Lab4: Source coding

B07901103 電機三 陳孟宏

Note: In problem 1, I use “python”, while I use “MATLAB” in problem 2&3

<Problem 1>

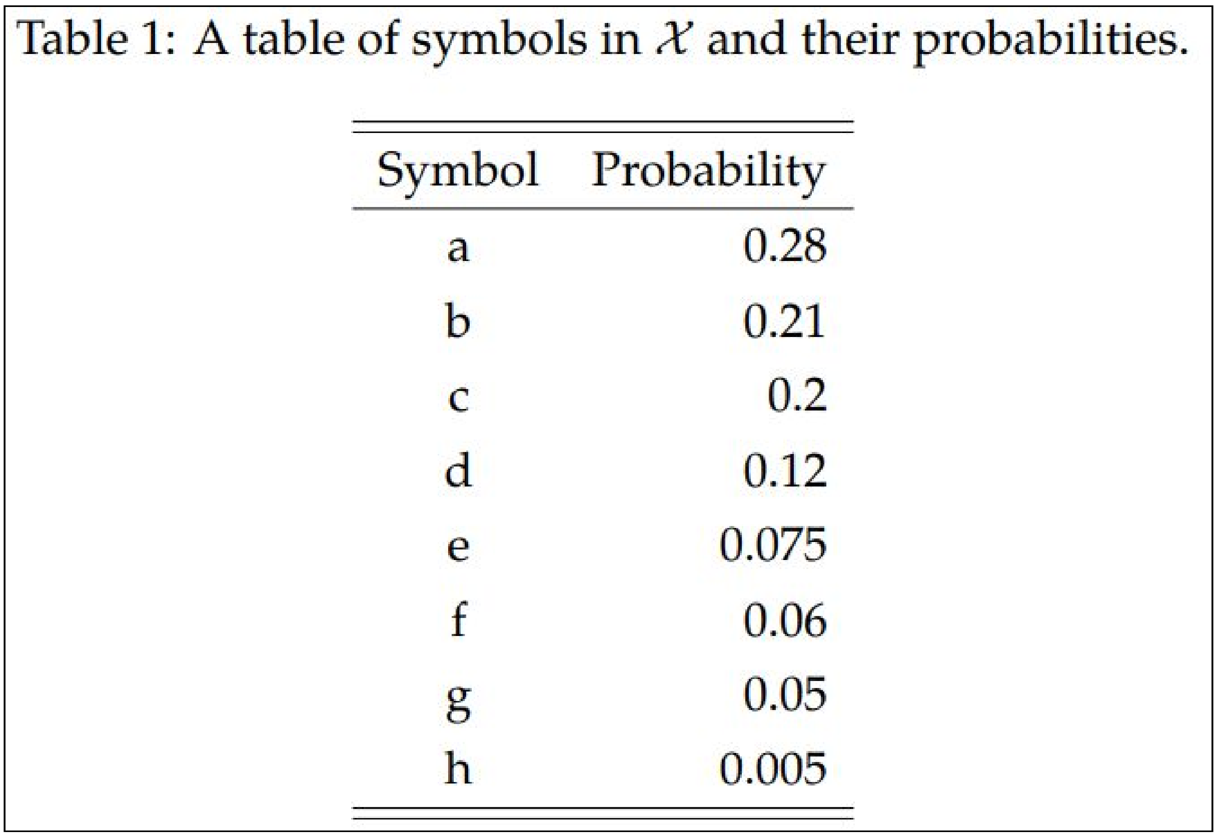
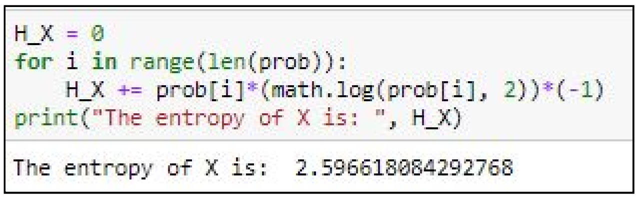


Table 1: A table of symbols in $\mathcal{X}$ and their probabilities.

Symbol	Probability
a	0.28
b	0.21
c	0.2
d	0.12
e	0.075
f	0.06
g	0.05
h	0.005

(a) By the definition of H[X], H[X] = -Σ(p*log(p)), hence, for the Table1, its entropy = 2.596618084292768

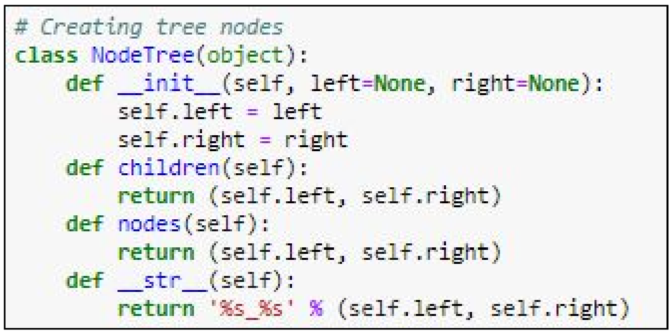


```
H_X = 0
for i in range(len(prob)):
    H_X += prob[i]*(math.log(prob[i], 2))*(-1)
print("The entropy of X is: ", H_X)
```

```
The entropy of X is:  2.596618084292768
```

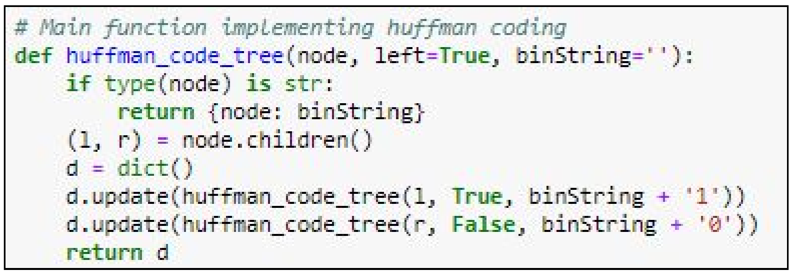
(b)

- First, I create a “class” to create a tree with children.



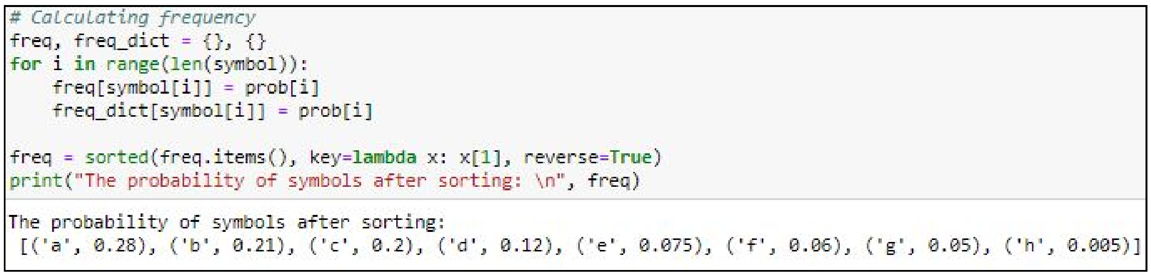
```
# Creating tree nodes
class NodeTree(object):
    def __init__(self, left=None, right=None):
        self.left = left
        self.right = right
    def children(self):
        return (self.left, self.right)
    def nodes(self):
        return (self.left, self.right)
    def __str__(self):
        return '%s_%s' % (self.left, self.right)
```

- Second, I define a function to encode the bits by Huffman tree.



```python
# Main function implementing huffman coding
def huffman_code_tree(node, left=True, binString=''):
    if type(node) is str:
        return {node: binString}
    (l, r) = node.children()
    d = dict()
    d.update(huffman_code_tree(l, True, binString + '1'))
    d.update(huffman_code_tree(r, False, binString + '0'))
    return d
```

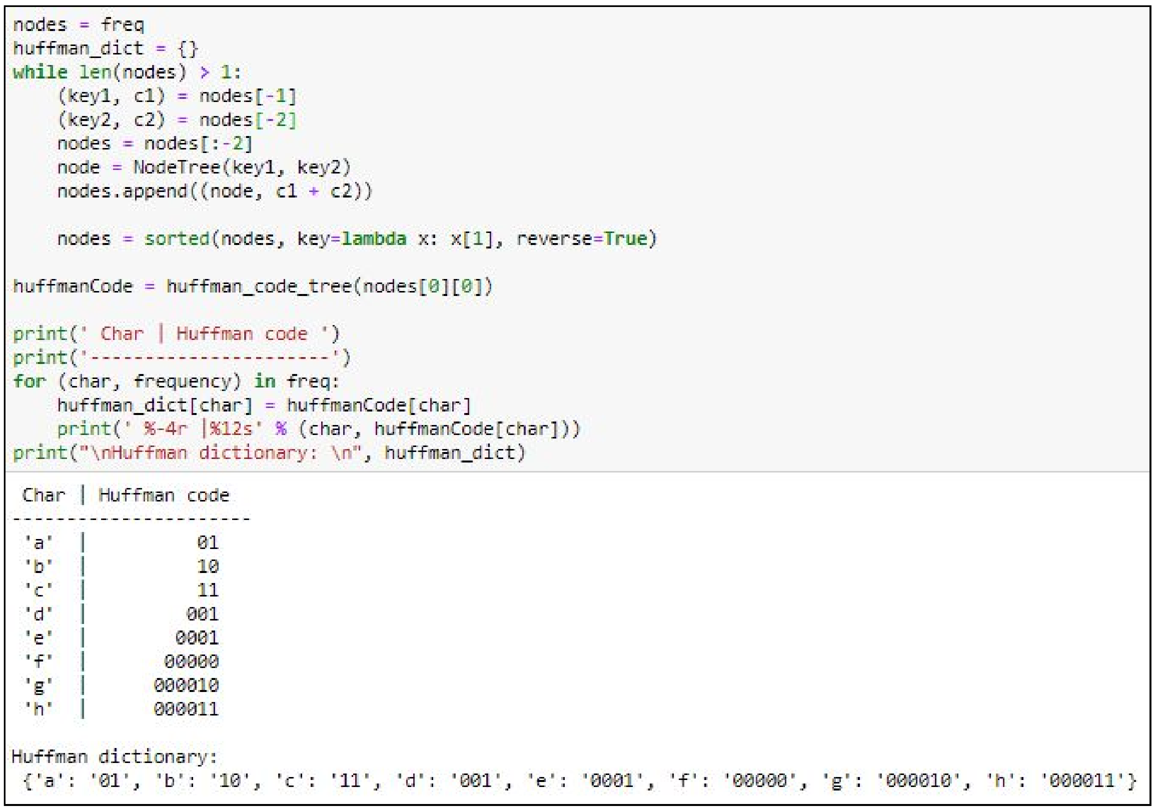
- Third, store their frequency, that is, the probability.



```python
# Calculating frequency
freq, freq_dict = {}, {}
for i in range(len(symbol)):
    freq[symbol[i]] = prob[i]
    freq_dict[symbol[i]] = prob[i]

freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)
print("The probability of symbols after sorting: \n", freq)
```

```
The probability of symbols after sorting:
 [('a', 0.28), ('b', 0.21), ('c', 0.2), ('d', 0.12), ('e', 0.075), ('f', 0.06), ('g', 0.05), ('h', 0.005)]
```

- At last, show the Huffman tree I create based on Table1.



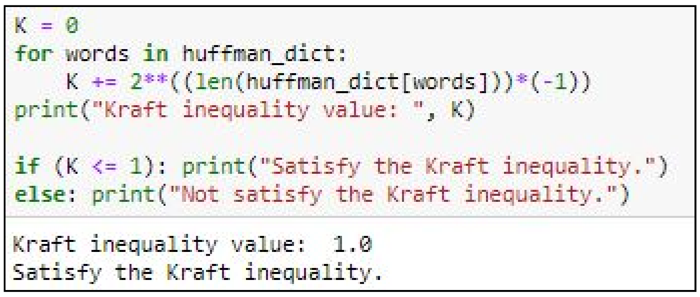
```python
nodes = freq
huffman_dict = {}
while len(nodes) > 1:
    (key1, c1) = nodes[-1]
    (key2, c2) = nodes[-2]
    nodes = nodes[:-2]
    node = NodeTree(key1, key2)
    nodes.append((node, c1 + c2))

    nodes = sorted(nodes, key=lambda x: x[1], reverse=True)

huffmanCode = huffman_code_tree(nodes[0][0])

print(' Char | Huffman code ')
print('----------------------')
for (char, frequency) in freq:
    huffman_dict[char] = huffmanCode[char]
    print(' %-4r |%12s' % (char, huffmanCode[char]))
print("\nHuffman dictionary: \n", huffman_dict)
```

```
 Char | Huffman code
----------------------
 'a'  |          01
 'b'  |          10
 'c'  |          11
 'd'  |         001
 'e'  |        0001
 'f'  |       00000
 'g'  |      000010
 'h'  |      000011

Huffman dictionary:
 {'a': '01', 'b': '10', 'c': '11', 'd': '001', 'e': '0001', 'f': '00000', 'g': '000010', 'h': '000011'}
```

(c) By the definition of “Kraft inequality”, we know that “X” satisfies.

$\mathcal{X} = \{a_1, \ldots, a_M\}$ with codeword lengths $\{l(a_j); 1 \le j \le M\}$

$$\sum_{j=1}^{M} 2^{-l(a_j)} \le 1.$$

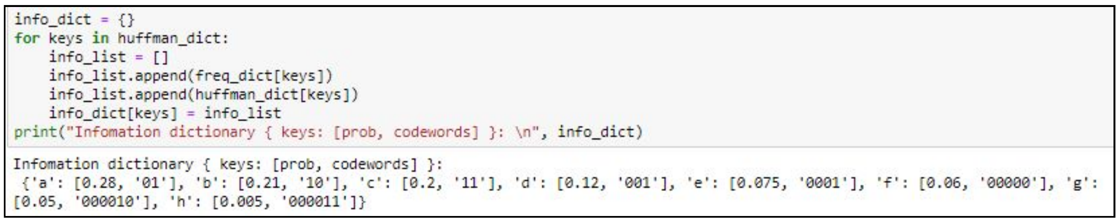


```
K = 0
for words in huffman_dict:
    K += 2**((len(huffman_dict[words]))*(-1))
print("Kraft inequality value: ", K)

if (K <= 1): print("Satisfy the Kraft inequality.")
else: print("Not satisfy the Kraft inequality.")
```

```
Kraft inequality value:  1.0
Satisfy the Kraft inequality.
```

(d)

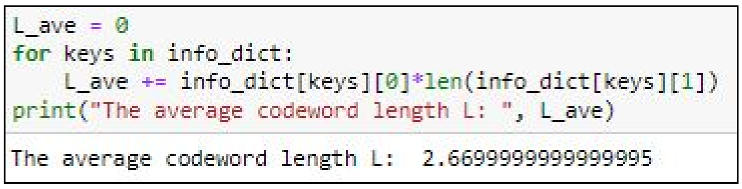
- First, I build a dictionary with (key, value) = (symbol, [prob, encoded bits]).



```
info_dict = {}
for keys in huffman_dict:
    info_list = []
    info_list.append(freq_dict[keys])
    info_list.append(huffman_dict[keys])
    info_dict[keys] = info_list
print("Infomation dictionary { keys: [prob, codewords] }: \n", info_dict)
```

```
Infomation dictionary { keys: [prob, codewords] }:
 {'a': [0.28, '01'], 'b': [0.21, '10'], 'c': [0.2, '11'], 'd': [0.12, '001'], 'e': [0.075, '0001'], 'f': [0.06, '00000'], 'g':
[0.05, '000010'], 'h': [0.005, '000011']}
```

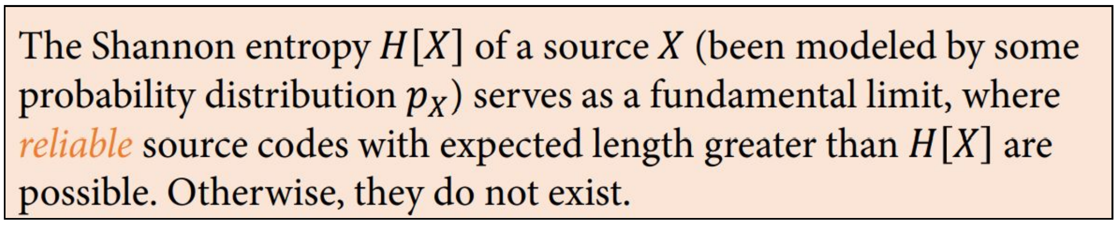
- Second, calculate the average codeword length.



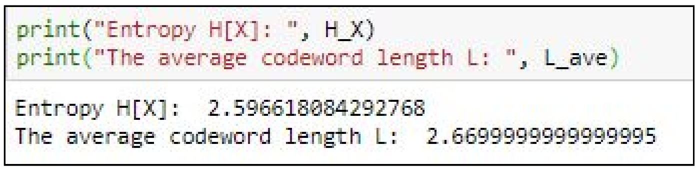
```
L_ave = 0
for keys in info_dict:
    L_ave += info_dict[keys][0]*len(info_dict[keys][1])
print("The average codeword length L: ", L_ave)
```

```
The average codeword length L:  2.6699999999999995
```

- At last, by the definition of "Source Theorem" and my result, we find that it satisfies this theorem.



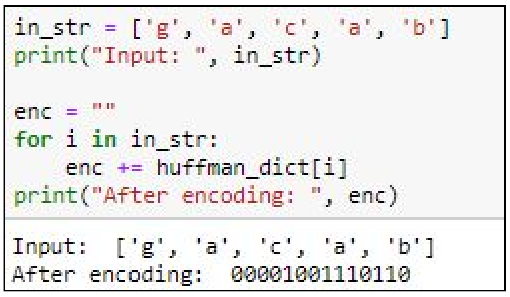
The Shannon entropy $H[X]$ of a source X (been modeled by some probability distribution p_X) serves as a fundamental limit, where *reliable* source codes with expected length greater than $H[X]$ are possible. Otherwise, they do not exist.



```
print("Entropy H[X]: ", H_X)
print("The average codeword length L: ", L_ave)
```

```
Entropy H[X]:  2.596618084292768
The average codeword length L:  2.6699999999999995
```

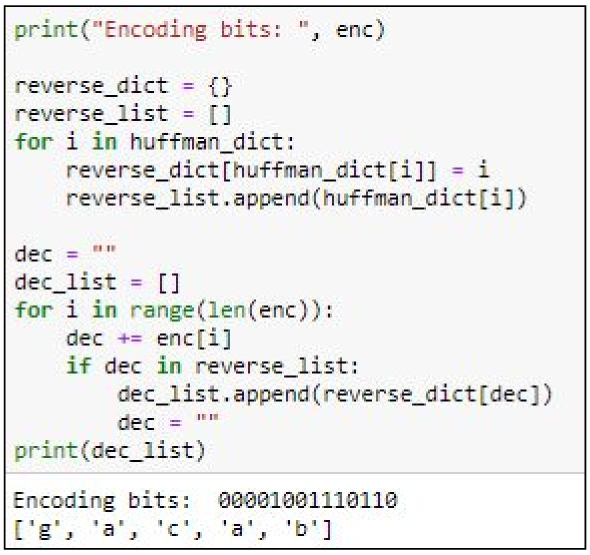
(e) Call the dictionary I create: "huffman_dict" to the input.



```
in_str = ['g', 'a', 'c', 'a', 'b']
print("Input: ", in_str)

enc = ""
for i in in_str:
    enc += huffman_dict[i]
print("After encoding: ", enc)
```

```
Input:  ['g', 'a', 'c', 'a', 'b']
After encoding:  00001001110110
```

(f) I reverse the "huffman_dict" 's (key, value) to be (value, key), and then compare the encoded bits with the dictionary one by one. Because of the uniqueness of the huffman encoding, the decoded result will be unique, too.

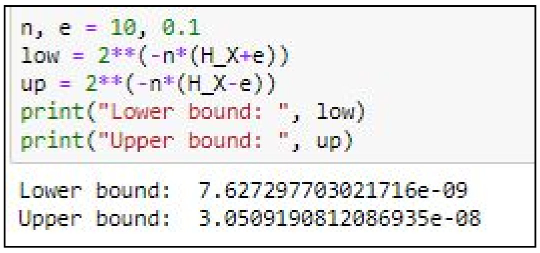


```
print("Encoding bits: ", enc)

reverse_dict = {}
reverse_list = []
for i in huffman_dict:
    reverse_dict[huffman_dict[i]] = i
    reverse_list.append(huffman_dict[i])

dec = ""
dec_list = []
for i in range(len(enc)):
    dec += enc[i]
    if dec in reverse_list:
        dec_list.append(reverse_dict[dec])
        dec = ""
print(dec_list)
```

```
Encoding bits:  00001001110110
['g', 'a', 'c', 'a', 'b']
```

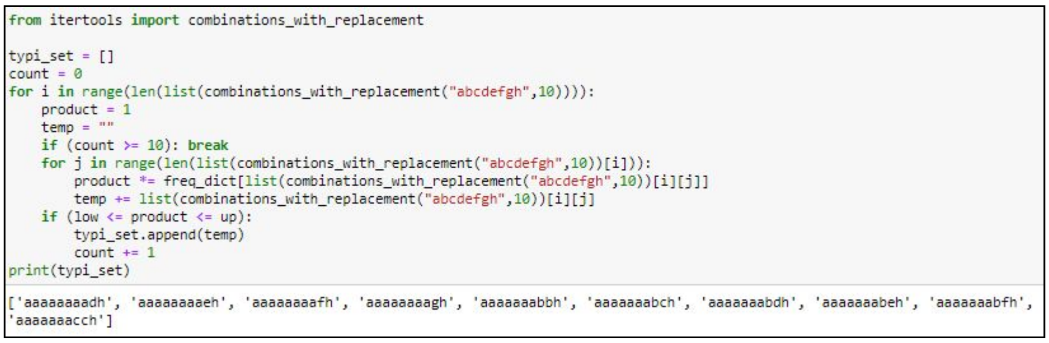
(g) Apply the proved bounds of a “typical set”, and I use a kit called “combinations_with_replacement” to list all of the combinations of alphabets “a~g” in a string with length 10. Give it the upper / lower bound and list 10 of the possibilities of a typical set.

$$T_\varepsilon^n = \left\{ x^n : 2^{-n(H[X]+\varepsilon)} < p_{X^n}(x^n) < 2^{-n(H[X]-\varepsilon)} \right\}$$



```
n, e = 10, 0.1
low = 2**(-n*(H_X+e))
up = 2**(-n*(H_X-e))
print("Lower bound: ", low)
print("Upper bound: ", up)
```

```
Lower bound:  7.627297703021716e-09
Upper bound:  3.0509190812086935e-08
```

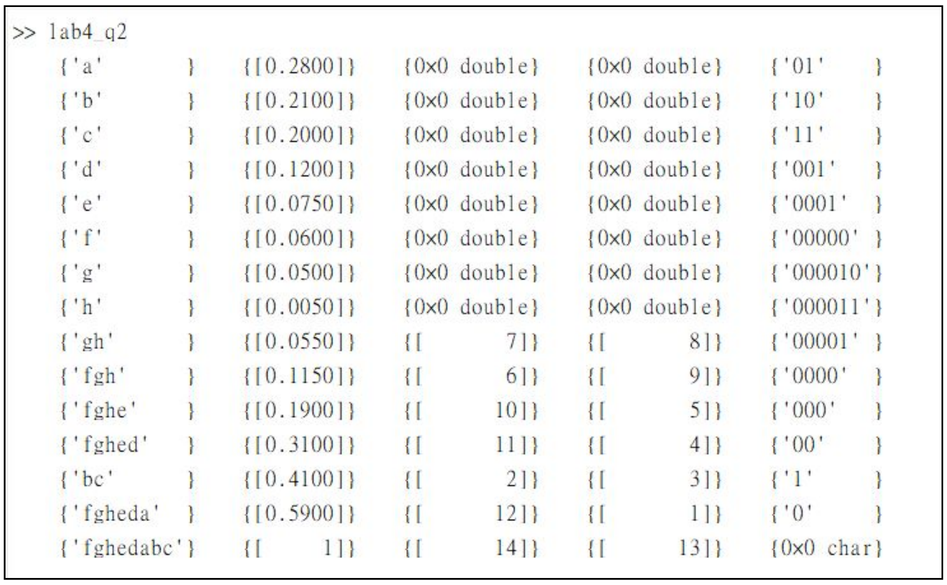


```
from itertools import combinations_with_replacement

typi_set = []
count = 0
for i in range(len(list(combinations_with_replacement("abcdefgh",10)))):
    product = 1
    temp = ""
    if (count >= 10): break
    for j in range(len(list(combinations_with_replacement("abcdefgh",10))[i])):
        product *= freq_dict[list(combinations_with_replacement("abcdefgh",10))[i][j]]
        temp += list(combinations_with_replacement("abcdefgh",10))[i][j]
    if (low <= product <= up):
        typi_set.append(temp)
        count += 1
print(typi_set)
```

```
['aaaaaaaadh', 'aaaaaaaaeh', 'aaaaaaaafh', 'aaaaaaaagh', 'aaaaaaabbh', 'aaaaaaabch', 'aaaaaaabdh', 'aaaaaaabeh', 'aaaaaaabfh',
'aaaaaaacch']
```

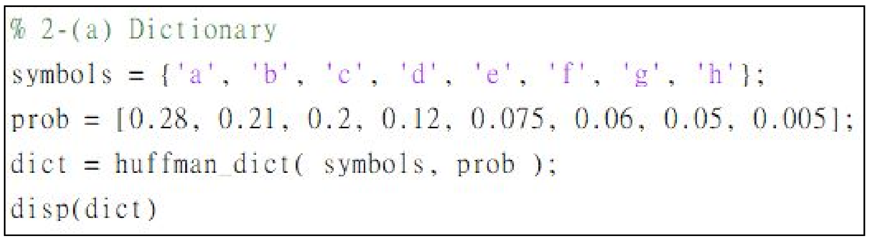
<Problem 2>

(a) Define a function called “huffman_dict”, and the following is the result.



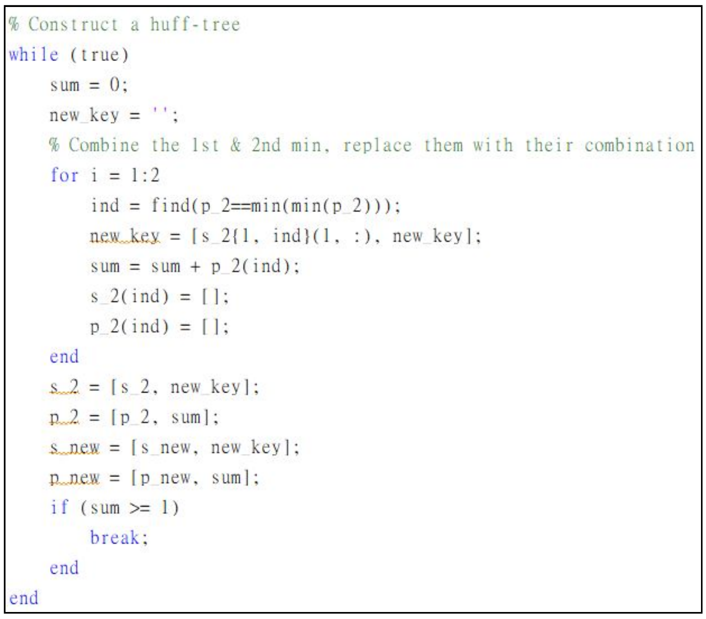
```
>> lab4_q2
    {'a'       }    {[0.2800]}    {0x0 double}    {0x0 double}    {'01'     }
    {'b'       }    {[0.2100]}    {0x0 double}    {0x0 double}    {'10'     }
    {'c'       }    {[0.2000]}    {0x0 double}    {0x0 double}    {'11'     }
    {'d'       }    {[0.1200]}    {0x0 double}    {0x0 double}    {'001'    }
    {'e'       }    {[0.0750]}    {0x0 double}    {0x0 double}    {'0001'   }
    {'f'       }    {[0.0600]}    {0x0 double}    {0x0 double}    {'00000'  }
    {'g'       }    {[0.0500]}    {0x0 double}    {0x0 double}    {'000010' }
    {'h'       }    {[0.0050]}    {0x0 double}    {0x0 double}    {'000011' }
    {'gh'      }    {[0.0550]}    {[         7]}  {[         8]}  {'00001'  }
    {'fgh'     }    {[0.1150]}    {[         6]}  {[         9]}  {'0000'   }
    {'fghe'    }    {[0.1900]}    {[        10]}  {[         5]}  {'000'    }
    {'fghed'   }    {[0.3100]}    {[        11]}  {[         4]}  {'00'     }
    {'bc'      }    {[0.4100]}    {[         2]}  {[         3]}  {'1'      }
    {'fgheda'  }    {[0.5900]}    {[        12]}  {[         1]}  {'0'      }
    {'fghedabc'}    {[       1]}  {[        14]}  {[        13]}  {0x0 char}
```

(i) The parameters we need.



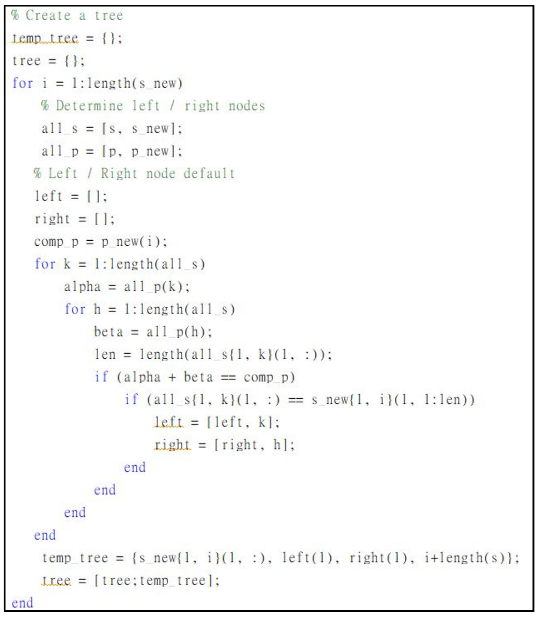
```
% 2-(a) Dictionary
symbols = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'};
prob = [0.28, 0.21, 0.2, 0.12, 0.075, 0.06, 0.05, 0.005];
dict = huffman_dict( symbols, prob );
disp(dict)
```

(ii) Construct a Huffman tree with adding the two minimum probability.



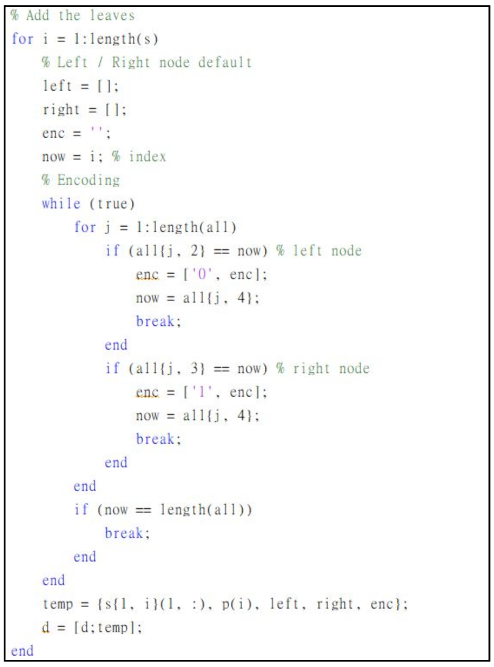
```
% Construct a huff-tree
while (true)
    sum = 0;
    new_key = '';
    % Combine the 1st & 2nd min, replace them with their combination
    for i = 1:2
        ind = find(p_2==min(min(p_2)));
        new_key = [s_2{1, ind}(1, :), new_key];
        sum = sum + p_2(ind);
        s_2(ind) = [];
        p_2(ind) = [];
    end
    s_2 = [s_2, new_key];
    p_2 = [p_2, sum];
    s_new = [s_new, new_key];
    p_new = [p_new, sum];
    if (sum >= 1)
        break;
    end
end
```

(iii) Build a tree with every node having its left and right children. I scan for every member's probability, if two of them's probability sums are equal to a node's probability, then it means the two nodes are respectively the node's children. To determine which of the two are left or right children, I compare the first element of the node's string and the first element of the two nodes' strings.



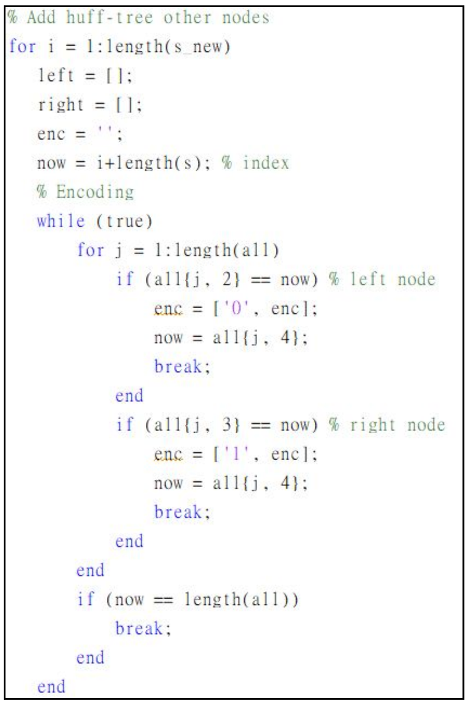
```
% Create a tree
temp_tree = {};
tree = {};
for i = 1:length(s_new)
     % Determine left / right nodes
     all_s = [s, s_new];
     all_p = [p, p_new];
    % Left / Right node default
    left = [];
    right = [];
    comp_p = p_new(i);
    for k = 1:length(all_s)
        alpha = all_p(k);
        for h = 1:length(all_s)
            beta = all_p(h);
            len = length(all_s{1, k}(1, :));
            if (alpha + beta == comp_p)
                if (all_s{1, k}(1, :) == s_new{1, i}(1, 1:len))
                    left = [left, k];
                    right = [right, h];
                end
            end
        end
    end
     temp_tree = {s_new{1, i}(1, :), left(1), right(1), i+length(s)};
     tree = [tree;temp_tree];
end
```

(iv) Add the leaves, that is, the symbols "a~g".Their left and right child are NULL. As for the encoding, I scan for the Huffman tree's left and right node, if the index of the leaf is in the left child, then add a bit of '0', otherwise, add a bit of '1'.

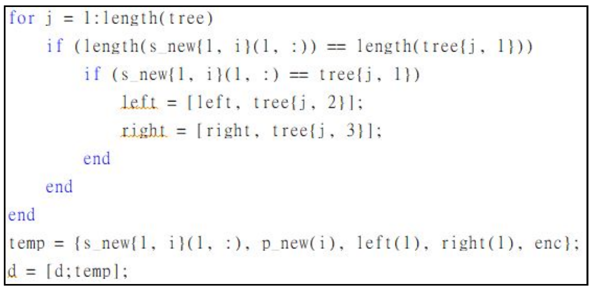


```
% Add the leaves
for i = 1:length(s)
    % Left / Right node default
    left = [];
    right = [];
    enc = '';
    now = i; % index
    % Encoding
    while (true)
        for j = 1:length(all)
            if (all{j, 2} == now) % left node
                enc = ['0', enc];
                now = all{j, 4};
                break;
            end
            if (all{j, 3} == now) % right node
                enc = ['1', enc];
                now = all{j, 4};
                break;
            end
        end
        if (now == length(all))
            break;
        end
    end
    temp = {s{1, i}(1, :), p(i), left, right, enc};
    d = [d;temp];
end
```

(v) By the same token, for nodes except for leaves (that is, the string length of the node will be greater than 1.), we do the same thing as leaf node to encode them.

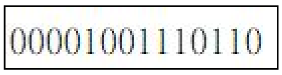


```
% Add huff-tree other nodes
for i = 1:length(s_new)
   left = [];
   right = [];
   enc = '';
   now = i+length(s); % index
   % Encoding
   while (true)
       for j = 1:length(all)
           if (all{j, 2} == now) % left node
               enc = ['0', enc];
               now = all{j, 4};
               break;
           end
           if (all{j, 3} == now) % right node
               enc = ['1', enc];
               now = all{j, 4};
               break;
           end
       end
       if (now == length(all))
           break;
       end
   end
```



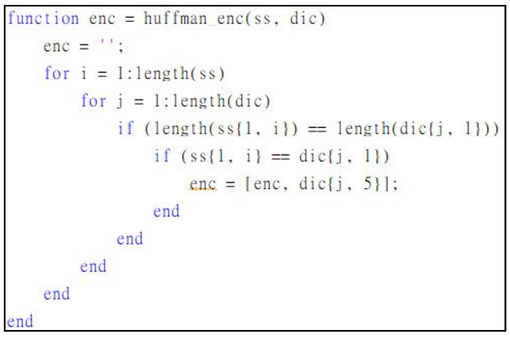
```
for j = 1:length(tree)
    if (length(s_new{1, i}(1, :)) == length(tree{j, 1}))
        if (s_new{1, i}(1, :) == tree{j, 1})
            left = [left, tree{j, 2}];
            right = [right, tree{j, 3}];
        end
    end
end
temp = {s_new{1, i}(1, :), p_new(i), left(1), right(1), enc};
d = [d;temp];
```

(b) Define a function called “huffman_enc”, and the following is the result of encoding [‘g’, ‘a’, ‘c’, ‘a’, ‘b’], which is the same as 1-(e).



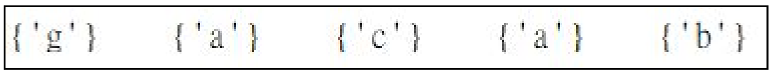
```
00001001110110
```

1. For every member in the input, we encode them based on the Huffman dictionary.



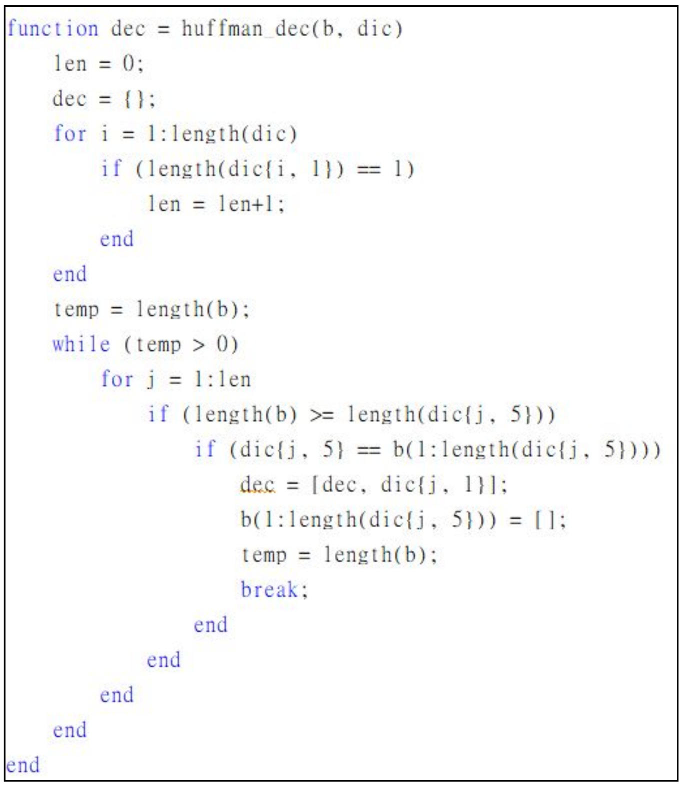
```
function enc = huffman_enc(ss, dic)
    enc = '';
    for i = 1:length(ss)
        for j = 1:length(dic)
            if (length(ss{1, i}) == length(dic{j, 1}))
                if (ss{1, i} == dic{j, 1})
                    enc = [enc, dic{j, 5}];
                end
            end
        end
    end
end
```

(c) Define a function called “huffman_dec”, and the following is the result of decoding ‘00001001110110’, which is the same as 1-(f).



```
{'g'}    {'a'}    {'c'}    {'a'}    {'b'}
```

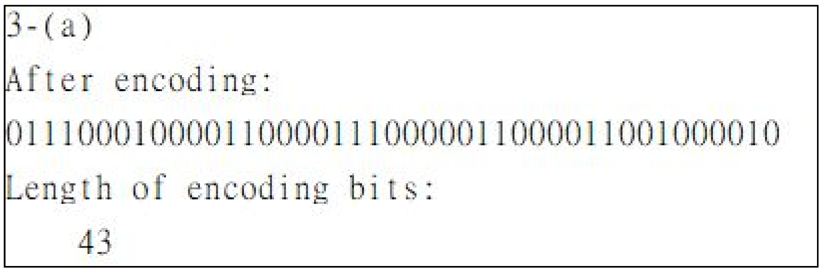
1. Scan for the Huffman dictionary, compare the same length of the i-th dictionary strings with our encoded input.If there is a match, then decode this string based on the Huffman dictionary, and delete the string in the input until the input becoming empty.



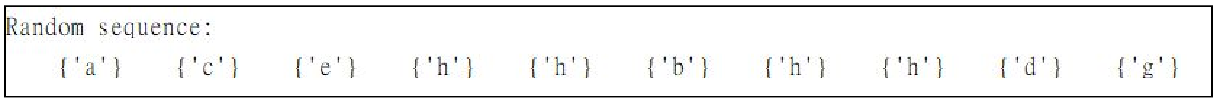
```
function dec = huffman_dec(b, dic)
    len = 0;
    dec = {};
    for i = 1:length(dic)
        if (length(dic{i, 1}) == 1)
            len = len+1;
        end
    end
    temp = length(b);
    while (temp > 0)
        for j = 1:len
            if (length(b) >= length(dic{j, 5}))
                if (dic{j, 5} == b(1:length(dic{j, 5})))
                    dec = [dec, dic{j, 1}];
                    b(1:length(dic{j, 5})) = [];
                    temp = length(b);
                    break;
                end
            end
        end
    end
end
```

<Problem 3>

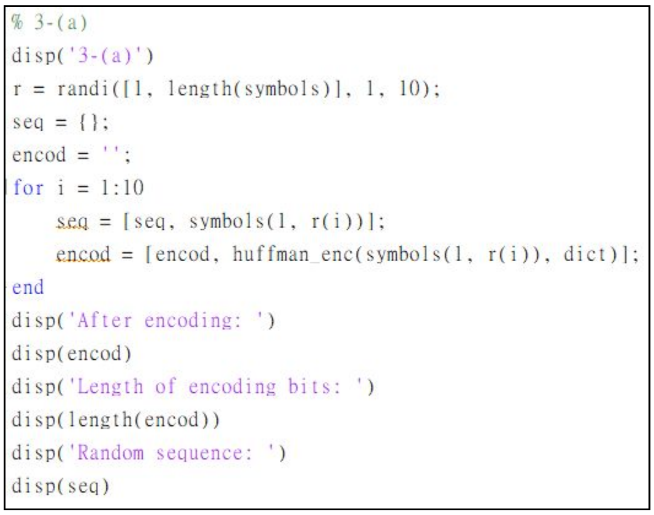
(a) Randomly generate an “n=10” symbols according to Table 1, encode it and calculate the length of this string.



```
3-(a)
After encoding:
0111000100001100001110000011000011001000010
Length of encoding bits:
    43
```

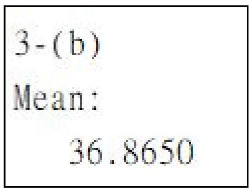


```
Random sequence:
   {'a'}    {'c'}    {'e'}    {'h'}    {'h'}    {'b'}    {'h'}    {'h'}    {'d'}    {'g'}
```

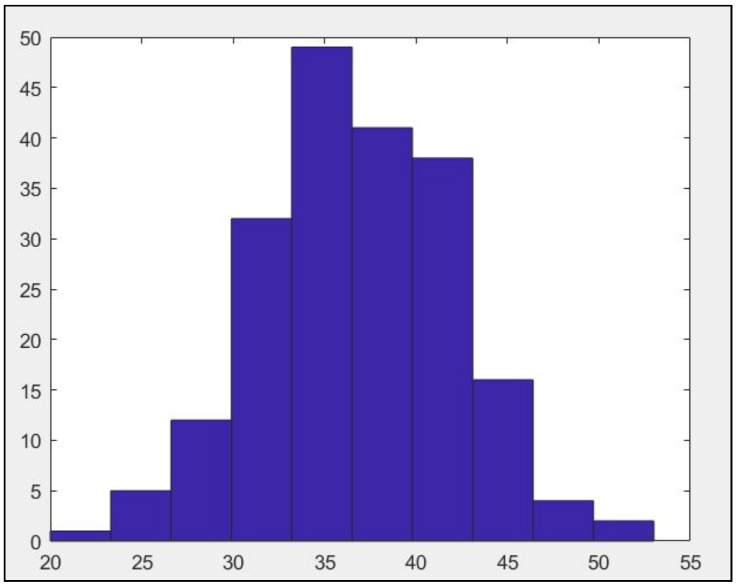


```
% 3-(a)
disp('3-(a)')
r = randi([1, length(symbols)], 1, 10);
seq = {};
encod = '';
for i = 1:10
    seq = [seq, symbols(1, r(i))];
    encod = [encod, huffman_enc(symbols(1, r(i)), dict)];
end
disp('After encoding: ')
disp(encod)
disp('Length of encoding bits: ')
disp(length(encod))
disp('Random sequence: ')
disp(seq)
```

(b) Iterate the step in 3-(a) by 200 times (randomly generate symbols). Use the build-in function "hist" to plot every time histogram. Also, calculate the "mean" average codeword length.

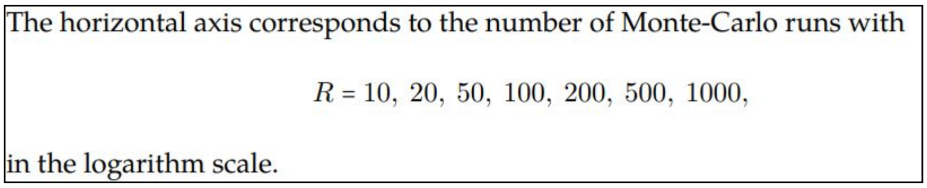


```
3-(b)
Mean:
   36.8650
```



(c) Plot with the specific condition.

(i)

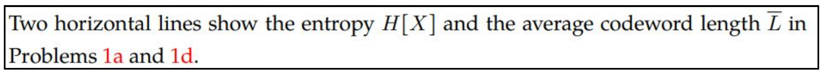


The horizontal axis corresponds to the number of Monte-Carlo runs with

$$R = 10,\ 20,\ 50,\ 100,\ 200,\ 500,\ 1000,$$

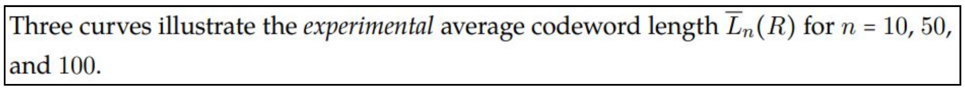
in the logarithm scale.

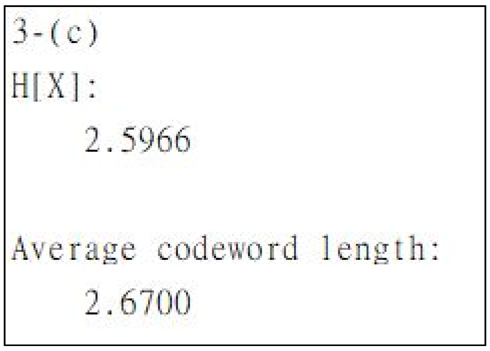
(ii)

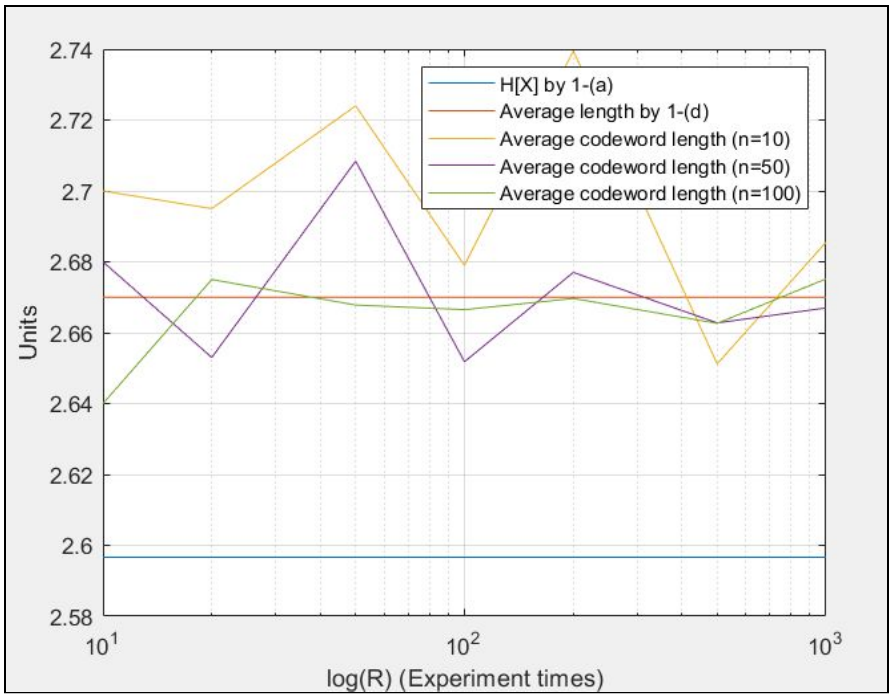


Two horizontal lines show the entropy $H[X]$ and the average codeword length $\overline{L}$ in Problems 1a and 1d.

(iii)


Three curves illustrate the *experimental* average codeword length $\overline{L}_n(R)$ for n = 10, 50, and 100.


```
3-(c)
H[X]:
    2.5966

Average codeword length:
    2.6700
```



(d) Comment on 3-(c)

(i) No matter what "n" is, the average codeword length will oscillate at the "expected" average codeword length, so the three curves are similar.

(ii) By source code theorem, the expected length must be greater than the entropy H[X], so the "orange line" is above the "blue line".

(iii) According to the entropy bounce for prefix-free code, the H[X] is 2.5966 here, so the L_min will be between 2.5966 and 3.5966.

$$H[X] \leq \bar{L}_{\min} < H[X] + 1 \ \frac{\text{bits}}{\text{symbol}}$$