Faculty of Engineering & Technology

Electrical & Computer Engineering Department

Second Semester 2022 – 2023

Information and Coding Theory – ENEE5304

Lempel-Ziv and Huffman Encoding

Prepared by

Abdallah Alabed - 1190781
Ahmaide AlAwawdah – 1190823

Instructor
Dr. Wael Hashlamoun

Section – 1

BIRZEIT

1 Table of Contents

2 List of Figures

3 Lempel-Ziv Encoding

3.1 Introduction

In this project, we will implement a program that generates a random sequence of symbols a, b, c, d, of size N, such that P(a)=0.4, P(b)=0.3, P(c) = 0.2, P(d)=0.1 and parses the data using LZ encoding and assign a number to each phrase. The program will Find the number of binary digits NB needed to encode the sequence and the number of bits per symbol. This step with be repeated 5 times, and the average results will be calculated. The program will run Lempel-Ziv Encoding for different sequence lengths, and outputs a table that shows the sequence length, size of encoded sequence, compression ratio and number of bits per symbol for different sequence lengths.

3.2 Theoretical Background

Lempel-Ziv Encoding works by processing an input sequence and maintaining a dictionary that stores the longest encountered words as code values. These words are replaced by their respective codes, resulting in the compression of the input file. As the number of long, repetitive words in the input data increases, the algorithm becomes more efficient in achieving compression.

The algorithm begins by initializing a dictionary that holds the phrases. It then scans the input string, identifying the longest section that is not yet present in the dictionary. This section is moved from the input string to the dictionary and assigned a unique index. Once the entire input string has been scanned, the number of bits required to represent each phrase can be determined.

Phrases consisting of an old phrase along with an additional bit are represented by the index of the old phrase and the additional bit. This encoding scheme enables data compression, and the level of compression increases as the size of the input string grows.

3.3 Results

The code is in Appendix1.

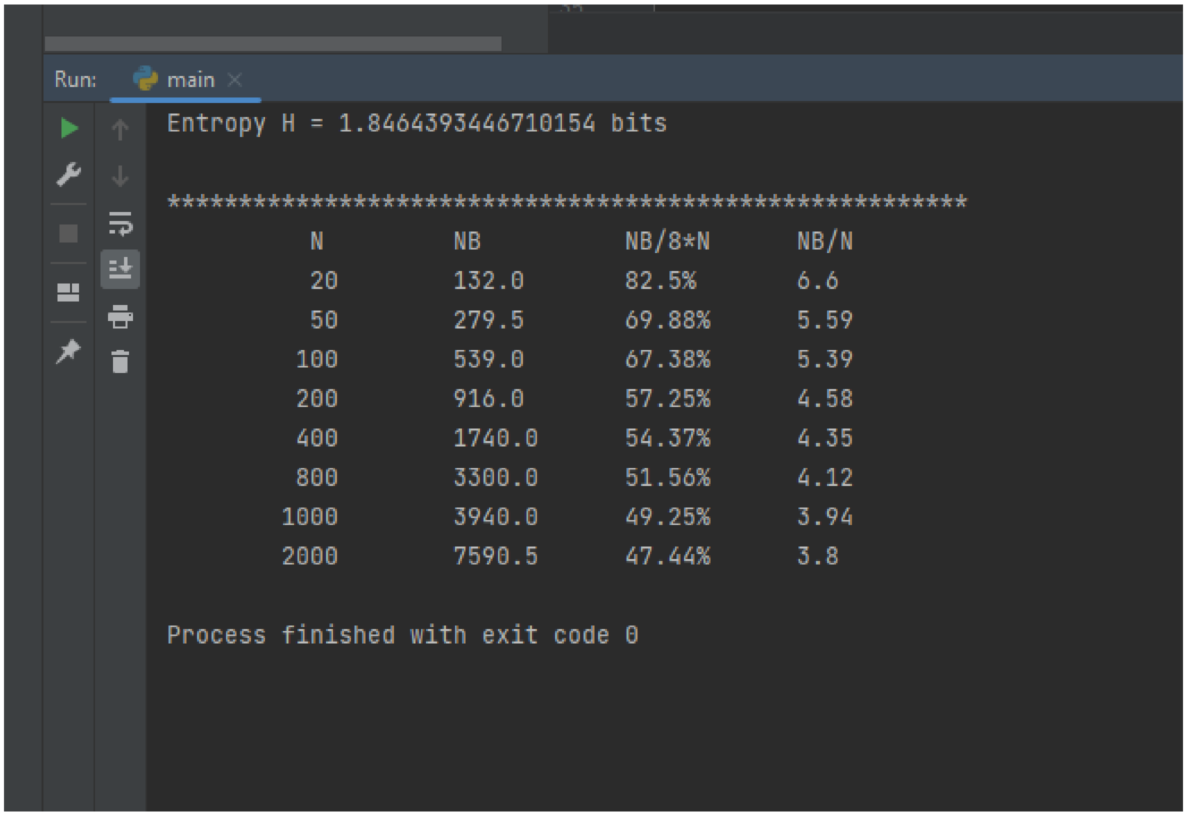
If the size N = 20 symbols, find:

- The number of binary digits NB needed to encode the sequence: NB = 123 bit
- The number of bits per symbol assuming that the tail of each codeword is in ASCII format (8 bits): 6.15 bit/symbol.

For various N values, the following table was obtained:

Table 1- Collected data for Lempel-Ziv Encoding

Sequence length N	Size of encoded sequence (NB)	Compression ratio NB /(8*N)	Number of bits per symbol (NB /N)
20	123.0	76.88%	6.15
50	289.25	72.31%	5.79
100	525.0	65.62%	5.25
200	939.75	58.73%	4.7
400	1732.5	54.14%	4.33
800	3272.0	51.12%	4.09
1000	3928.0	49.1%	3.93
2000	7548.0	47.17%	3.77



```
Entropy H = 1.8464393446710154 bits

**********************************************************
          N          NB          NB/8*N       NB/N
         20          132.0       82.5%        6.6
         50          279.5       69.88%       5.59
        100          539.0       67.38%       5.39
        200          916.0       57.25%       4.58
        400          1740.0      54.37%       4.35
        800          3300.0      51.56%       4.12
       1000          3940.0      49.25%       3.94
       2000          7590.5      47.44%       3.8

Process finished with exit code 0
```

Figure 1- results of Lempel-Ziv Encoding

```
N = 20, Generated Sequence: cbcccacbacbbabbdbcba
N = 20, Generated Sequence: abbaadbdacbacbcaacaa
N = 20, Generated Sequence: bbaadbaadaccbdaabbbd
N = 20, Generated Sequence: adbbabbbbdaaabaadbcc
N = 20, Generated Sequence: bbbccaaaababbacbcacc
N = 50, Generated Sequence: aacbdabcabcaacbaabbbbaddacaaabcdabaaaabcabbcdaabbc
N = 50, Generated Sequence: adaaababccbcacbdddbbbbcccddbbbcbbaaaacbbabaccbbaac
N = 50, Generated Sequence: aabaabbcbbdbdaadcbccaaacadbaaabaabaacdccccbdcadcbb
N = 50, Generated Sequence: aaaabbaabbadbbdaaadaacccbbbaaabdabaacabbbbcdbccadb
N = 50, Generated Sequence: ccabcadbcdacabbcaccbabcaababddacbcaadabababbabcada
N = 100, Generated Sequence: baacaabbcaccbabcaccbbdccbadcabbbccbbabddcadbabaabacdccaccbbacbcdaaabaadbbccbbaabcbcaaaacbbbddbcaabcb
N = 100, Generated Sequence: aaaccadaaaabaabdbbacbbababcccccabcdbadbdabaabaaacaadbacbdcbbabacabcbadbabcacbbbbaacbbacaacaacbbbaaa
N = 100, Generated Sequence: bbdccababacbbbbabbaabbbdabdadbacaababaabcabaababacaadacccbbababbcaabbacaacaabaadadcdaadcbbaacccaab
N = 100, Generated Sequence: abbabbaaaaccbacaadaaaccadabbaabbcaaaaacdaabacccaaadbbcacacabdbabdaabbcaacaaaabddaadbbabcaaabcabbcccd
N = 100, Generated Sequence: dccabaacdbdbcbcabcbacaadcbbdcbcbccdabbadabbbbbabccdbabdbababbbaadabacdbddccddbabbcbacaacaaabdbababca
N = 200, Generated Sequence: aacababdbacadaaaaacbbbabbbdaabddaaccabaacacabbababaacacbabaacaadcdaacbbbbaccdbacaaabbbbbbcabaacbdababaabaacabacbdccaabddcacbdbcaaabadbaaadaddacbcabaaacbcabdacaabcbabbbcbaacdbbacabbaaabccdacacabbbccca
N = 200, Generated Sequence: bcbaabccbaaacabcaaadbdaabaadaaacbdbcaabdaaabbabccbabdbbcbcbbbcdabcacaaaaaacaabdcbbdcaadbbaaaadbacccaabaaaabacaababacacadbbbbaadacbcbcbaacadabadaadacabdacaacbabaaadbdbbcdcbaacaaccaadbbaccacdaabaadbaaaa
N = 200, Generated Sequence: abbbacbcbdaacaaacdcbacbaabdbacaababccdbbbcccadabcbbaadbaaacbbcdcbcbcabacabbcaadbbacaabadadaacbaadacbacaadabaaccbdababaadcdbabbacbdcbabdabbcabdabaabdaacaabbbbcaabcabbaabbdaabbcabbadacaaacadabbaaadacaca
N = 200, Generated Sequence: bbcaaabcaccbbcaaaacaaabbadbacaaabacbbbbcccaacdbbaaccaabccacabcdbbcbdbbbbbdcaacbbdaccdbcdacbaabbbcacabadaacaacbdbbababbabbbbbacbabacabdbddbbcbbabadabbbdbbacbaddbccaabcaaabacaaadaacbaabbaabbabacabbbbbc
N = 200, Generated Sequence: aaaabcaadbabcbbbababababcabdbadaabbdcacabaabbaaddaabdbdbababaabadcaabbcbcbacaaaaabbcdbcbdabccabcabacabbacabacaaadcbdbcbbcaadacbccbabddadacacacbbababbaacadbcbcbaaabbaacbbacacabacbaaadbacbbbabbabbdbcaada
N = 400, Generated Sequence: aaabbacaabaaaacaadcabbabbbccbcabcabddbbbaddcaabdbccccaabbaaabcddbacababaaacabaaaaabadaabccbdbaddacabbabbcaabbdccbadddadbdbbaaabbcbababdcaaabcadadbbccabacdacaabacacaabbabdabbcacabbdcbdcbdbccbcabaababdadbbbabdaabbabbac
N = 400, Generated Sequence: aaaabbbbcbaccacbabcccaababdcbcaaaccbcaadcabaaacaaacaddbcabbaaabbbabacbcdbabcabacabbabcbcacaacbabbbabcaaccccbbaaadcbadababdaaaabdcbaabaabaabbaaaacbbaacacaaaacbcababbdbacaaabaababcaccdbbaaaaaabcbdacbbaacbaaaabadaaacdda
N = 400, Generated Sequence: abbabaaccabbbababcacaaabbaabdabaabaadbbbadbbabbbbbcccaabdabaabababacbdababaccbaacbaaacbcbaaaaacdaccabccabccbcbabaaaaabdbcbbbbacbcabccabdacbdbabadaabbccbbdabaabbbaadaccaaacbccdbdabbacbdaacddacccacdabacaabbdbbacddbabab
N = 400, Generated Sequence: aaaddabcaaabbbbdcbaabbacabaadbaacbcdbbcabbdcdbaaaccaaaddbbbabaaaabadaadbacbacbbacaadbbdacacacccdadbaaaacabddadcbaadabbcbbabaacbaaabbbaadbcaabbaaabdbaabacaaaaccacadcabcbabbabaabbdaacacdaccbaaabdcacbaaabcadaccaaacadadabb
N = 400, Generated Sequence: caaddaaaaabdbbbcbaabbbbdcabaabbbabbadccaaabcaaababbbadcbaacaababacaabaabbcacababaabacbdcbbcaaabbaacabdaaabbacbcabaabaaabbacadbadabacbababbbbcabbaababbacaacdaadbbbabbddbbcabcbbbbacacaaaabcabcabcaaabaaaaaacccbababdbbaaa
N = 800, Generated Sequence: acbcaacabccbaaacbbacacccabaaccababbcbabbadadaabbaabbcbacaabdabcacdbabcbaabccacbaaaadaaaadbcaaabbabcbbbabacbbababdcacabbbbbaacbabcbadcccdabaccbababdacaaabacaaadddbcabcaabaabaababcaadacababdaccccbaacbbbbdcaaababbacbac
N = 800, Generated Sequence: dcabbcbcaaacbcbaabbabbaaabbaaadbacbccacaacbbabcbbabcacdcdbcbbabbbaacdbcabbaabdcabbbbcbdaccabaddbbaabaabbdcaaaacacabcbbcdbabcbbacbacabbbababcaaaabaaaaccaaadbacaacabbbdaabacacaddcaaabcaaacabadaabcabbccaabaaabbabcacdacacb
N = 800, Generated Sequence: bcaaadaabadccaacaaabdaabaacbccacdcbacbaabcddbcaaababccaaabbabccababaaabdbdabcbbaddbacbabbababcbdbbccacbaaccbbbbcbaaabadacdbbcbadabbcbdaabaabbcabbbccbddbbccbbacabaaccabcacbadccaacaaabcbbaaabdbbbcacbbaaaabaabcdbcbaaadbb
N = 800, Generated Sequence: abababacbabdabaabbbabbcaabbbacbabcabbbbbdacabcacbcbabccdaabccaaacbccaaabaababbccabacacdcaabaaaacaabacbadbbadaabcaaadabbdaaaaabacaaaaabdbacaaaabadabbccaaabdbabbadabbadbbacabdccaaaacaccbabcbacabbbababbdcbdabbaaabdaabbdab
N = 800, Generated Sequence: cdacabababaabbcacbcaacbdbaaaadddaaaccdbcabacaadbcacabcbbabaabdbcbcbddaabbccabbacdcadbadabddcaadabccabbbaabcabadaabaddadaaacbcaabbcacbbcdccbdabcccdabadcbabccabbbdacbcbbcbacaccbaccbaacccbbcdabaaabadabdabbadbbccabbbabaabb
N = 1000, Generated Sequence: bbabddaacbcccadbbbabbaababbcbabacbabbaaacbabbadcaaaabbaaacbbaaacbaabdabacdbddadcaccbabbbaaabbaaadcbaabadacbacdbacddabbccaacabcaaaabbacbabaaaaaaacaaccdbaacdcbbcaacbcaadaabbbbacabcbdbabaadacaaaaaddaddbaddcbacaaaacabaab
N = 1000, Generated Sequence: aaaabadbbbabcabbbabaaccaccaabbaacbadcbcdbabbddbaacacadaabcadbacbbcbbabbbaabcabbdabdaccacbbbabcbaaadadbcaddcbbaabaacadaaadcbbdbdcaacabbcabaacabbdaacccbacbbabdbccabbccbcababcaacaaaacdbaaddccabbacccdbcdaabcaacccaaadabaac
N = 1000, Generated Sequence: cababbbbaabbbdacdccaabcbaabbbbccbabbbcbbcabdababaacbccbaaaabaabacdcdcacabbaabbacbbbacbcabccbabbadcaababcbabcbbbaaaaabbbcbccacaabdabbadbaaacbbaaccabaabbbbcbbaabcbaaaccabcbaacaaccdabdbacbdbbbacbbdabbacbcabbabcbbbaadacb
N = 1000, Generated Sequence: baaabbbbaacdabacaacdbdccdaaabccababbcdaaabacadaccbdbcaadabacdbdbadbadcbacaacabbadabaaaaaaacbccbcaaaaadbaabcabdcbaaacdbbadcbdcbbaaabcbcaabaacaacbbabaacbabbabcabcaaaabbbabcccaaabbbbaaaabaaacacaacbbdbbbdbaacbdbadadcad
N = 1000, Generated Sequence: dbaaabadaaaadbadbbcdacbcacacccbcbaccaabcbbbabadbbabdbdccbacbbcbadadaacababbabbbcbcbadbacbacadadacaabbccbaaaabddcbaccaaabaaaaacbabbdbacacadaaadaaabbaaabbacabdbaabccabdabbbaabdabbbbaaccaacaababcbbdbbaaadacbccccdbbabaab
N = 2000, Generated Sequence: daacabbaababaabbadaabbbcaaabccbacbbaaccaabbbdbaaaaaadbcaabacacbbcdcbbcdbcbbccbcbbaabbaaaaabcccabbabacaabbaabbcdabdaabaacacccbaaaadbbcacdabaacbcbabcbdbaabbcdcbcaaaadacbbcbdbaabbddacacacaabaadaaabaabaabaacbcababcaaba
N = 2000, Generated Sequence: cdabaaaaccaadacaccaabaaabcaccbabaabbcadbbdbcabaccabccbdaacaabccdbbbabbaabdaaacabbabdcaaabbaabbabcacbabccaabbbabbdaacabbbcacbcabbdabbadcdabbbbcdcaadacdaaccadacdacccbcabaaaacaaddaaababccaaccbabbbabbbaccaacbcdbaaababbdba
N = 2000, Generated Sequence: abaacdaaabbaabbaadabcabbbbacabacacabbbacbcabbbbbbadccacacabaaabcbcdacaaabacbaabdbacaabbaabbbcaadabdabbababadbababcbabcbbabaaaccccbdbcaaaaddbacaaacababaacacaacaababaaaacbbbccdaacdcaaabbaadabaabacaabbaaaaaadbbccbbabbcb
N = 2000, Generated Sequence: aaaacdbbbaddacbaaabcaaaadacdabdacbcaccddbaaabdabbadbcaabbacaaaccabaccabddbacbbaabdbaabbdacbcabaccadbbcccaaaccaccacdaccdbaccdcaaacbcabbbbdbbaaabadabacaaabaaacbbacbbaacabbbbababaadbcbaabbcaccbabbaabacbdbabbcacabbaccaaab
N = 2000, Generated Sequence: dabaabaaaaabbcbbaaabbcbcabbaaaccaaaabacccabcbbccabbadbabdacabdabbaaacdaacbaabcbccbbabdcaaacabbcdacbaaaaccadcabaacaaaaaaaaaaaaccbacabaaccbbabacaccbaadbaaaacdabbccccacbbcdacabdbbaaabaccbadaacbbabdbbaccdbbabbdaabbcdbbbac
```

Figure 2- results of Lempel-Ziv Encoding

4 Huffman Algorithm

4.1 Introduction

Developed in 1952 by David A. Huffman, the Huffman Encoding technique is a well-known one for lossless data compression. By allocating shorter bit sequences to characters that appear more frequently, Huffman encoding aims to reduce the total data size. Since it uses the frequency distribution of the dataset's letters or symbols, it is a frequency-based encoding technique [3].

4.2 Theoretical Background

Huffman Encoding starts by determining the frequency of each character in the input data. The technique builds a tree structure or priority queue with nodes that stand in for the letters and their frequencies (often referred to as a Huffman Tree). The node with the lowest frequency is set to be first to do the Huffman operation with.

Then it moves forward by removing the last two nodes from the priority queue with the fewest frequencies (the first in order) and merging them to create a new node, whose frequency is the sum of its two children's frequencies. The Huffman tree's root is the last node in the queue to be reached after repeating these actions. The frequency of the root node is the same as the sum of all character frequencies, as shown below in figure 1.

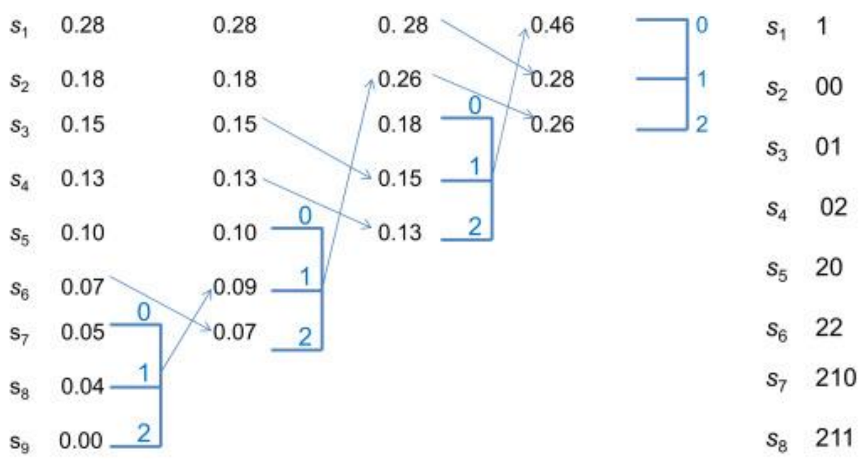


Figure 3: Huffman Encoding algorithm.

As shown in figure 1. The algorithm assigns '0' to the edge with higher probability and '1' to the edge with lower probability of each node, counting from the tree's root. The set of bits that are encountered on the way from the root to the node that corresponds to a specific character is then known as the Huffman code for that character. The beauty of Huffman encoding is that no code is a prefix of another, ensuring that the compressed data may be used to correctly reconstruct the original data, which made the algorithms impact in the world today.

Huffman encoding successfully decreases the average code length and, as a result, the amount of data through this process. This makes it the best tactic to use when it's necessary to reduce the quantity of data for storage or transmission without sacrificing the data's integrity.

The python code that was written for the Huffman code can be found in Appendix 2.

4.3 Huffman Encoding Results

After running the code that is in Appendix 2 which takes the input from a file as the input comes as a symbol with its probability, first it gives each symbol a code depending on its probability, then constructs a random sequency of 100 symbols by taking the consideration of the probability of each symbol, encodes it then calculates the size of the encoded sequence, the compression, and finally the average bits per symbol, so the input was as follows:

```
a 0.4
b 0.3
c 0.2
d 0.1
```

And the output was:

The codeword for each symbol:

a 1

b 01

c 000

d 001

Random String:

aaabcccaccbcbacababcabcbbaacbaaccaccbbdbadbaaaaadacaacaaabcbcdabdbbadbcb babaaabcabbbbcabbdbacacbaaaa

Random String Code Word:

1110100000000001000000010000110001011010001010000101110000111000000100
000001010010110010111111001100011000111010000100000110100101011001010
00010110111101000101010101000101010010110001000011111

Size of encoded sequence: 191 bit

Compression Ratio: 23.875%

Number of bits per symbol: 1.91 bit/symbol

5 Comparison between the Huffman and Limpel-Ziv codes

Sequence length N	Size of encoded sequence (NB)	Compression ratio NB /(8*N)	Number of bits per symbol (NB /N)
100 (Limpel-Ziv)	525.0	65.62%	5.25
100 (Huffman)	191	23.875%	1.91

In summary, the results suggest that Huffman coding performs better in terms of compression ratio and number of bits per symbol compared to Lempel-Ziv coding. Huffman coding typically achieves more efficient compression by assigning shorter codes to more frequently occurring symbols. On the other hand, Lempel-Ziv coding excels in cases where repetitive patterns or long sequences occur frequently. The choice between the two techniques depends on the specific characteristics and requirements of the data being compressed.

6 Conclusion

In conclusion it was shown how sequences of different symbols can be encoded in different ways with different results, this project helps in showing how efficient the Huffman code and the Lempel-Ziv can be in reducing the amount of data that represents the information without any losses of information, these encoding techniques can be very useful in applications that require transmitting huge amounts of data such as commination networks and multimedia storages.

7 References

[1]. *LZW (Lempel–Ziv–Welch) Compression technique*. (n.d.). https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/

Accessed on June 21st 2023

[2]. *LZW* Brilliant Math & Science Wiki. https://www.scaler.com/topics/lzw-compression/

Accessed on June 21st 2023

[3]. *Huffman Code | Brilliant Math & Science Wiki*. (n.d.). Huffman Code | Brilliant Math & Science Wiki. https://brilliant.org/wiki/huffman-encoding/

Accessed on June 21st 2023

[4]. *Huffman Code - an overview | ScienceDirect Topics*. (n.d.). Huffman Code - an Overview | ScienceDirect Topics. https://doi.org/10.1016/B978-075066310-6/50007-6

Accessed on June 21st 2023

8 Appendix

8.1 Appendix 1

```
import random
import math


def calcEntropy(sequence):
    modified_numbers = [-e * math.log2(e) for e in sequence]
    return sum(modified_numbers)


def calcNB(sequence, bit):
    return len(sequence.keys()) * (bit + 8)


def calculate_average(array):
    total = sum(array)
    average = total / len(array)
    return average


def findPhrases(input_str, keys_dict):
    ind = 0
    inc = 1
    i = 1
    while True:
        if not (len(input_str) >= ind+inc):
            break
        sub_str = input_str[ind:ind + inc]
        if sub_str in keys_dict:
            inc += 1
        else:
            keys_dict[sub_str] = i
            i += 1
            ind += inc
            inc = 1


def find_phrases(num_of_bits, keys_dict):
    temp = {}
    for present_value in keys_dict.keys():
        if len(present_value) == 1:
            temp[present_value] = '0'.rjust(num_of_bits, '0') +
present_value[0]
        else:
            for past_value, past_key in keys_dict.items():
                if present_value[:len(present_value) - 1] == past_value:
                    temp[present_value] = str(bin(past_key).replace("0b",
"")).rjust(num_of_bits, '0') + present_value[len(present_value) - 1]
    return temp


def LZW(probabilities):
    for N in (20, 50, 100, 200, 400, 800, 1000, 2000):
        compressionValues = []
        NB_Values = []
        numOfBitsPerSymbol = []
        for i in range(0, 5):

            random_sequence = random.choices(list(probabilities.keys()),
list(probabilities.values()), k=N)
            input_str = "".join(random_sequence)
            keys_dict = {}
            findPhrases(input_str, keys_dict)
            bits = math.ceil(math.log2(len(keys_dict.items())))
            codewords = find_phrases(bits, keys_dict)
            encoded_sequence = ''.join([str(item) for item in
codewords.values()])
            NB = calcNB(keys_dict, bits)
            BPS = NB / N
            NB_Values.append(NB)
            compressionValues.append(NB/(N*8))
            numOfBitsPerSymbol.append(BPS)
            # print("N = "+str(N)+", Generated Sequence:
"+"".join(random_sequence))
            # print(list(keys_dict))
            # print("Encoded Sequence: " + encoded_sequence)
            # print("NB = " + str(NB))
            # print("compression ratio: " + str(compressionValues[i]))
            # print("Number of bits per symbol:" + str(BPS) + "\n")
        print(f"{str(N):>12}" + "\t\t" +
str(calculate_average(NB_Values)) + "\t\t" +
              str(round(calculate_average(compressionValues)*100, 2)) +
"%"
              + "\t\t" + str(round(calculate_average(numOfBitsPerSymbol),
2)) + "\t\t")


def main():
    probabilities = {'a': 0.4, 'b': 0.3, 'c': 0.2, 'd': 0.1}
    print("Entropy H = " + str(calcEntropy(probabilities.values())) + "
bits\n")
    print("********************************************************")
    print("\t\t  N\t\t\tNB\t\t\tNB/8*N\t\tNB/N")
    LZW(probabilities)


if __name__ == "__main__":
    main()
```

8.2 Appendix 2

```
import random
import math

class Symbol:
    def __init__(self, char, probability):
        self.char = char
        self.probability = probability
        self.codeword=''

class Node:
    def __init__(self, probability, elements):
        self.elements=elements
        self.probability = probability

input=open("input.txt", "r")
data = input.read().splitlines()
input.close()
symbols={}
nodes=[]
for i in range(len(data)):
    element=data[i].split()
    sym= Symbol(element[0], float(element[1]))
    symbols[element[0]]=(sym)
    ele=[]
    ele.append(sym)
    node=Node(sym.probability, ele)
    nodes.append(node)

nodes = sorted(nodes, key=lambda x:x.probability, reverse=True)

while len(nodes) > 2:
    ele=nodes[len(nodes)-1].elements.copy()
    ele.extend(nodes[len(nodes)-2].elements)
    prob = nodes[len(nodes)-1].probability + nodes[len(nodes)-
2].probability
    for element in nodes[len(nodes)-1].elements:
        element.codeword = '1' + element.codeword

    for element in nodes[len(nodes)-2].elements:
        element.codeword = '0' + element.codeword

    newNode=Node(prob, ele)

    nodes.pop()
    nodes.pop()
    nodes.append(newNode)
    nodes = sorted(nodes, key=lambda x: x.probability, reverse=True)

for element in nodes[0].elements:
    element.codeword = '0' + element.codeword

for element in nodes[1].elements:
    element.codeword = '1' + element.codeword

symbols2={}
```

```python
for key, value in symbols.items():
    print(key, value.codeword)
    symbols2[key]=value.probability

random_sequence = random.choices(list(symbols2.keys()),
list(symbols2.values()), k=100)
random_sequence= ''.join(random_sequence)

print("\nRandom String:")
print(random_sequence)
FinalCodeword=''

for r in random_sequence:
    FinalCodeword = FinalCodeword+symbols[r].codeword
print("\nRandom String Code Word:")
print(FinalCodeword)

Nb=len(FinalCodeword)
print("\nSize of encoded sequence:", Nb)

Ratio=Nb/800
print("\nCompression Ratio:", Ratio)

nOfBits=Nb/100
print("\nNumber of bits per symbol:", nOfBits, "b/s")
```