Lempel Ziv 77 implementation code

Submitted in partial fulfilment of Digital Communication (ELL712) course

Name: Dibyajyoti Jena

Entry Number: 2022EEE2712

Language: Python

Environment: Jupyter Notebook

Advantages of Jupyter Notebook: Blockwise code execution, edit and make reports alongside code for explaination

LZ77 Algorithm

Charactericstics of LZ77 Compression Algorithm

1. It is a universal data compressor which operates without prior knowledge of source statistics.
2. Uses variable to variable length codes, in which number of string symbols and number of encoded bits per symbol are variable.
3. Attempts to model the source and encode it simultaneously, with instantaneous decoding.

What we aim to achieve:

1. Creating a dependent source model with a known entropy (defined Markov Source with known symbol probabilities)
2. Implementation (encoding and decoding) of the Lempel Ziv algorithm.
3. Prove that the Lempel Ziv algorithm achieves an $L_{avg-min}$ close to the lower bound **H(X/S)** (conditional entropy)

Setup for the experiment

Let x_1 , $x_2 ... x_n$ be output string of our defined Markov source. M is the alphabet length(number of distinct symbols present). A window is taken from the string, with length as a power of 2, w = 2^z (label z as "window power"). Symbols in the window are encoded without any compression, since length of the window is considered negligible compared to the string length. The rest of the string is considered as a source sequence and encoded following these steps:

Note: x_m^n denotes a string $x_m, x_{m+1} .... x_n$

1. Encode the first w symbols without compression using **ceil**($log_2 M$) bits per symbol.
2. Set a pointer **p = w**. Then x_{p+1} is the first new symbol to be encoded. Find the largest such length of string such that x_{p+1}^{p+n} = x_{p+1-u}^{p+n-u} where u lies between 1 and w and n is string matching length.
3. Prioritize increasing n and decreasing u.

4. Encode positive integer n in binary, preceded by a prefix of log_n zeros. Eg: 3 is encoded as 011. 5 is encoded as 00101.
5. Encode u in binary using a fixed length code of length $log_2 w$ bits.

The decoded knows n and can simply count back by u steps in the window to find the appropriate n-tuple

6. Set the pointer to **p+n**, go to step 3 and repeat till sequence ends.

Implementation Steps

Implementation of the whole project is in 3 parts:

1. Markov Chain Generation
2. Encoding
3. Decoding

To check:

- Initial string matches output string
- $L_{min-avg}$ approaches conditional entropy of the markov chain *H(X/S)*

Sequence of steps:

1. Write the code for encoding and decoding in functional blocks.
2. Generate markov chain with known entropy and transition probabilities.
3. Make function calls for encoding, decoding
4. Check conditions

Encoding

Encoding steps in program

0. We have window, source, window power ($log_2 w$)
1. Follow steps 1,2 and 3 from Setup. We have n and u values of the matches. For programming convenience and easy debugging, I chose to first convert n and u to a decimal string and then into a binary string.
2. Before n, append the same number of zeros as the number of digits in n. And encode u preceeded by zeros such that total digits in u is 6. (In all experiments I have considered window power = 18, 2*18 has 6 decimal digits in it.) For example if (n,u) = (15,45), the decimal encoded n and u become (0015,000045)

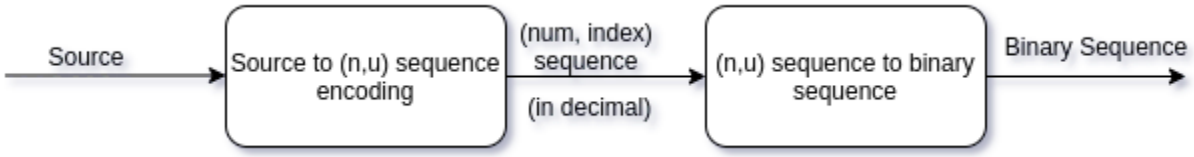
Note: number of zeros precceding n are same as digits in n in decimal

make total number of digits in u to be 6 in decimal

This format is helpful later while decoding.

3. Binary encoding of n and u (following steps 4 and 5 from Setup), n is preceeded by a prefix of $log_2 n$ zeros and u is encoded to be of $log_2 w$ bits, where w is length of window.

The block diagram below shows the steps reiterated.



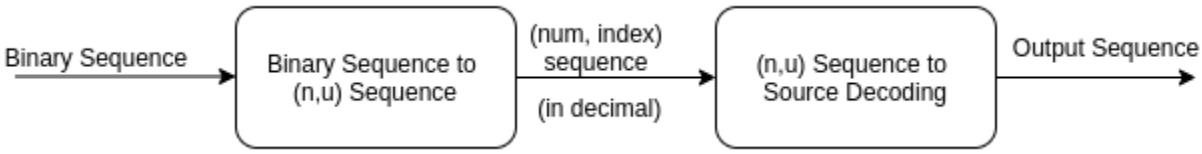
Decoding

Decoding steps in program

0. We have a binary sequence with individual n and u for every match.
1. Binary Decoding: From the start of string, count the number of zeros initially until a 1 is received. Let number of zeros be 'zeros'. Then consider the 'zeros' + 1 proceeding bits. Convert these bits to decimal to get n, follow step 2 of **Encoding**, append it to the decimal sequence. Then consider the proceeding 18 bits, convert to decimal to get u, follow step 2 of **Encoding** and append it to the sequence.
2. Decimal Decoding: We have n and u in decimal form. From start of string, count the number of zeros and consider the same number of proceeding digits to be n. Then consider the next 6 proceeding digits as u. In that case, the match string will be x_{p+1-u}^{p+n-u}, where p is the pointer where window ends. Repeat step 2 for all the matched strings and retrieve the original source sequence.

NOTE: Decimal string encoding and decoding is formally not a part of LZ77 algorithm. However, I decided to add an extra intermediate step because it helped while debugging for any problems in the code (I can read decimal faster than binary!) and overall understand the process better.

The block diagram below shows the steps reiterated.



Finally we have to check:

1. (num, index) decimal sequence in encoding matches with (num,index) decimal sequence in decoding
2. source matches with output sequence

If these two conditions are satisfied, we can be assured that our decoder works perfectly according to ur encoder.

Ofcourse we have to check if our $L_{avg-min}$ approaches conditional entropy **H(X/S)** of the Markov Source.

This will be checked after string is generated from Markov source and encoding is achieved.

In [1]:
```
###.....................Actual Program Starts Here.............................
##.....................................................................................
#...................Import useful libraries to be used.........................

import numpy as np
import math
import matplotlib.pyplot as plt

# Numpy is an independent numerical operations library in
## Python that makes dealing with array operations easy
# Math is default math operations library of Python
# matplotlib is a plotting library for graphs and visualisations
```

Defining some functions that will be used for encoding/decoding

Return binary function

Returns the binary form of input decimal integer.

In [2]:
```
def reverse(z):   ## reverses a string
    rez = ''
    for i in z:
        rez = i + rez
    return(rez)

def binary(x):
    ## Function to return binary value of decimal integer input
    z = ''
    while(x>0):
        if (x%2) == 1:
            z = z + '1'
        else:
            z = z + '0'
        x = x//2
    rez = reverse(z)
    return(rez)

print(binary(12))  ## Testing the function
```

```
1100
```

Binary encoder function for u and n (distance from p, string match length)

In [3]:
```
def encoder_nu(n,u,win_pow):

    ## win_pow is window power, earlier labeled as z
    ## window length = 2^win_pow

    bin_n = binary(n)
    bin_u = binary(u)

    unary = '0'*int(math.log(n,2))
    bin_n = unary + bin_n

    bin_u = '0'*(win_pow-len(bin_u)) + bin_u
    return(bin_n,bin_u)
    #print(bin_n)
    #print(bin_u)

bn,bu = encoder_nu(4,8,18)
## Testing the function with window power = 18
print(bn,bu)
```

```
00100 000000000000001000
```

Binary decoder function for u and n (distance from p, string match length)

In [4]:
```
## bianry decoder function
## converts binary input; returns decimal
def decoder_nu(n):
    x = 0
    i = 0
    while(n>0):
        x = x + (n%10)*(2**i)
        n = n//10
        i+=1
        #print(num,n)
    return x
print(decoder_nu(1000))  ## Testing the function
```

```
8
```

See PMF (probability mass function) of symbols and the Entropy of a given input string

In [47]:
```
def pmf_entropy(sample):
    ## input argument = sample
    pmf = {}
    ## pmf dictionary to store symbols as key;
    ## their probabilities as value
    dicto = {}
    ## just another dictionary for convenience


    for i in sample:
        ## in this block, new symbols are added to dict,
        ## along with their occurances
        try:
            dicto[i] = dicto[i]+1
        except:
            dicto[i] = 1
    Entropy = 0

    for i in dicto:
        pmf[i] = dicto[i]/len(sample)
        dicto[i] = -1*pmf[i]*(math.log(pmf[i],2))
        ## Entropy is avg of log pmfs
        Entropy = Entropy + dicto[i]
    print("Entropy = ", Entropy)
    plt.figure(figsize=(8, 8))
    plt.stem(pmf.keys(), pmf.values())
    print("Sum of pmfs should be 1; sum = ",sum(pmf.values()))
    ## to check if probabilities are correct, sum = 1

    print('pmf of symbols in string:')
```

Intermediate step, converts integer n and u to string, to form (n,u) encoded string sequence, this maybe skipped and integer may be converted directly to binary, but I included this step to help with debugging and understand the problem better

In [6]:
```
### string encode n and u  (num and index)
### int >> string
def string_encode_nu(num,index):
    n = str(num)
    u = str(index)
    n = '0'*len(n)+n
    u = '0'*(6-len(u))+u
    ## 2^18 (max window size) has a decimal length of 6
    return n,u

n,u = string_encode_nu(23,234)

print(n,u)
print(type(n))
```

```
0023 000234
<class 'str'>
```

Encoding begins

Function for string sliding and matching with window, generating output string with n and u values (steps 2 and 3)

In [7]:
```
def source_to_sequence_encode(window,source):
    ## arguments are window and source
    L = len(source)
    output_string = ''
    p = 0
    rev_window = window[::-1]
    ## matching from reversed window to minimize u
    while(L>0):
        num = 0
        indicator = 0
        index= 0
        prev_index = 0

        while(indicator != 1):
            match = source[p:p+1+num]
            ## match = matching string
            match = match[::-1]
            ## reverse match
            index = rev_window.find(match)
            ## default function to give index of a string in larger string,
            ## returns value of -1 of no match is found

            if (index != -1):       ## condition for match
                L = L-1             ##
                num = num+1         ## increase string by 1 and try to match
                prev_index = index
                if (L<1):           ## if length of string is less than 1,
                    ##                 break out of loop, string encoding done

                    n,u = string_encode_nu(num,index)
                    ## n and u in string form

                    output_string = output_string+n+u
                    ## append n and u to output_string
                    break

            else:               ## condition for no match
                indicator = 1   ## means matching is done,break out of loop,
                                ## move on to next match
                p = p+num
                n,u = string_encode_nu(num,prev_index)
                output_string = output_string+n+u
    return(output_string)       ## returns all n and u in string form


## trying out the function with window and source
## from qsn 2.33 of book(Principles of Digital Communication) page 61
window = '00011101'
source = '0010101100'
output_string = source_to_sequence_encode(window,source)
print(output_string)
```

```
0300000402000000030000030200000 5
```

Binary Encoder, takes output string as argument, returns binary code (steps 4 and 5)

In [8]:
```
## binary encoder

###..output_string >>>> compressed binary code
def binary_encoder_sequence(output_string,win_pow): ## win_pow is "window power"

    code = output_string
    binary_sequence = ''
    L = len(code)
    n = 0
    zeros = 0
    while(L>0):
        if code[n] == '0':
            zeros+=1  ## count the number of zeros
            n+=1
        else:
            num = int(code[n:n+zeros])
            ## take n as the number of digits as number of zeros
            ##                                        preceeding it

            index = int(code[n+zeros:n+zeros+6])
            ## take u  as the next 6 digits in the string

            bin_n,bin_u = encoder_nu(num,index,win_pow)
            ##binarize n and u

            sequence = bin_n + bin_u
            binary_sequence = binary_sequence + sequence
            code = code[n+zeros+6:]
            n = 0
            zeros = 0

        if (len(code)<1):
            break
    type(binary_sequence)
    return(binary_sequence)

## trying out the fuction for the previous achieved output_string
binary_sequence = binary_encoder_sequence(output_string,8)
                  ## take win_pow = 8 as given in question
print(binary_sequence)
```

```
01100000100010000000000110000001101000000101
```

Decoder

Binary Decoder, takes binary sequence as argument, returns (n,u)decimal string (As discussed in programming steps)

In [9]:

```
## binary decoder
##binary sequence >> output_string or retrieved sequence

def binary_decoder_sequence(binary_sequence, win_pow):
    retrieved_sequence = ''
    code = binary_sequence
    i = 0
    zeros = 0
    L = len(code)
    while (L>0):
        if code[i] == '0':
            zeros+=1 ## counting the number of zeros
            i+=1
        else:
            num = int(code[i:i+zeros+1])
            ## take n as the number of digits +1
            ## as number of zeros preceeding it

            index = int(code[i+zeros+1:i+zeros+1+win_pow])
            ## u is simply the next (window power) number of digits

            num = decoder_nu(num)
            index = decoder_nu(index)## convert to decimal

            num,index = string_encode_nu(num,index)## convert to string
            retrieved_sequence = retrieved_sequence + num + index
            ## append to the sequence

            code = code[i+zeros+1+win_pow:]
            i = 0
            zeros = 0
        if (len(code)<1):
            break


    return(retrieved_sequence)


## Trying out the function with the binary sequence
## containing n and u in binary

retrieved_sequence = binary_decoder_sequence(binary_sequence, 8)
## win_pow = 8 as per question

print(retrieved_sequence)

if retrieved_sequence == output_string:
    print('Matched !!')
```

```
0300000402000000030000030200000 5
Matched !!
```

In [10]:
```
## string decoder from code

## string decoding from nu code
##.....output_string or retrieved sequence >> decode_sequence
## decode sequence match with source


def sequence_to_source_decode(retrieved_sequence):
    code = retrieved_sequence
    rev_wind = window[::-1]
    decode_sequence = ''
    #print(rev_wind)
    L = len(code)
    n = 0
    zeros = 0
    while(L>0):
        #zeros = 0
        #n = 0
        if code[n] == '0':
            zeros+=1 ## counting the number of zeros
            n+=1
        else:
            #print(n, zeros)
            num = int(code[n:n+zeros])

            ## take n as the number of digits as number of zeros
            ## preceeding it

            index = int(code[n+zeros:n+zeros+6])
            ## take u  as the next 6 digits in the string

            sequence = rev_wind[index:index+num]
            ## finding out the matched string

            decode_sequence = decode_sequence + sequence[::-1]
            ## append matched string in final result

            code = code[n+zeros+6:]
            n = 0
            zeros = 0

        if (len(code)<1):
            break

    return(decode_sequence)

## trying out function with retrieved_sequence string
##containing n and u in decimal

decode_sequence = sequence_to_source_decode(retrieved_sequence)
print(decode_sequence)

if decode_sequence == source:
    print('Matched !!')
```

```
0010101100
Matched !!
```

Creating a Markov source

A finite state Markov chain is a sequence S0, S1... of discrete chance variables from a finite alphabet S where q(s) is a pmf on S. The notation q(s) is also called **stationary pmf**, denoting the probabilities of symbols when the length of source string tends to be very large. Essentially we can say that, in a Markov chain, the probability of occurance of a symbol depends on the previous state. Then a transition matrix is required to represent those probabilities.

Let a Markov chain generate 3 symbols 'a','b' and 'c'. Their inital occurance probability can be denoted by a row vector B. Then we need to define a transition matrix A:

$$\begin{bmatrix} A & a & b & c \\ a & 0.6 & 0.3 & 0.1 \\ b & 0 & 0.7 & 0.3 \\ c & 0.2 & 0.3 & 0.5 \end{bmatrix}$$

This matrix shows transition probabilities. Probability of getting an output of 'a' while previous state was 'a' is 0.6. Similarly, probability of getting an output of 'b' while previous state was 'c' is 0.3.

Conditional probability can be defined as P(x/s) where X is output and S is previous state.

P(x/s) = probability of getting x when previous state was s = transition probability

q(s) = probability of a state s = previous probability * transition probability

When the output string length is very large, q(s) approaches a stationary value, hence also called **stationary probability**. Denoted in code as stat_b. Can be achieved by repeated multiplication of B.A and storing the result in B

Entropy

After a string sequence is generated, average of log pmfs of the symbols gives the Entropy **H(X)**, where X is the random variable denoting symbol. H(X) shows uncertainty of a symbol.

Conditional Entropy

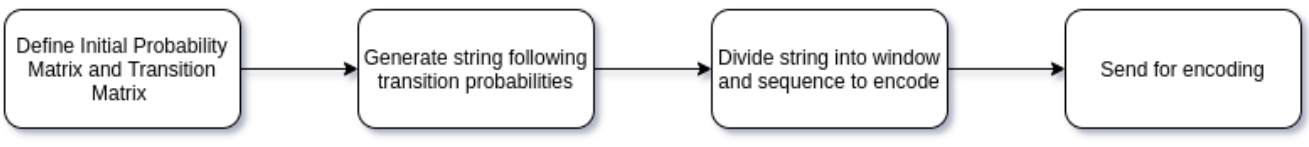
Denoted by **H(X/S)** where X is the symbol and S is the previous state, showing uncertainity of a symbol given a previous state is known. Thus theoretically H(X/S) is less than or equal to H(X).

Formuala for H(X/S) is given by:

$$\left(\sum_{x} \sum_{s} q(s).\, p(x/s) log_2 \frac{1}{p(x/s)} \right)$$

where **q(s)** denotes stationary probability of a symbol and **p(x/s)** denotes transition probability from state 's' to symbol 'x'.

The steps followed further are shown in the flow-chart below



In [67]:
```
n = 5
## user defined transition matrix
a = np.array([[0.4,0.2,0.3,0.1,0],
              [0,0.6,0.1,0.2,0.1],
              [0.2,0.2,0.5,0.1,0],
              [0.2,0.2,0,0.5,0.1],
              [0,0,0.3,0.1,0.6]])
print('A = ',a)
## user defined initial probability matrix
b = np.array([0.2,0.2,0.3,0.2,0.1])
print('B = ',b)
```

```
A =  [[0.4 0.2 0.3 0.1 0. ]
 [0.  0.6 0.1 0.2 0.1]
 [0.2 0.2 0.5 0.1 0. ]
 [0.2 0.2 0.  0.5 0.1]
 [0.  0.  0.3 0.1 0.6]]
B =  [0.2 0.2 0.3 0.2 0.1]
```

In [68]:
```
stat_b = b
for i in range(50):
    stat_b = np.matmul(stat_b,a)
print(stat_b)### stationary probabilities
sum(stat_b)  ### check if sum of stationary probabilities is 1
```

```
[0.14556962 0.29113924 0.22151899 0.21518987 0.12658228]
```

Out[68]:
```
1.0000000000000002
```

In [69]:
```
source = []
for i in range(97,97+n):
    x = chr(i)
    source.append(x)
source = np.array(source)
source  ## taking characters of english language as source symbols
```

Out[69]:
```
array(['a', 'b', 'c', 'd', 'e'], dtype='<U1')
```

In [70]:
```
string = '' ## character generation from defined markov chain
string = np.random.choice(source, p = b)
ind = np.where(source == string)[0][0]
for i in range(1000000): ## taking 1000,000 characters as a string
    char = np.random.choice(source, p = a[ind])
    ind = np.where(source == char)[0][0]
    string = string + char
```

In [71]:
```
sample = string
len(string)
```

Out[71]:
```
1000001
```

In the next block we check probabilities of the symbols in string, essentially their pmf. Compare it with stationary probabilities, they should match!

In [72]:
```
### matching probabilities:
pmf = {}
for i in string:
    try:
        pmf[i] = pmf[i] + 1
    except:
        pmf[i] = 1
L = len(string)

for i in pmf:
    pmf[i] = pmf[i]/L


print(pmf)
print(stat_b)
```

```
{'b': 0.290986709013291, 'e': 0.12581687418312582, 'c': 0.22166277833722167,
'a': 0.14612785387214614, 'd': 0.21540578459421542}
[0.14556962 0.29113924 0.22151899 0.21518987 0.12658228]
```

probability of 'a' in string: 0.146127

stationary probability of 'a': 0.145569

Matching to 3 places after decimal!!

In [73]:
```
### Entropy given earlier is avg of lof pmfs
### entropy should be less than 3                 H(X)

entropy = pmf

for i in entropy:
    entropy[i] = -entropy[i]*math.log(entropy[i],2)
Entropy = sum(entropy.values())
Entropy
```

Out[73]:
```
2.258862530697833
```

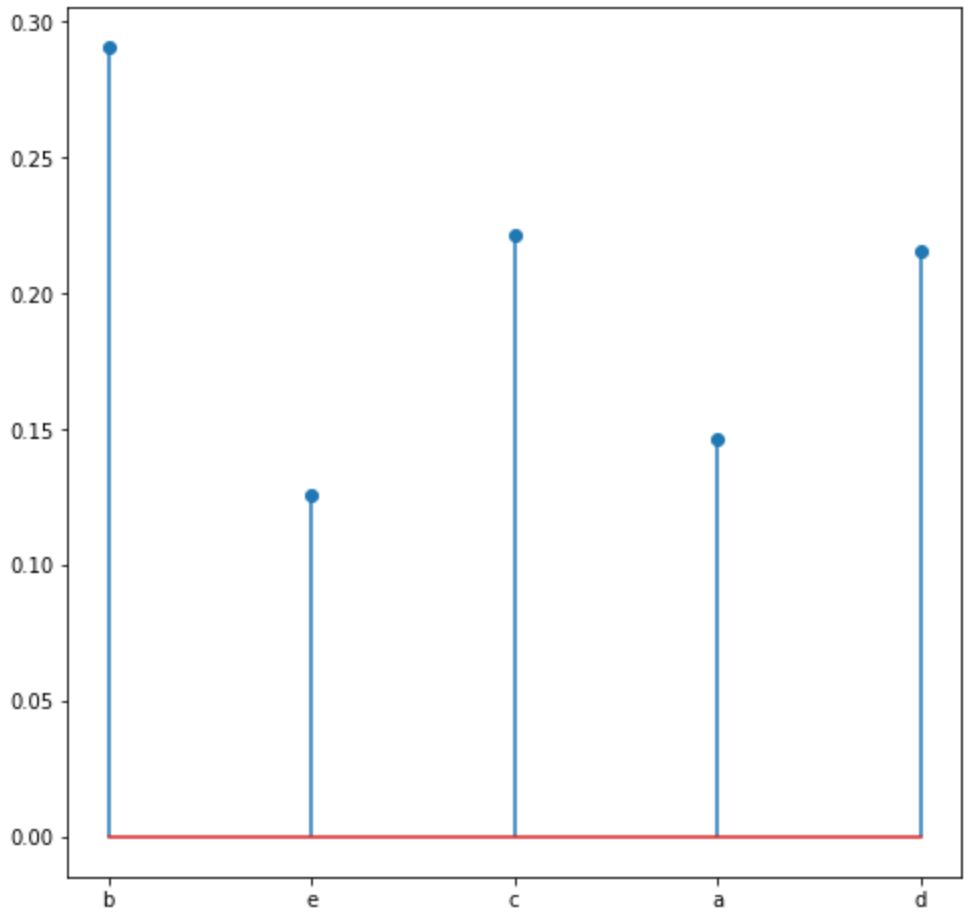
In [74]:
```
## conditional entropy H(X/S)  Found using formula given above
Entropy = 0
for i in range(n):
    for j in range(n):
        try:
            ent = -stat_b[i]*a[i,j]*math.log(a[i,j],2)
        except:
            ent = 0
        Entropy = Entropy + ent
Entropy

## Proved H(X/S) <= H(X) !!
```

Out[74]:
```
1.6591619520932208
```

In [75]:
```
pmf_entropy(sample)          ### see pmf of source
```

```
Entropy =  2.258862530697833
Sum of pmfs should be 1; sum =  1.0
pmf of symbols in string:
```



In [76]:
```
### define window and source

#####.....................USE WITH INDIVIDUAL SOURCE.....................
## test

win_pow = 18
p = 2**win_pow          # starting point
window = sample[0:p]    # window

source = sample[p:]     # symbols to code
L = len(source)         # length of symbols to code
places = {}             ## match dictionary, key = u; value = n

print('Length of source string =', L, 'p =' , p)

## source is the string we need to encode
```

```
Length of source string = 737857 p = 262144
```

Encoder

source >> output_string

output_string >> binary_sequence

Decoder

binary_sequence >> retrieved_sequence

retrieved_sequence >> decode_sequence

In [77]:
```
output_string = source_to_sequence_encode(window,source)## calling previous funct
```

In [78]:
```
binary_sequence = binary_encoder_sequence(output_string,win_pow)
```

In [79]:
```
retrieved_sequence = binary_decoder_sequence(binary_sequence, win_pow)
```

In [80]:
```
decode_sequence = sequence_to_source_decode(retrieved_sequence)
```

In [81]:
```
if retrieved_sequence == output_string:
    print('Matched')
```

```
Matched
```

In [82]:
```
if decode_sequence == source:
    print('Matched, input string and output string are matching')
    print('Means coding and decoding are correct')
```

```
Matched, input string and output string are matching; Means coding and decoding
are correct
```

In [84]:
```
l = len(binary_sequence) + len(window)*3
ls = 1000000
L_avg = l/ls
L_avg
```

Out[84]:
```
2.520148
```

L_avg is 2.77, which is less than 3, So compression is happening.

But numerically it is far from the conditional entropy value of 1.6.

Running the experiment again for 3 states instead of 5

In [60]:
```
n = 3
## user defined transition matrix
a = np.array([[0.6,0.3,0.1],[0,0.7,0.3],[0.2,0.3,0.5]])
print('A = ',a)
## user defined initial probability matrix
b = np.array([0.3,0.4,0.3])
print('B = ',b)

stat_b = b
for i in range(50):
    stat_b = np.matmul(stat_b,a)

source = []
for i in range(97,97+n):
    x = chr(i)
    source.append(x)
source = np.array(source)

string = ''                    ## character generation from defined markov chain
string = np.random.choice(source, p = b)
ind = np.where(source == string)[0][0]
for i in range(1000000):       ## taking 1000,000 characters as a string
    char = np.random.choice(source, p = a[ind])
    ind = np.where(source == char)[0][0]
    string = string + char

sample = string
len(sample)
```

```
A =  [[0.6 0.3 0.1]
 [0.  0.7 0.3]
 [0.2 0.3 0.5]]
B =  [0.3 0.4 0.3]
```

Out[60]:
```
1000001
```

In [61]:
```
### Entropy given earlier is avg of lof pmfs
### entropy should be less than 2                  H(X)

pmf = {}
for i in string:
    try:
        pmf[i] = pmf[i] + 1
    except:
        pmf[i] = 1
L = len(string)

for i in pmf:
    pmf[i] = pmf[i]/L

entropy = pmf

for i in entropy:
    entropy[i] = -entropy[i]*math.log(entropy[i],2)
Entropy = sum(entropy.values())
Entropy
```

Out[61]:
```
1.4583953679549222
```

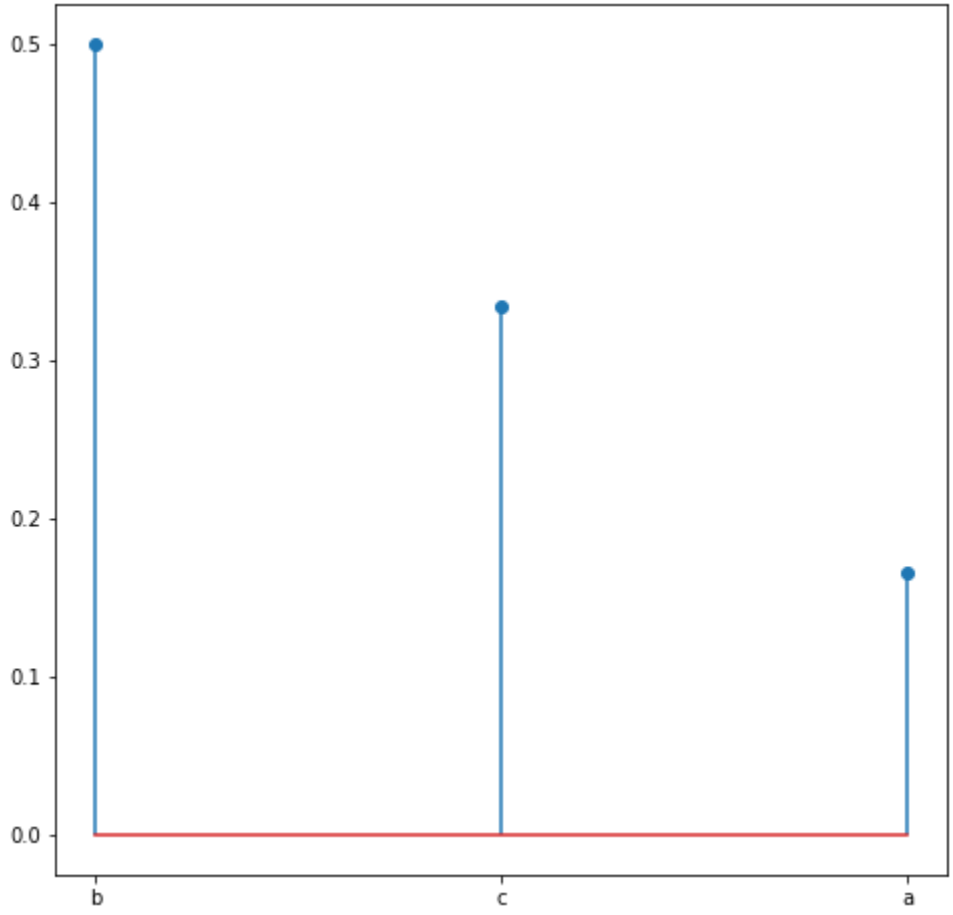
In [62]:
```
## conditional entropy H(X/S)  Found using formula given above
Entropy = 0
for i in range(n):
    for j in range(n):
        try:
            ent = -stat_b[i]*a[i,j]*math.log(a[i,j],2)
        except:
            ent = 0
        Entropy = Entropy + ent
Entropy

## Proved H(X/S) <= H(X) !!
```

Out[62]:
```
1.1517141893975102
```

In [63]:
```
pmf_entropy(sample)          ### see pmf of source
```

```
Entropy =  1.4583953679549222
Sum of pmfs should be 1; sum =  1.0
pmf of symbols in string:
```



In [64]:
```
### define window and source

#####.....................USE WITH INDIVIDUAL SOURCE.....................
## test

win_pow = 18
p = 2**win_pow          # starting point
window = sample[0:p]    # window

source = sample[p:]     # symbols to code
L = len(source)         # length of symbols to code
places = {}             ## match dictionary, key = u; value = n

print('Length of source string =', L, 'p =' , p)

## source is the string we need to encode
```

```
Length of source string = 737857 p = 262144
```

In [86]:
```
output_string = source_to_sequence_encode(window,source)## calling previous funct
binary_sequence = binary_encoder_sequence(output_string,win_pow)
retrieved_sequence = binary_decoder_sequence(binary_sequence, win_pow)
decode_sequence = sequence_to_source_decode(retrieved_sequence)

if retrieved_sequence == output_string:
    print('Decimal string containing n and u Matched')

if decode_sequence == source:
    print('Matched, input string and output string are matching')
    print('Means coding and decoding are correct')
```

```
Decimal string containing n and u Matched
Matched, input string and output string are matching
Means coding and decoding are correct
```

In [66]:
```
l = len(binary_sequence) + len(window)*2
ls = 1000000
L_avg = l/ls
L_avg
```

Out[66]:
```
1.747668
```

Compression is happening but numerically far from conditional entropy value of 1.1.

REFERENCES

1. Robert Gallager, course materials for 6.450 Principles of Digital Communications I, Fall 2006. MIT OpenCourseWare(http://ocw.mit.edu/ (http://ocw.mit.edu/)), Massachusetts Institute of Technology.
2. https://numpy.org/doc/stable/ (https://numpy.org/doc/stable/)