4 Huffman coding

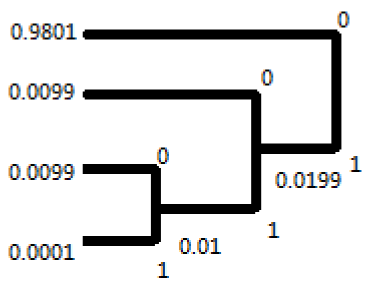
1. Consider a binary DMS X, with probabilities 1-10^-2 and 10^-2.

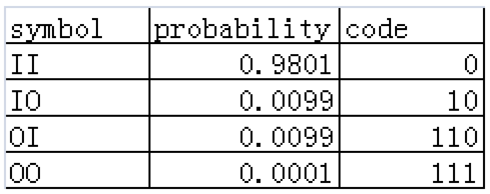
(a) (1 mark) what is the entropy H(X) of X?

$$P(X = 1) = 0.99, P(X = 0) = 0.01$$

$$H(X) = -0.99 * \log_2 0.99 - 0.01 * \log_2 0.01 = 0.0808$$

(b) (2 marks) derive a second extension binary Huffman code for this source. Determine the average number R of bits per source letter. Is your code optimal?





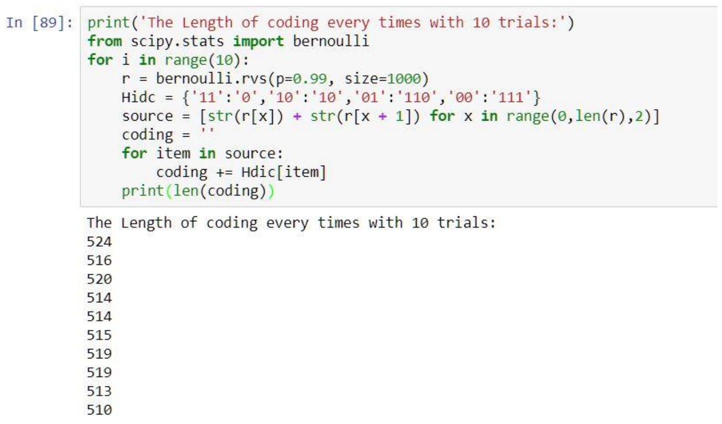
symbol	probability	code
II	0.9801	0
IO	0.0099	10
OI	0.0099	110
OO	0.0001	111

The average number of bits per source letter:

$$R = 0.9801 * \left(\frac{1}{2}\right) + 0.0099 * \left(\frac{2}{2}\right) + 0.0099 * \left(\frac{3}{2}\right) + 0.0001 * \left(\frac{3}{2}\right) = 0.515$$

Compared to the optimal rate 0.081, my code is not optimal.

(c) (11 marks) generate a binary sequence x1 of length 1000 in MATLAB that is output by this source (use the specified probabilities to generate the bits). Write MATLAB programs to encode and decode your sequence, using your Huffman coder of part (b). Determine the number R(x1) of bits per source letter and compare with (b). Do you get compression, expansion or neither? Can you think of a more efficient way to compress your binary sequence?



```
In [89]: print('The Length of coding every times with 10 trials:')
         from scipy.stats import bernoulli
         for i in range(10):
             r = bernoulli.rvs(p=0.99, size=1000)
             Hidc = {'11':'0','10':'10','01':'110','00':'111'}
             source = [str(r[x]) + str(r[x + 1]) for x in range(0,len(r),2)]
             coding = ''
             for item in source:
                 coding += Hdic[item]
             print(len(coding))
```

```
The Length of coding every times with 10 trials:
524
516
520
514
514
515
519
519
513
510
```

Huffman coding length with p=0.99

The code gives different Huffman code lengths every time, which depends on the exact random sequence given by the binary distribution.

Take the third time of experiment as an example, the number of bits per source letter R(x1)=520/1000 = 0.520, which is the compression of the source sequence.

Since the rate 0.527 is still far greater than the optimal rate 0.08, we can increase the symbol length in the Huffman method to achieve a more efficient compression.

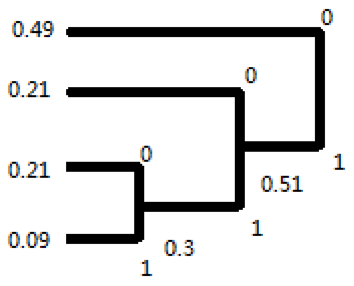
2. Repeat (a)-(c) for a binary DMS Y, with probabilities 0.7 and 0.3.

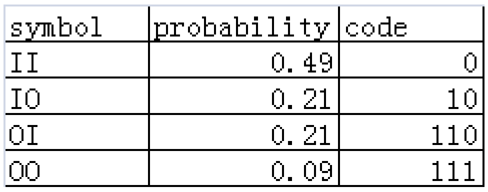
(i) What is the entropy H(Y) of Y?

$$\mathrm{P}(\mathrm{X}=1)=0.7, \mathrm{P}(\mathrm{X}=0)=0.3$$

$$\mathrm{H}(\mathrm{X})=-0.7*\log_2 0.7-0.3*\log_2 0.3=0.8813$$

(ii) Derive a second extension binary Huffman code for this source. Determine the average number R of bits per source letter. Is your code optimal?





symbol	probability	code
II	0.49	0
IO	0.21	10
OI	0.21	110
OO	0.09	111

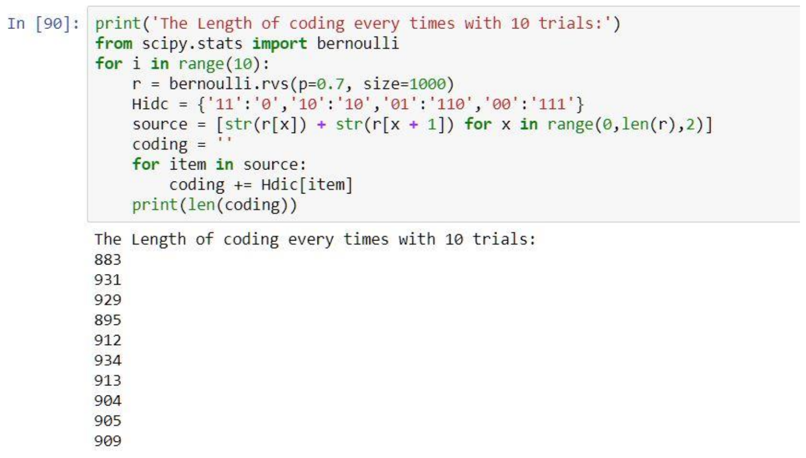
The average number of bits per source letter:

$$\mathrm{R}=0.49*\left(\frac{1}{2}\right)+0.21*\left(\frac{2}{2}\right)+0.21*\left(\frac{3}{2}\right)+0.09*(\frac{3}{2})=0.905$$

Compared to the optimal rate 0.88, my code is not optimal.

(iii) (11 marks) generate a binary sequence y1 of length 1000 in MATLAB that is output by this source (use the specified probabilities to generate the bits). Write MATLAB programs to encode and decode your sequence, using your Huffman coder of part (b). Determine the number R(y1) of bits per source letter and compare with (b). Do you get compression, expansion or neither? Can you think of a more efficient way to compress your binary sequence?

The code gives different Huffman code lengths every time, which depends on the exact random sequence given by the binary distribution.



```
In [90]: print('The Length of coding every times with 10 trials:')
         from scipy.stats import bernoulli
         for i in range(10):
             r = bernoulli.rvs(p=0.7, size=1000)
             Hidc = {'11':'0','10':'10','01':'110','00':'111'}
             source = [str(r[x]) + str(r[x + 1]) for x in range(0,len(r),2)]
             coding = ''
             for item in source:
                 coding += Hdic[item]
             print(len(coding))
```

```
The Length of coding every times with 10 trials:
883
931
929
895
912
934
913
904
905
909
```

Huffman coding length with p=0.7

Take the third time of experiment as an example, the number of bits per source letter

R(x1)=929/1000 = 0.929, which is the compression of the source sequence.
Since the rate 0.929 is still slightly greater than the optimal rate 0.88, we can increase the symbol length in the Huffman method to achieve a more efficient compression.

5 Lempel-Ziv coding & arithmetic coding

1. In this question we focus on LZ78, which is explained in the textbook.

(a) (2 marks) Explain in your own words how LZ78 works. Use any reference material you can find (the textbook, other books, web, scientific papers, survey papers...); make sure that you refer to your references in the text, for this include a References section. How to do this? Read "ieeecitationref.pdf", see LMS-this workshop. Do not use word-for-word citations (paraphrasing in your own words is ok).

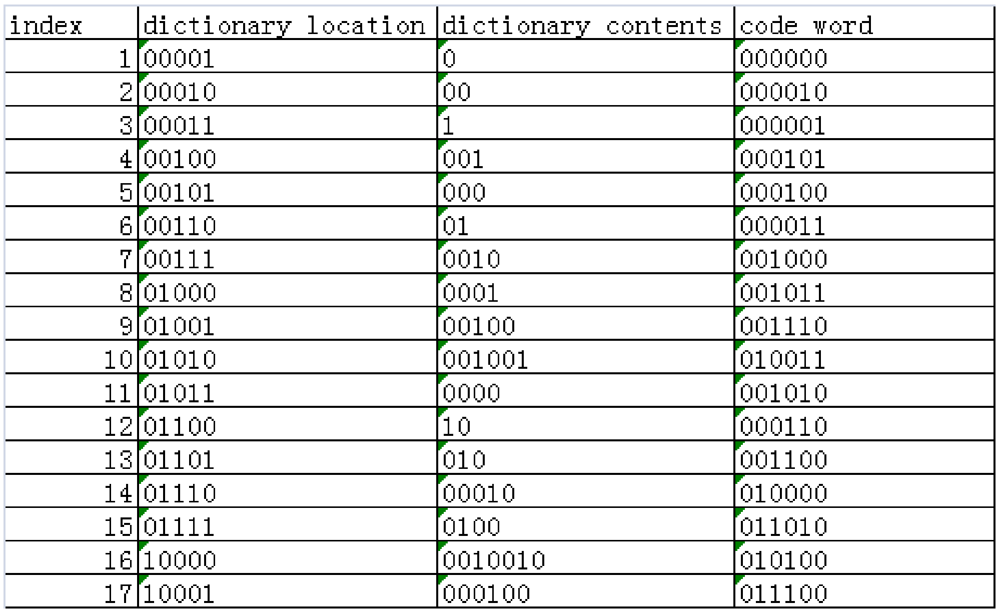
In LZ78, the sequence from the source output is divided into many phrases with different length. A new phrase is introduced when the encoder detects a new combination of symbols. Usually the new phrase is composed of an existing phrase with a new symbol appended to the end. Subsequently, a dictionary is built based on the order in which new phrases occur. The first phase is located in the first position in the dictionary, denoted by the address of 0000, and then the second phrase is located in the second address, which is 0001, and so on and so fore. The code word of each phrase is defined as the location of the existing phrase with the last symbol of the phrase appended to the end of that location. At the very beginning, there is no existing phrase in the dictionary, so the location of the existing phrase is defined as 0000. However, this method will no doubt lead to the overflow in the dictionary. Hence, we need to discard the useless items.

(b) (1 mark) Recall that there are 4 types of compression, namely Type I, Type II, Type III and Type IV as explained in section 1. What type is LZ78?
Since the phrase length is varying, but the code word length is not, the LZ78 is (Type III) variable-to-fixed source coding

(c) (3 marks) By using LZ78, compress the following binary source sequence (00010010 repeatedly):

0001001000010010000100100001001000010010000100100001001000010010000100

Parsing the sequence:
0, 00, 1, 001, 000, 01, 0010, 0001, 00100, 001001, 0000, 10, 010, 00010, 0100, 0010010, 000100,



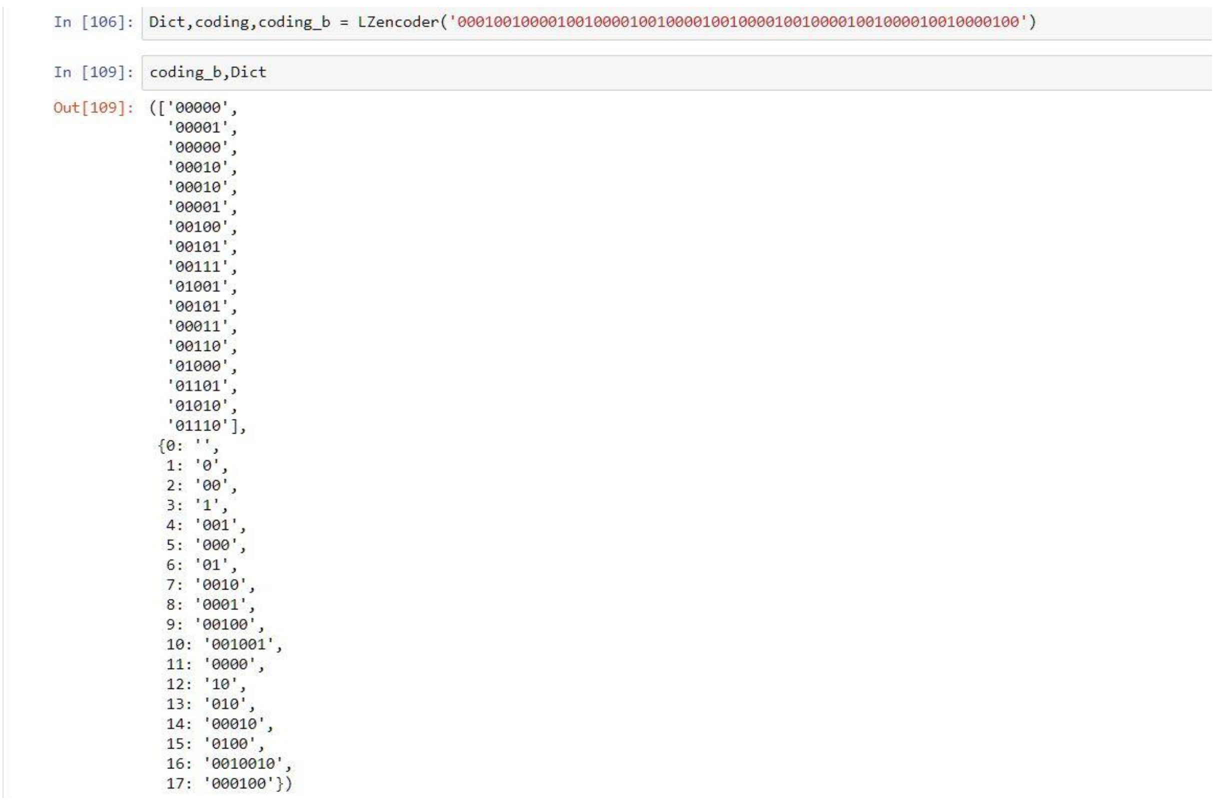
index	dictionary location	dictionary contents	code word
1	00001	0	000000
2	00010	00	000010
3	00011	1	000001
4	00100	001	000101
5	00101	000	000100
6	00110	01	000011
7	00111	0010	001000
8	01000	0001	001011
9	01001	00100	001110
10	01010	001001	010011
11	01011	0000	001010
12	01100	10	000110
13	01101	010	001100
14	01110	00010	010000
15	01111	0100	011010
16	10000	0010010	010100
17	10001	000100	011100

The encoded sequence is (the spaces are only for the visual effect):

000000 000010 000001 000101 000100 000011 001000 001011 001110 010011
001010 000110 001100 010000 011010 010100 011100

(d) (7 marks) Write a MATLAB program for an LZ78 encoder; also write a MATLAB program for the corresponding decoder. Make sure that you validate your programs, using the sequence from part (c).

The python output are shown below and scripts can be found in appendix

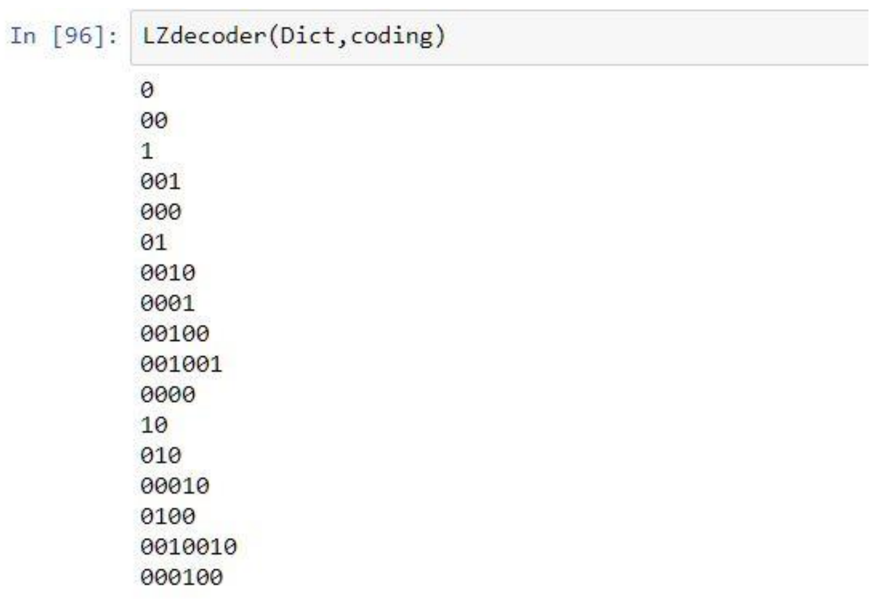


```
In [106]: Dict,coding,coding_b = LZencoder('00010010000100100001001000010010000100100001001000010010000100')

In [109]: coding_b,Dict

Out[109]: (['00000',
            '00001',
            '00000',
            '00010',
            '00010',
            '00001',
            '00100',
            '00101',
            '00111',
            '01001',
            '00101',
            '00011',
            '00110',
            '01000',
            '01101',
            '01010',
            '01110'],
           {0: '',
            1: '0',
            2: '00',
            3: '1',
            4: '001',
            5: '000',
            6: '01',
            7: '0010',
            8: '0001',
            9: '00100',
            10: '001001',
            11: '0000',
            12: '10',
            13: '010',
            14: '00010',
            15: '0100',
            16: '0010010',
            17: '000100'})
```

LZ78 Encoding



```
In [96]: LZdecoder(Dict,coding)
```

```
0
00
1
001
000
01
0010
0001
00100
001001
0000
10
010
00010
0100
0010010
000100
```

LZ78 Decoding

2. (11 marks) LZ coding is a "stream coding" method. Another type of stream coding is "arithmetic coding". It is widely used in a range of applications; see the LMS additional material "WangOstermannZhang pages 234-241". Another good description that you may find helpful is in the book by David McKay "Information Theory, Inference, and LearningAlgorithms", 2005 edition:

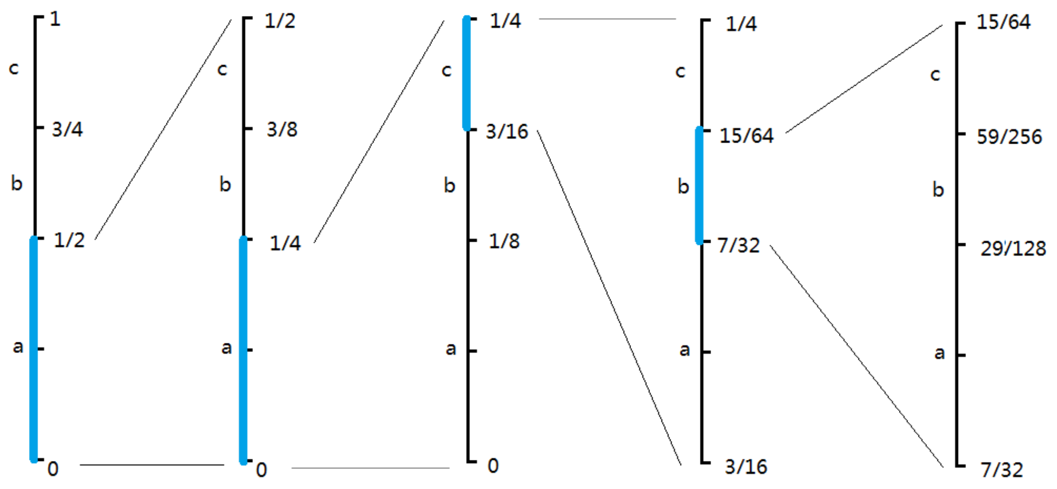
(a) (1 mark) Recall that there are 4 types of compression, namely Type I, Type II, Type III and Type IV as explained in section 1. What type is arithmetic coding?
Since the arithmetic coding converts symbols with a variable length of into code words with a variable length, it belongs to (Type IV) variable-to-variable source coding.

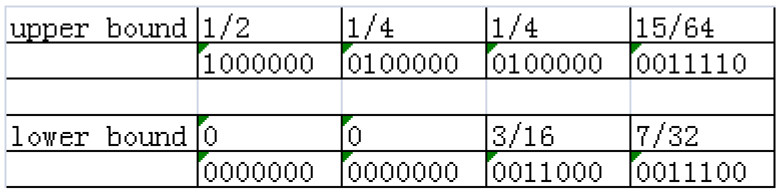
(b) (5 marks) Explain in your own words how arithmetic coding works. For this, use any reference material you can find (the textbook, other books, web, scientific papers, survey papers...); make sure that you refer to your references in the text, for this include a references section. How to do this? Read "ieeecitationref.pdf", see LMS-this workshop. Do not use word-for-word citations (paraphrasing in your own words is ok).

The arithmetic coding represents a source sequence with a segment in the interval from zero to one. The greater possibility a sequence has, the bigger the segment is. Firstly, all the single symbols from the source are assigned to a segment of the [0, 1] interval according to their

possibilities. The upper bound and the lower bound are represented by the inverse binary code, where abcdef are corresponded to a decimal number of a*(1/2) + b*(1/4) + c*(1/8)....
Starting from the right, the same bits in the upper bound and the lower bound are transmitted, until we meet the first different bits.
On the receiver side, we can determine the upper bound and the lower bound from the coded sequence, because the coded sequence is the lower bound value, while that sequence plus one unit of the least significant bit is the upper bound. Since we can figure out the bounds, we can decipher the coded sequence according to the relation between the segment in the [0,1] interval and the corresponding source sequence mentioned above.

(c) (5 marks) consider the Example 8.4 in "WangOstermannZhang pages 234-241". Explain in detail how "a a c b" is encoded by an arithmetic encoder.





upper bound	1/2	1/4	1/4	15/64
	1000000	0100000	0100000	0011110
lower bound	0	0	3/16	7/32
	0000000	0000000	0011000	0011100

First and foremost, we assign the segment of [0, 1/2] to the letter ‘a’, the segment [1/2, 3/4] to the letter ‘b’, and the segment [3/4, 1] to the letter ‘c’.
When the first letter ‘a’ comes, we can determine the upper bound to be 1/2, the lower bound to be 0, which correspond to the inverse binary number 1000000 and 0000000 respectively.
When the second letter ‘a’ comes, we can decide the upper bound to be 1/4, the lower bound to be 0, which correspond to the inverse binary number 0100000 and 00000000 respectively.
When the third letter ‘c’ comes, we can decide the upper bound to be 1/4, the lower bound to be 3/16, which correspond to the inverse binary number 0100000 and 0011000 respectively.
When the forth letter ‘b’ comes, we can decide the upper bound to be 15/64, the lower bound to be 7/32, which correspond to the inverse binary number 0011110 and 0011100 respectively.
Comparing the final upper bound and lower bound, the same bits from the right are 00111. Hence, the coded sequence is 00111.
To check if it is correct, we decipher the coded sequence. On the receiver side, we get the first bit as 0, which means the first interval is [0, 1) in binary, or [0, 1/2) in decimal. This shows that the

first letter is ‘a’.
We get the second bit as 0, which means the second interval is [00, 01) in binary, or [0, 1/4) in decimal. This shows that the sequence is ‘aa’.
We get the third bit as 1, which means the third interval is [001, 010) in binary, or [1/8, 1/4) in decimal. This shows that the sequence is ‘aab’ or ‘aac’.
We get the forth bit as 1, which means the third interval is [0011, 0100) in binary, or [3/16, 1/4) in decimal. This shows that the sequence is ‘aac’.
We get the fifth bit as 1, which means the third interval is [00111, 01000) in binary, or [7/32, 1/4) in decimal. This shows that the sequence is ‘aacb’ or ‘aacc.

6 Reflection and comparison (12 marks)

Explain the differences between LZ, Huffman and arithmetic coding, using any reference material you can find (the textbook, other books, web, scientific papers, survey papers...; make sure that you refer to your references in the text, for this include a References section. How to do this? Read "ieeecitationref.pdf", see LMS-this workshop. Do not use word-for-word citations (paraphrasing in your own words is ok). Particularly pay attention to:

1. In which situation would one type of coding be preferred over another? (6 comparisons)

We choose Huffman coding over LZ coding when the statistics are known, and we require the decoder to be instantaneous.
We choose Huffman coding over arithmetic coding when we require the decoder to be instantaneous.
We choose LZ coding over Huffman when the statistics are unknown, and the source sequence is long enough.
We choose LZ coding over arithmetic coding for the same reason as over the Huffman coding.
We choose arithmetic coding over Huffman coding when we require a better rate, or we are handling sources with memory, or we are handling time-varying source statistics.
We choose arithmetic coding over LZ when the source sequence is not very long, or the memory for dictionaries is limited.

2. Where is LZ coding used in practice?

The LZ coding is universally adopted when compressing or decompressing the computer files, especially under the UNIX operating system and the MS-DOS operating system.

3. Where is arithmetic coding used in practice?

Compression of VLSI test data.

4. Where is Huffman coding used in practice?

Huffman coding is used in the lossy compression of still images (JPEG).

Reference:
[1] J.G. Proakis and M. Salehi, “Digital Communications”, 4th ed, McGraw Hill 2008, pp. 106-108
[2] Y. Wang J. Ostermann and Y. Zhang, “Video Processing and Communications”, Prentice-Hall, 2002. Section 8.4 “Binary Encoding”, pp 234-241.

Appendix

Huffman coding and decoding

```
from scipy.stats import bernoulli
def huffman(p = 0.99):
    print('For p = {},the result of 10 times trials are: '.format(p))
    for i in range(10):
        r = bernoulli.rvs(p, size=1000)
        # Get the distribution r with 1000 values

        key = ['11','10','01','00']
        value = ['0','10','110','111']
        Hdic = dict(zip(key,value))
        # store the source coding

        # Creat the source by every two letters for encoding and decoding
        source = [str(r[x]) + str(r[x + 1]) for x in range(0,len(r),2)]
        coding = ''
        for item in source:
            coding += Hdic[item]
        print('The length of coding in {0}th trilas is 
{1}'.format(i+1,len(coding)))

        '''
        Decoding
        '''
        Hdic_reverse = dict(zip(value,key))

        # change the value and keys for easier looking up
        curStr = ''
        decodedStr = ''
        for item in coding:
            curStr += item
            if curStr in Hdic_reverse.keys():
                decodedStr += Hdic_reverse[curStr]
                curStr = ''


        if ''.join(source) == decodedStr:
            print('Decoding Successfully' )
        # check whether the decoded string is the same as source
```

```
if __name__=='__main__':
    huffman(p=0.99)

    huffman(p=0.7)
```

LZ coding & decoding

```
def LZencoder(info):
    table = dict()
    table[''] = 0

    curStr = ''
    coding = []
    coding_b = []
    # Creat the Dictionary for the Lz78
    # The coding_b is binary format and coding is decimal format
    for item in info:
        curStr += item
        if curStr not in table.keys():
            table[curStr] = len(table)
            ss = '{0:b}'.format(table[curStr[0:len(curStr)-1]])
            s = str(ss)
            s = s.zfill(5)
            # Add pre zeors to make the codeward be fixed 5 bits
            coding_b.append(s)

            coding.append(str(table[curStr[0:len(curStr)-1]])+','+i
tem)
            curStr = ''

    table_index = {v: k for k, v in table.items()}
  # change the values and keys for easier decoding
    return table_index,coding,coding_b

def LZdecoder(Dict,coding_source):
    decodeStr = ''
   # for decoding ,split the coding into two parts: location + last letter
   # find the value of the location in dictionary and connect last letter to
decode


    for item in coding_source:
        index,last_str = item.split(',')
            # split it into two parts
        decodeStr += (Dict[int(index)]+last_str)
          # Add them

    return decodeStr
if __name__ == '__main__':
    source =
'00010010000100100001001000010010000100100001001000010010000100'
    Dict,codeword,codeword_b = LZencoder(source)
    print('The codeword are :',coding_b)
    print('The Dictionary are:',Dict)
    if LZdecoder(Dict,codeword) == source:
        print('Decoding Successfully')
```