EEE 431
Computational Assignment I

Name: Gökcan
Surname: Kahraman
ID: 21702271
Date: 21.03.2023

Introduction:

On that assignment, we need to study basic source coding algorithms we learned duting the class. Those basic source coding algorithms ensure that messages are sent in a smaller size than normal when they are sent to the intended destinations. Those algoritms are two different types which names are Huffman and Lempel Ziv encoding/decoding systems. We tried to answer five different questions in the assignment by coding the programmes here via MATLAB.

Answers:

1) On the first question, we need to create a discrete memoryless source. We have an alphabet which interval from one to five. Also, we have their probability mass functions which listed (½ , ¼ , 1/12 , 1/12 , 1/12). On that question, we need to design a binary Huffman code by taking 10.000 samples. We need to encode it and calculate expected length by algorithm. First of all, our condition for getting Huffman algorithm is:

H(X) = - ½ * log ½ – ¼ * log ¼ – 1/4 * log 1/12 = 0.15 + 0.15 + 0.27 = 0.57 bits/sample

To encode it, we only need R > 0.57 bits/sample. After that, I designed my design:

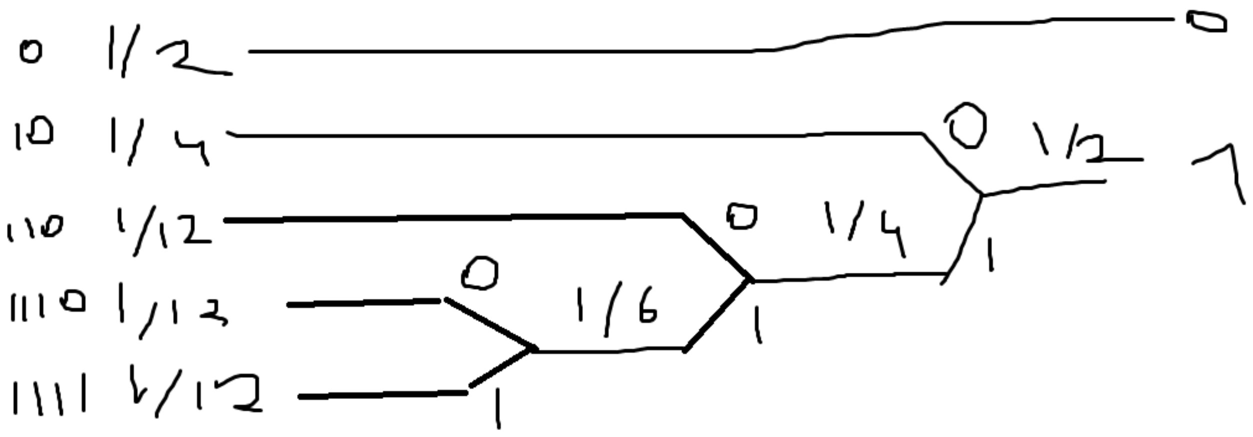

Image 1: Basic Huffman algorithm for question 1

After finding our encoding versions, I need to find entropy which is:

R = 1 * ½ + 2 * ¼ + 3 * 1/12 + 4 * 1/12 + 4 * 1/12 = 1 + ¼ + 2/3 = 23/12 = 1.92 bits/source output (entropy)

For that thing, our expected length would be = 10000 * 1.92 = 19200

On MATLAB, after my coding, its result is 19240.

2) For second question, we need to answer that problem by taking two outputs at the same time. When we do this, our alphabet and its probability mass function is:

Alphabet	**Probability Mass Function**
11	¼
12	1/8
13	1/24
14	1/24
15	1/24
21	1/8
22	1/16
23	1/48
24	1/48
25	1/48
31	1/24
32	1/48
33	1/144
34	1/144
35	1/144
41	1/24
42	1/48
43	1/144
44	1/144
45	1/144
51	1/24
52	1/48
53	1/144
54	1/144
55	1/144

List 1: Our alphabet and its probability mass function for question 2

In this part of our question, our algorithm will be just like in the first question. However, I used the ready-made code in the MATLAB part of this. The reason I did this was that there were five different parameters for my code that I prepared in the first part. Since there are exactly twenty-five different branches in this one, I would have to write twenty-four different cases. Since this would be too long, I used MATLAB's "huffmandeco" code. **Otherwise, if we do it with the previous question, it will give the same result.**

When we do our calculations, our entropy is 3.8264 which is larger than first question. According to our work, when we combine our outputs, our entropy becomes bigger than single output. Also our expected length is 38290.

3) On third question, it has two parts. Firstly, we need to compute entropy for Huffman code which probability mass function is [0.48 0.22 0.1 0.1 0.1]. After that, we need to generate 10.000 samples based on first questions and we need to solve those by Huffman encoding by using [0.48 0.22 0.1 0.1 0.1] probability mass function. For part a:

H(X) = - 0.48 * log 0.48 – 0.22* log 0.22 – 0.3 * log 0.1 = 0.15 + 0.14 + 0.3 = 0.59 bits/sample

To encode it, we only need R > 0.59 bits/sample. After that, I designed my design:

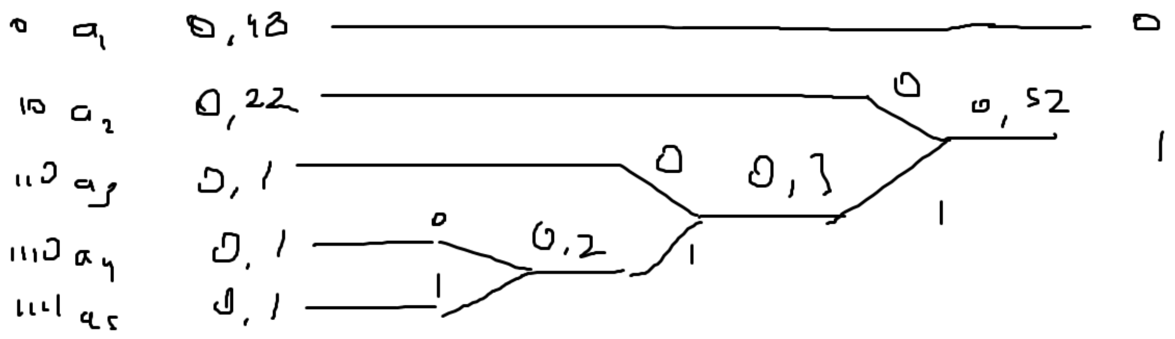

Image 2: Basic Huffman algorithm for question 3

After finding our encoding versions, I need to find entropy which is:

R = 1 * 0.48+ 2 * 0.22 + 3 * 0.1 + 4 * 0.1 + 4 * 0.1 =0.48 + 0.44 + 1.1 = 23/12 = 2.02 bits/source output (entropy)

For that thing, our expected length would be = 10000 * 2.02 = 20200

For part b, we need to use MATLAB by generating 10.000 samples using question 1. After that result, our encoded length is equal to 19244 while our entropy becomes 2.02. Our expected length is not around 20200 because when we generate our random samples by using one PMF, it becomes to use that. Normally, our function has 0.48 probability for a1. But we generate a1 by the probability of 0.1. This is the main reason why our expected length is closer than first questions calculations.

4) On fourth question, we need to write a MATLAB code that implements Lempel ziv algorithm by using binary alphabet. Also, we have Bernoulli random variables with probability of “0” of 0.9 and probability of “1” of 0.1. We have two different parts and we need to generate 100.000 and 1.000.000 realizations and we need to say that compression rate.

First of all, when we check 100.000 realization with my algorithm, we sent our 100.000 messages by making 5185 different dictionary path. Therefore, our compression rate is %19.28.

Secondly, when we do this operation with 1.000.000 realizations, we have 39166 different dictionary path. Therefpre, our compression rate is %25.53.

5) On last question, we need to do same operation with question 4 but with Lempel Ziv Welch.

First of all, when we check 100.000 realization with my algorithm, we sent our 100.000 messages by making 5233 different dictionary path. Therefore, our compression rate is %19.1.

Secondly, when we do this operation with 1.000.000 realizations, we have 39211 different dictionary path. Therefpre, our compression rate is %25.5.

Conclusion:

As a result, in this assignment we have understood how basic source coding algorithms work. Although Huffman is a bit simpler, it is not very functional when there are too many parameters. Lempel Ziv's, on the other hand, are very functional where there are high parameters. However, I had to wait for a very long time in part b of the fourth and fifth questions. Maybe it is related to my own algorithm, but it may have taken a long time because it is not an easy task to look at a million parameters.

Appencides:

```
% Note: To open those questions, you need to use "Run Sections".
%% Q1
symbols = 1:5; %This is our alphabet.
p = [1/2 1/4 1/12 1/12 1/12]; % This is our pmf.
dictt = cell(5,2); % This is our dictionary for encoding.
dictt{1,1} = 1;
dictt{2,1} = 2;
dictt{3,1} = 3;
dictt{4,1} = 4;
dictt{5,1} = 5;
sum = 0; % This thing is for comparing part in Huffman encoding.
k = 4; % Since we make four comparisons in our code, this number is for comparison.
while k >= 1
   sum = p(k + 1) + p(k);
   if k == 4 % This is for first comparison.
      if p(k + 1) >= p(k)
            dictt{k + 1,2}(k) = 1;
            dictt{k,2}(k) = 0;
      else
            dictt{k,2}(k) = 1;
            dictt{k + 1,2}(k) = 0;
      end
   elseif k == 3 % This is for second comparison.
      if sum >= p(k)
            dictt{k + 2,2}(k) = 1;
            dictt{k + 1,2}(k) = 1;
            dictt{k,2}(k) = 0;
      else
            dictt{k + 2,2}(k) = 1;
            dictt{k,2}(k) = 1;
            dictt{k + 1,2}(k) = 0;
      end
   elseif k == 2 % This is for third comparison.
      if sum >= p(k)
            dictt{k + 3,2}(k) = 1;
            dictt{k + 2,2}(k) = 1;
            dictt{k + 1,2}(k) = 1;
            dictt{k,2}(k) = 0;
      else
            dictt{k + 3,2}(k) = 1;
            dictt{k + 2,2}(k) = 1;
            dictt{k + 1,2}(k) = 0;
            dictt{k,2}(k) = 1;
      end
   elseif k == 1 % This is for last comparison.
      if sum >= p(k)
         dictt{k + 4,2}(k) = 1;
         dictt{k + 3,2}(k) = 1;
         dictt{k + 2,2}(k) = 1;
         dictt{k + 1,2}(k) = 1;
         dictt{k,2}(k) = 0;
      else
         dictt{k + 4,2}(k) = 1;
         dictt{k + 3,2}(k) = 1;
         dictt{k + 2,2}(k) = 1;
         dictt{k + 1,2}(k) = 0;
         dictt{k,2}(k) = 1;
```

```
        end
    end
k = k - 1;
end


a = 1;
entropy = 0;
while a < 6 % This is for calculating entropy.
    entropy = entropy + length(dictt{a,2})*p(a);
    a = a + 1;
end
inputSig = randsrc(10000,1,[symbols;p]); % This is for our random symbols
fixedentropy = fix(entropy);
% This converts original signal to binary and it shows its lenght.
binarySig = int2bit(inputSig,fixedentropy);
seqLen = numel(binarySig);
% This converts Hufmann encoded to binary, lenght for binary symbols.
aa = 1;
encodedLen = 0;
while aa < 10000
    dictelement = dictt{inputSig(aa),2};
    encodedLen = encodedLen + length(dictelement);
aa = aa + 1;
end

%% Q2

% As I wrote report, I used "huffmanenco" algorithm.
% Because when I started to write same algorithm with question 1,
% There will be 24 different branches which is very long.
% For decreasing my work, I used "huffmanenco".
% ! Otherwise, if we do it with the previous question, it will give the same result. !
% https://www.mathworks.com/help/comm/ref/huffmanenco.html
symbols = [11 12 13 14 15 21 22 23 24 25 31 32 33 34 35 41 42 43 44 45 51 52 53 54 55]; % This is our alphabet.
p = [1/4 1/8 1/24 1/24 1/24 1/8 1/16 1/48 1/48 1/48 1/24 1/48 1/144 1/144 1/144 1/24 1/48 1/144 1/144 1/144 1/24 1/48
1/144 1/144 1/144]; % This is its probability mass function.
dict = huffmandict(symbols,p); % This is our huffman dictionary.
inputSig = randsrc(10000,1,[symbols;p]); % This generates random symbols.
encd = huffmanenco(inputSig,dict); % This makes huffman encoding
% To calculate total entropy
entropy = 0;
for i = 1:length(symbols)
    addedentropy = p(i)*length(dict{i, 2});
    entropy = entropy + addedentropy;
end
fixedentropy = fix(entropy);
% Making our huffman encoding - decoding true
sig = huffmandeco(encd,dict);
equality = isequal(inputSig,sig);
% This converts original signal to binary and it shows its lenght.
binarySig = int2bit(inputSig,fixedentropy);
seqLen = numel(binarySig);
% This converts Hufmann encoded to binary, lenght for binary symbols.
binaryComp = int2bit(encd,fixedentropy);
encodedLen = numel(binaryComp);

%% Q3

symbols = 1:5; %This is our alphabet.
```

```
p1 = [1/2 1/4 1/12 1/12 1/12]; % This is our pmf1.
p2 = [0.48 0.22 0.1 0.1 0.1]; % This is our pmf1.
dictt = cell(5,2); % This is our dictionary for encoding.
dictt{1,1} = 1;
dictt{2,1} = 2;
dictt{3,1} = 3;
dictt{4,1} = 4;
dictt{5,1} = 5;
sum = 0; % This thing is for comparing part in Huffman encoding.
k = 4; % Since we make four comparisons in our code, this number is for comparison.
while k >= 1
  sum = p2(k + 1) + p2(k);
  if k == 4 % This is for first comparison.
    if p2(k + 1) >= p2(k)
        dictt{k + 1,2}(k) = 1;
        dictt{k,2}(k) = 0;
    else
        dictt{k,2}(k) = 1;
        dictt{k + 1,2}(k) = 0;
    end
  elseif k == 3 % This is for second comparison.
    if sum >= p2(k)
        dictt{k + 2,2}(k) = 1;
        dictt{k + 1,2}(k) = 1;
        dictt{k,2}(k) = 0;
    else
        dictt{k + 2,2}(k) = 1;
        dictt{k,2}(k) = 1;
        dictt{k + 1,2}(k) = 0;
    end
  elseif k == 2 % This is for third comparison.
    if sum >= p2(k)
        dictt{k + 3,2}(k) = 1;
        dictt{k + 2,2}(k) = 1;
        dictt{k + 1,2}(k) = 1;
        dictt{k,2}(k) = 0;
    else
        dictt{k + 3,2}(k) = 1;
        dictt{k + 2,2}(k) = 1;
        dictt{k + 1,2}(k) = 0;
        dictt{k,2}(k) = 1;
    end
  elseif k == 1 % This is for last comparison.
    if sum >= p2(k)
      dictt{k + 4,2}(k) = 1;
      dictt{k + 3,2}(k) = 1;
      dictt{k + 2,2}(k) = 1;
      dictt{k + 1,2}(k) = 1;
      dictt{k,2}(k) = 0;
    else
      dictt{k + 4,2}(k) = 1;
      dictt{k + 3,2}(k) = 1;
      dictt{k + 2,2}(k) = 1;
      dictt{k + 1,2}(k) = 0;
      dictt{k,2}(k) = 1;
    end
  end
k = k - 1;
end
```

```
a = 1;
entropy = 0;
while a < 6 % This is for calculating entropy.
    entropy = entropy + length(dictt{a,2})*p2(a);
    a = a + 1;
end
inputSig = randsrc(10000,1,[symbols;p1]); % This is for our random symbols
fixedentropy = fix(entropy);
% This converts original signal to binary and it shows its lenght.
binarySig = int2bit(inputSig,fixedentropy);
seqLen = numel(binarySig);
% This converts Hufmann encoded to binary, lenght for binary symbols.
aa = 1;
encodedLen = 0;
while aa < 10000
    dictelement = dictt{inputSig(aa),2};
    encodedLen = encodedLen + length(dictelement);
aa = aa + 1;
end

%% Q4
binaries = 0:1; % This is our binaries.
prob = [0.9 0.1]; % This is our probabilities.
randlist = randsrc(1000000,1,[binaries;prob]); % This is our random list. If you want to check, you need to write
100.000 to 1.000.000
randlist(end+1) = "#"; % For all Lempel Ziv algorithm, we need to have hashtag for ending.
dict = ["#"; 0; 1]; % This is our dictionary.
nxtdict = length(dict) + 1; % This is our adding dictionary for checking next variables.
k = 1;
cont = ""; % This side is for controlling digits for adding new element to dictionary.
cont = cont + randlist(1);
% This side is for encoding.
encd = zeros(1,length(randlist));
incdt = 1;
while k < length(randlist)
    for m = 1:length(dict)
        if cont == dict(m)
            add = m;
            nextt = "";
            nextt = nextt + randlist(k+1);
            if nextt == '#'
                k = length(randlist)+1;
                break
            end
            cont = cont + nextt;
            k = k+1;
            m = 1;
        end
    end
    if nextt ~= '#'
        encd(incdt) = add;
        incdt = incdt + 1;
        dict(nxtdict) = cont;
        cont = nextt;
        nxtdict = nxtdict + 1;

    else
        encd(incdt) = add;
        incdt = incdt + 1;
    end
end
encd = nonzeros(encd) - 1;

% Since we convert all digits zero and one for dictionary, this is for encoding part.
tot = length(de2bi(length(dict)));
binaryencd = zeros(length(encd), tot);
for k = 1:length(encd)
    binaryencd(k,:) =  [zeros(1,tot-length(de2bi(encd(k)))) de2bi(encd(k), 'left-msb')];
end
% For getting our values, you can use those commands which names are:
% disp(dict), disp(randlist) and disp(encd)

%% Q5

p0 = 0.9;  % This is our probability for 0
p1 = 0.1;  % This is our probability for 1
N = 1000000;  % This is number of source output, you can change it for 100.000
totdict = 2;  % This is our initial dictionary.

% This is for generating random samples
src = rand(1,N) > p0;

% This is for Lempel-Ziv-Welch encoding part
dict = containers.Map({'0','1'},{0,1});  % This is our dictionary part.
encd = [];  % This is our encoded message
seq = '';  % This is our current check part
for i = 1:N
    nxt = num2str(src(i));  % This is for next output
    if isKey(dict, [seq nxt])  % This is for putting our variables in dictionary
        seq = [seq nxt];
    else  % This is for not part of dictionary
        encd(end+1) = dict(seq);  % Adding new variable
        dict([seq nxt]) = totdict;  % Add new entry to dictionary
        totdict = totdict + 1;  % Increase our dictionary by one
        seq = nxt;  % Start next sequence
    end
end
encd(end+1) = dict(seq);  % This is for last sequence which is "#"
```