test_data = [test_data; data];
end

L = 50;
[~, level, encoded_audio_string, codeMap, Efficiency] =
HuffmanCoding(training_data, L);
quantized_test_data = zeros(size(test_data));

%Test
test_data(test_data == 0) = []; % Remove the silent part, if any
quantized_test_data = quantizeSound(test_data, level);

encoded_test_string = HuffmanEncode(quantized_test_data, codeMap);
decoded_test_data = HuffmanDecoding(encoded_test_string, codeMap);

tolerance = 1e-5;
% Check if the two arrays are exactly equal
are_equal = isequal(quantized_test_data, decoded_test_data);

errors = abs(quantized_test_data - decoded_test_data);
are_close = all(errors < tolerance);

disp(['Arrays are equal: ', num2str(are_equal)]);
```

```
Arrays are equal: 0
```

```
disp(['Arrays are close within tolerance: ', num2str(are_close)]);
```

```
Arrays are close within tolerance: 1
```

Question 3

Due to personal computer memory limitations, it is impossible to encrypt whole audio data with Hamming code. Here I only use the part of the Huffman encoding string of audio data as a demonstration.

```
clc
clear
sound = audioread("SA1.wav");

L = 50;
[quantized_sound, level, encoded_audio_string, codeMap, Efficiency] =
HuffmanCoding(sound, L);

encoded_audio_string = encoded_audio_string(1:10000);
[padded_vector, num_added_zeros, n, k] =
padEncodedString(encoded_audio_string);

encData = encode(padded_vector,n,k,'hamming/binary');

%Add perturbation
errLoc = randerr(1,n);
encData = mod(encData + errLoc',2);

decData = decode(encData,n,k,'hamming/binary');

str_without_padding = removePadding(decData, num_added_zeros);

if length(str_without_padding) == length(encoded_audio_string) &&
all(str_without_padding == encoded_audio_string)
    disp('The string perturbed by the Hamming code has not changed.')
end
```

```
The string perturbed by the Hamming code has not changed.
```

Function definition

```
function [padded_vector, num_added_zeros, n, k] =
padEncodedString(encoded_test_string)
    % Check and convert the input string
    if isnumeric(encoded_test_string) || islogical(encoded_test_string)
        % Assume encoded_test_string is a binary array, convert it to a
character array
        encoded_test_string = num2str(encoded_test_string(:).');
        encoded_test_string = strrep(encoded_test_string, ' ', ''); %
Remove the spaces added by num2str
```

```
    end

    original_length = length(encoded_test_string);

    m = 2;
    while (2^m - m - 1) < original_length
        m = m + 1;
    end

    n = 2^m - 1;
    k = n - m;

    num_complete_info_bits = ceil(original_length / k) * k;

    num_added_zeros = num_complete_info_bits - original_length;

    padded_string = [encoded_test_string, repmat('0', 1, num_added_zeros)];

    %Adjust data format
    padded_vector = padded_string - '0';
    padded_vector = padded_vector';
end

function [data_str] = removePadding(decData, num_added_zeros)
    if num_added_zeros > 0
        data_without_padding = decData(1:end-num_added_zeros);
    else
        data_without_padding = decData;
    end

    data_str = num2str(data_without_padding');
    data_str = data_str(~isspace(data_str));
end

function [quantized_sound2, level, encoded_audio_string, codeMap,
Efficiency] = HuffmanCoding(sound, L)
    max_ = max(sound);
    min_ = min(sound);
    step = (max_ - min_) / (L-1);

    level = min_ : step : max_;

    % Map the sound value to the corresponding level
    differences = abs(sound - reshape(level, [1, 1, length(level)]));
    [~, indices] = min(differences, [], 3);
    quantized_sound2 = level(indices)';

    % Get the probabilities of different levels
    prob2 = zeros(size(level));
    for i = 1:length(level)
```

```
        prob2(i) = sum(quantized_sound2 == level(i));
    end
    prob2 = prob2 / length(quantized_sound2);

    % Remove those value with 0 probability, and resort value.
    sorted_matrix = sortrows([prob2; level]', 1, 'descend')';
    sorted_prob2 = sorted_matrix(1, :);
    sorted_level = sorted_matrix(2, :);

    nonzero_prob = sorted_prob2(sorted_prob2 > 0);
    nonzero_level = sorted_level(1:size(nonzero_prob,2));

    num = nnz(nonzero_prob);

    % Applying Huffman coding
    % prob = [0.25, 0.25, 0.2, 0.15, 0.15];  % Example Probability
    % symbols = {'A', 'B', 'C', 'D', 'E'};  % Example Symbols
    prob = nonzero_prob;
    symbols = arrayfun(@(x) num2str(x), nonzero_level, 'UniformOutput',
false);

    num = nnz(prob);
    HuffmanTree = cell(num, 1);

    for i = 1:num
        HuffmanTree{i} = {symbols{i}, prob(i), {}, {}, {}};  % Each node：
{Symbol, Prob, LeftNode, RightNode， Huffman Code}
    end

    % Creating Huffman Tree
    while length(HuffmanTree) > 1
        [~, order] = sort(cellfun(@(x) x{2}, HuffmanTree), 'descend');
        HuffmanTree = HuffmanTree(order);

        newNode = {[], HuffmanTree{end}{2} + HuffmanTree{end-1}{2},
HuffmanTree{end}, HuffmanTree{end-1}, {}};

        % Add new node and remove the last node
        HuffmanTree{end-1} = newNode;
        HuffmanTree(end) = [];
    end

    % Generate Huffman code
    HuffmanTree = HuffmanCode(HuffmanTree);  % From the root node

    % Calculate Entropy
    H = 0;
    for i = 1:num
        H = H - prob(i)*log2(prob(i));
    end
```

```
    % Encode the speech signal
    symbolCodes = ExtractHuffmanCodes(HuffmanTree);
    Length = 0;
    for i = 1:num
        prob = symbolCodes{i, 2};  % Extract probability
        codeLength = length(symbolCodes{i, 3});  % Calculate the length of
the Huffman code
        Length = Length + (prob * codeLength);  % Accumulate probability
multiplied by code length
    end

    % Calculate the maximum efficiency of Huffman code for this speech
signal
    Efficiency = (H / Length)*100;

    % Encode the speech signal
    encoded_audio = cell(length(quantized_sound2), 1);
    codeMap = cell(num, 2);

    for i = 1:num
        codeMap{i, 1} = symbolCodes{i, 1};  % Quantization levels
        codeArray = cell2mat(symbolCodes{i, 3});  % Convert 1x4 cell to 1x4
array
        codeStr = num2str(codeArray);  % Convert numeric array to string
        codeStr = strrep(codeStr, ' ', '');  % Remove spaces
        codeMap{i, 2} = codeStr;  % Huffman encoding
    end

    for i = 1:length(quantized_sound2)
        levelStr = num2str(quantized_sound2(i));  % Convert quantization
level to string
        for j = 1:num
            if strcmp(codeMap{j, 1}, levelStr)
                encoded_audio{i} = codeMap{j, 2};  % Add corresponding
Huffman code to the result cell array
                break;
            end
        end
    end

    % Iterate through each element in encoded_audio
    encoded_audio_string = strcat(encoded_audio{:});  % Concatenate into
the final string

    % fprintf('Length of compression is: %d\n',
length(encoded_audio_string));
end
```

```
% Decode the speech signal
function decoded_audio = HuffmanDecoding(encoded_audio_string, codeMap)
    decoded_audio = [];
    start_index = 1;

    while start_index <= length(encoded_audio_string)
        match_found = false; % Flag
        for i = 1:size(codeMap,1)
            huffmanCode = codeMap{i,2};
            codeLength = length(huffmanCode);
            if start_index + codeLength - 1 <= length(encoded_audio_string)
&& ...
                    strcmp(encoded_audio_string(start_index:start_index +
codeLength - 1), huffmanCode)

                decoded_audio = vertcat(decoded_audio,
str2double(codeMap{i, 1}));
                start_index = start_index + codeLength;
                match_found = true;
                break; % Break the loop after a match
            end
        end

        if ~match_found
            warning('No match found for the sequence starting at index
%d.', start_index);
            break;
        end
    end
end


function Check(quantized_sound2, decoded_audio, tolerance)
    % Check if the two arrays are exactly equal
    are_equal = isequal(quantized_sound2, decoded_audio);

    % Or compare using a certain tolerance
    % tolerance = 1e-5;
    errors = abs(quantized_sound2 - decoded_audio);
    are_close = all(errors < tolerance);

    % disp(['Arrays are equal: ', num2str(are_equal)]);
    % disp(['Arrays are close within tolerance: ', num2str(are_close)]);
    if are_close == 0
        disp(['Arrays are not close within tolerance'])
    end
end


function nodes = HuffmanCode(nodes)  % Build Code Huffman Tree
```

```
    if length(nodes) == 1
        nodes = nodes{1}; % Extract rootnode
    end

    if ~isempty(nodes{3})  % There is left node
        nodes{3}{5} = [nodes{5} 0];
        nodes{3} = HuffmanCode(nodes{3});
    end

    if ~isempty(nodes{4})  % There is right node
        nodes{4}{5} = [nodes{5} 1];
        nodes{4} = HuffmanCode(nodes{4});
    end
end

function symbolCodes = ExtractHuffmanCodes(Nodes)
    if length(Nodes) == 1
        Nodes = Nodes{1};  % Extract rootnode
    end

    symbolCodes = {};%{Symbol, Prob, Huffman Code}

    % if this is a leaf node
    if isempty(Nodes{3}) && isempty(Nodes{4})
        symbolCodes = {Nodes{1}, Nodes{2}, Nodes{5}};
    else
        % Otherwise, recursively traverse the tree
        if ~isempty(Nodes{3})  % If there is a left child node
            codesLeft = ExtractHuffmanCodes(Nodes{3});  % Recursively
obtain the encoding of the left child node
            symbolCodes = [symbolCodes; codesLeft];  % Merge cell arrays
        end
        if ~isempty(Nodes{4})  % If there is a right child node
            codesRight = ExtractHuffmanCodes(Nodes{4});  % Recursively
obtain the encoding of the right child node
            symbolCodes = [symbolCodes; codesRight];  % Merge cell arrays
        end
    end
end

function quantized_sound = quantizeSound(sound, level)
    % Map the sound value to the corresponding level
    differences = abs(sound - reshape(level, [1, 1, length(level)]));
    [~, indices] = min(differences, [], 3);
    quantized_sound = level(indices)';
end

function encoded_audio_string = HuffmanEncode(quantized_sound, codeMap)
    encoded_audio = cell(length(quantized_sound), 1);
```

```
    for i = 1:length(quantized_sound)
        levelStr = num2str(quantized_sound(i));

        for j = 1:size(codeMap, 1)
            if strcmp(codeMap{j, 1}, levelStr)
                encoded_audio{i} = codeMap{j, 2};
                break;
            end
        end
    end

    encoded_audio_string = strcat(encoded_audio{:});
end
```