FACULTY OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF COMPUTER ENGINEERING

ENEE5304 - INFORMATION AND CODING THEORY

Second semester 2022/2023

Course Project on Source Coding

Prepared by:

Tariq Odeh – 1190699

Ali Mohammed - 1190502

Section: 1

Instructor: Prof. Wael Hashlamoun

Date: 1st Jul 2023

Table of Contents

List of Figures

List of Tables

1. Part one – Lempel-Ziv Encoding of Random Symbols

1.1. Introduction

We have 4 different symbols: a, b, c, d. Their probabilities are P(a) = 0.4, P(b) = 0.3, P(c) = 0.2, P(d) = 0.1. The task starts from generating multiple sequences with different lengths (20, 50, 100, 200, 400, 800, 1000, 2000) till encoding them using Lempel-Ziv method. The first task of this part is to calculate the source entropy H. Then, generating a random five sequences of length 20 symbols and finding the average value of N_B, where N_B is the number of binary digits needed to encode the sequence. Then, the compression ratio relative to the ASCII code (average value of $N_B/N*8$) is found. Then, the size of encoded sequence, compression ratio, and the number of bits per symbol are calculated for the rest of lengths.

1.2. Theoretical Background

1.2.1. Source entropy

"A cornerstone of information theory is the idea of quantifying how much information there is in a message. More generally, this can be used to quantify the information in an event and a random variable, called entropy and is calculated using probability.[1]" Source entropy, denoted as H(S), is a measure of the uncertainty or randomness associated with a source of information. It quantifies the average amount of information contained in each symbol or event emitted by the source. The formula to calculate the source entropy is given by:

$$H(S) = -\sum_i p_i \cdot log_2 p_i \qquad 1.1$$

"… the Shannon entropy of a distribution is the expected amount of information in an event drawn from that distribution. It gives a lower bound on the number of bits […] needed on average to encode symbols drawn from a distribution P. [2]"

The source entropy provides an indication of the average amount of information required to represent or encode symbols from the source. Higher entropy values indicate greater uncertainty and randomness, while lower entropy values indicate more predictability or regularity in the source.

1.2.2. Lempel-Ziv encoding

Brief introduction

Lempel-Ziv encoding (LZ78) is a lossless data compression technique devised by Abraham Lempel and Jacob Ziv in 1978. The method involves constructing a dictionary of unique strings encountered in the input data. Whenever a new string is encountered, the algorithm checks if it already exists in the dictionary. If found, it outputs the corresponding index. If the string is not present in the dictionary, the algorithm outputs the string's length followed by the string itself, and then adds it to the dictionary. Despite its effectiveness in achieving data compression, LZ78 has certain limitations. For instance, compressing large data volumes with LZ78 can be time-consuming. Additionally, implementing LZ78 in hardware can pose challenges.

How it works?

1. The algorithm starts with an empty dictionary.
2. Then, the algorithm reads the sequence character by character.
3. For each character, it checks if it is in the dictionary or not.
4. If the character is not in the dictionary, the algorithm adds it to the dictionary and outputs the character.
5. If the character is in the dictionary, the algorithm outputs the index of the dictionary entry that contains the character.

6. The algorithm repeats steps 2-5 until it reaches the end of sequence.

The following figure is a detailed example of how Lempel-Ziv works:

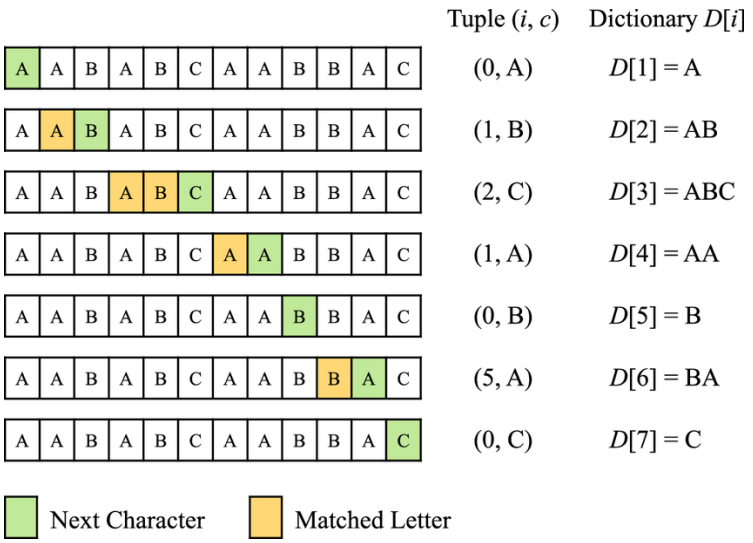

Figure 1-1: how Lempel-Ziv works [3]

1.3. Results and Analysis

1.3.1. Part1: Source entropy

This part calculates the source entropy H in bits, the code of this part is in appendix 5.1.

$$H(S) = -\sum_i p_i \cdot log_2 p_i$$

$$H(S) = -\big((0.4 \cdot log_2 0.4) + (0.3 \cdot log_2 0.3) + (0.2 \cdot log_2 0.2) + (0.1 \cdot log_2 0.1)\big)$$

$$H(S) = 1.8464393446710154$$

```
Entropy: 1.8464393446710154
--------------------------
```

Figure 1-2: source entropy

1.3.2. Part2: Multiple sequences of 20 symbols and average value of N_B

The code of this part is in appendix 5.2 and 5.3. This part generates 5 different sequences of 20 symbols. The program outputs the generated sequence along with the encoded one. Also, the content of the dictionary, number of phrases, number of bits per phrase, and the

number of binary digits needed to encode the sequence (N_B). The figure below shows how the output of the program is.

```
N = 20
------
Generated sequence: aacaacacdbaabccbbaaa
Encoded sequence: 0000a0001c0001a0000c0010d0000b0011b0100c0110b0011a
------------------------------------------------------------------------
Dictionary Location     Dictionary Contents Codeword
       1 001                          a            <0,a>
       2 010                         ac            <1,c>
       3 011                         aa            <1,a>
       4 100                          c            <0,c>
       5 101                        acd            <2,d>
       6 110                          b            <0,b>
       7 111                        aab            <3,b>
      8 1000                         cc            <4,c>
      9 1001                         bb            <6,b>
     10 1010                        aaa            <3,a>
------------------------------------------------------------------------
# of phrases: 10
# of bits per phrase: 12 -> [4 + 8]
Nb: 120 -> [12 * 10]
```

Figure 1-3: program's output for each generated sequence

The following figure is the summary table of the five generated sequences that contains the number of binary digits N_B of each sequence. Then, the average N_B is calculated.

```
----------------------
          N      NB
    1     20     120
    2     20     132
    3     20     132
    4     20     108
    5     20     132

Average NB = 124.8
```

Figure 1-4: average NB

1.3.3. Part3: Generating sequences with different number of symbols

The idea of this part is to compare the results as long as the length of the sequence increases. However, the code of this part is in appendix 5.4. The program simply outputs the size of encoded sequence (N_B), compression ratio **(**N_B/[8*N]**)**, and the number of bits per symbol (N_B/N) of each sequence generated with the mentioned lengths.

```
************************************************
   N      Nb        Nb/8*N          Nb/N
  20     120       75.00%          6.00
  50     273       68.25%          5.46
 100     532       66.50%          5.32
 200     975       60.94%          4.88
 400    1710       53.44%          4.28
 800    3360       52.50%          4.20
1000    3920       49.00%          3.92
2000    7650       47.81%          3.83
```

Figure 1-5: comparison table of encoding sequences with different lengths

Table 1-1: comparison table of encoding sequences with different lengths

Sequence length N	**Size of encoded sequence (NB)**	**Compression ratio NB /(8*N)**	**Compression ratio NB /(8*N)**
20	120	75%	6
50	273	68.25%	5.46
100	532	66.5%	5.32
200	975	60.94%	4.88
400	1710	53.44%	4.38
800	3360	52.5%	4.2
1000	3920	49%	3.92
2000	7650	47.81%	3.83

After reaching this point, it is obvious that as the length of the sequence increases, the compression ratio decreases. This result confirms that Lempel-Ziv is efficient for long sequences.

2. Part Two – Huffman Code

2.1. Introduction

In the beginning, in this part, a program in the Python language is going to be implemented to find the codewords for the characters (a, b, c, d) with the probabilities P(a) = 0.4, P(b) = 0.3, P(c) = 0.2, P(d) = 0.1 using Huffman code. The codewords for each character will be generated, and all Huffman code results, such as the average number of bits per character for the code and the number of bits needed to encode 100 randomly generated symbols, will be shown. A comparison between the Huffman and Lempel-Ziv codes will be made for the case when N = 100.

2.2. Theoretical Background

Huffman coding is a well-known Greedy Algorithm widely used for lossless data compression. Its primary objective is to minimize the number of bits used for frequently occurring characters. This coding technique employs variable-length encoding, assigning variable-length codes to individual characters. The length of a character's code is determined by its frequency in the given text, with the least frequently occurring character receiving the longest code. The Prefix Rule is a crucial aspect of Huffman coding, aimed at preventing ambiguities during decoding. It ensures that no code assigned to a character serves as a prefix for the code assigned to any other character. The following are the major steps involved in Huffman Coding:

1. Determine the frequencies for each character in the given text.
2. Sort the symbols in descending order based on their frequencies.
3. Merge the two symbols with the lowest frequencies by creating a new branch with a distinct binary digit and summing their frequencies.

4. The binary digits from the previous node, where merging occurred, are appended to the original symbols, forming the codeword for each symbol.
5. Repeat the merging process until no more symbols remain to be merged.
6. By following these steps, Huffman coding achieves efficient compression by assigning shorter codes to more frequent characters and longer codes to less frequent characters.

2.3. Results and Analysis

The code of this part is in appendix 5.5. Using Huffman coding and the given probabilities, the codewords for the characters (a, b, c, d) are as follows:

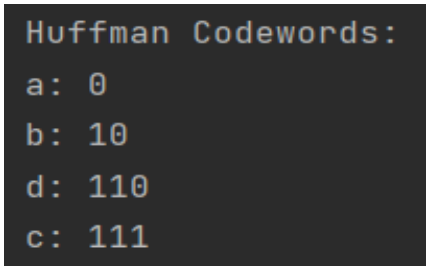

Figure 2-1: Huffman Codewords

To find the average number of bits per character for the Huffman code, we need to calculate the weighted average of the codeword lengths based on the probabilities of the characters.

- Average bits per character = (P(a)*Length of codeword for 'a') + (P(b)*Length of codeword for 'b') + (P(c)*Length of codeword for 'c') + (P(d)*Length of codeword for 'd')

Substituting the given probabilities and codeword lengths, we get:

- Average bits per character = (0.4 * 1) + (0.3 * 2) + (0.2 * 3) + (0.1 * 3) = 0.4 + 0.6 + 0.6 + 0.3 = 1.9 bits per character

Figure 2-2: Average bits per character

Therefore, the average number of bits per character for the given Huffman code is 1.9 bits.

To find the number of bits needed to encode 100 randomly generated symbols known that each character requires 8 bits (1 byte) for encoding, we multiply the number of symbols by the number of bits per symbol: Number of bits = 100 * 8 = 800 bits.

```
['d', 'b', 'a', 'c', 'a', 'b', 'a', 'b', 'a', 'c', 'b', 'b', 'a', 'a', 'a', 'b', 'a', 'b', 'a', 'b', 'a
Number of bits needed using ASCII code: 800
```

Figure 2-3: Number of bits needed to code 100 symbols

```
Huffman part
------------

Huffman Codewords:
a: 0
b: 10
d: 110
c: 111
Average bits per character (Huffman): 1.9000000000000001
['d', 'b', 'a', 'c', 'a', 'b', 'a', 'b', 'a', 'c', 'b', 'b', 'a', 'a', 'a
Number of bits needed using ASCII code: 800
Size of encoded sequence (NB): 188 bits ( 24 bytes)
Compression ratio (NB/(8*N)): 0.235
Number of bits per symbol (NB/N): 1.88

Limpel-ziv part
---------------

Size of encoded sequence (NB): 546 bits
Compression ratio (NB/(8*N)): 68.25
Number of bits per symbol (NB/N): 5.46
```

Figure 2-4: The full results

Table 2-1: Comparison between the Huffman and LimpelZiv codes

Sequence length N	Size of encoded sequence (NB)	Compression ratio NB /(8*N)	Compression ratio NB /(8*N)
100 (Limpel-Ziv)	546	68.25%	5.46
100 (Huffman)	188	23.5%	1.88

- From the above table, we can see that the size of the encoded sequence (NB) for Huffman coding is 188 bits, while for Lempel-Ziv coding it is 546 bits. This indicates that Huffman coding achieves a smaller encoded sequence size compared to Lempel-Ziv coding.
- The compression ratio (NB/(8*N)) for Huffman coding is 0.235, meaning that the compressed data is approximately 23.5% of the original size. In contrast, the compression ratio for Lempel-Ziv coding is 68.25%, indicating a higher level of compression.
- The number of bits per symbol (NB/N) for Huffman coding is 1.88, while for Lempel-Ziv coding it is 5.46. This implies that Huffman coding achieves a lower number of bits per symbol, resulting in more efficient encoding compared to Lempel-Ziv coding.

It's important to note that the performance of Huffman coding and Lempel-Ziv coding can vary depending on the specific characteristics of the data, the chosen encoding parameters, and the compression requirements. While Huffman coding is generally more efficient, and Lempel-Ziv coding algorithms outperform it in cases of highly repetitive or structured data. Therefore, the selection of the encoding method should be based on careful analysis and consideration of the specific use case and requirements.

3. Conclusion

In this project, we learned about the use of Huffman coding and Lempel-Ziv coding for data compression. Huffman coding proved to be efficient in terms of compression ratios and average number of bits per character, especially for data with known probability distributions. Lempel-Ziv coding, on the other hand, demonstrated its effectiveness in compressing highly repetitive or structured data. This highlights the importance of selecting the appropriate coding method based on the specific characteristics of the data and the desired compression goals. By comparing these two coding techniques, we gained insights into their strengths and limitations, enabling us to make informed decisions in future data compression tasks.

4. References

[1] Shukueian, S. (2022, August 20). *Information Theory — Entropy - Sobhan Shukueian - Medium*. Medium. https://shukueian.medium.com/information-theory-entropy-8dd381ebd6b9

[2] Goodfellow, I., Bengio, Y., & Courville, A. (2016, November 18). *Deep Learning*. MIT Press. P.74

[3] *Example of compression process of LZ78 for the text data "AABABCAABBAC."* (n.d.). ResearchGate. https://www.researchgate.net/figure/Example-of-compression-process-of-LZ78-for-the-text-data-AABABCAABBAC_fig1_337580417

[4] *LZW Lempel Ziv Welch Compression technique*. (2023, March 15). GeeksforGeeks. https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/

[5] GeeksforGeeks. (2023, April 6). *Huffman coding: Greedy Algo-3*. GeeksforGeeks. https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

[6] *Huffman code*. Brilliant Math & Science Wiki. (2022, May 2). https://brilliant.org/wiki/huffman-encoding/#:~:text=Huffman%20code%20is%20a%20way,more%20rarely%20can%20be%20longer.

5. Appendix

5.1. Entropy code

```
import math


def calculate_entropy(probability):
    return -sum([prob * math.log2(prob) for prob in probability])


probabilities = [0.4, 0.3, 0.2, 0.1]
entropy = calculate_entropy(probabilities)
entropy_str = "Entropy: " + str(entropy)
print(entropy_str)
print('-' * len(entropy_str) + '\n')
```

5.2. LimpelZiv code

```
import math
import random


def generate_sequence(symbols, probabilities, length):
    return ''.join(random.choices(symbols, probabilities, k=length))


def parse_sequence(sequence, dictionary):
    index = 1
    phrase = ''
    for character in sequence:
        if phrase in dictionary.values() or phrase == '':
            phrase = phrase + character
            continue
        dictionary[index] = phrase
        index += 1
        phrase = character
    dictionary[index] = phrase


def encode_phrases(num_of_head_bits, dictionary):
    encoded_phrases = {}
    for present_value in dictionary.values():
        if present_value in encoded_phrases.keys():
            prev_index =
list(dictionary.keys())[list(dictionary.values()).index(present_value)]
            encoded_phrases[present_value + "0"] =
str(bin(prev_index).replace("0b", "")).rjust(num_of_head_bits,

'0') + 'x'
        elif len(present_value) == 1:
            encoded_phrases[present_value] = '0'.rjust(num_of_head_bits, '0')
+ present_value[0]
        else:
            for past_key, past_value in dictionary.items():
                if present_value[:len(present_value) - 1] == past_value:
```

```
                    encoded_phrases[present_value] =
str(bin(past_key).replace("0b", "")).rjust(num_of_head_bits, '0') + \

present_value[len(present_value) - 1]
                    break
    return encoded_phrases


def calculate_nb(head, tail, num_of_phrases):
    return (head + tail) * num_of_phrases


def calculate_average_nb(iterations, n):
    average_nb = 0
    for i in range(iterations):
        temp_lz_dict = {}
        sequence = generate_sequence(n)
        parse_sequence(sequence, temp_lz_dict)
        head = math.ceil(math.log2(len(temp_lz_dict.items())))
        nb = calculate_nb(head, 8, len(temp_lz_dict.items()))
        average_nb += nb
    return average_nb / iterations
```

5.3. Multiple sequences of 20 symbols and average value of NB

```
from LimpelZiv import *
import pandas as pd

summary_output = "-----------------------\n\t \t   N\t  NB\n"
symbols = ['a', 'b', 'c', 'd']
probabilities = [0.4, 0.3, 0.2, 0.1]
locations = []
content = []
codeword = []
i = 0
Nb_avg = 0
N = 20
for i in range(5):
    lz_dict = {}
    generated_sequence = generate_sequence(symbols, probabilities, N)
    parse_sequence(generated_sequence, lz_dict)
    head_bits = math.ceil(math.log2(len(lz_dict.items())))
    codewords = encode_phrases(head_bits, lz_dict)
    encoded_sequence = ''.join([str(item) for item in codewords.values()])
    Nb = calculate_nb(head_bits, 8, len(lz_dict.items()))
    Nb_avg += Nb
    print("N = " + str(N) + "\n" + '-' * len("N = " + str(N)))
    print(
        "Generated sequence: " + generated_sequence + '\n' + "Encoded
sequence: " + encoded_sequence + '\n' + '-' * len(
            "Encoded sequence: " + encoded_sequence))
    for key, value in lz_dict.items():
        locations.append(
            str(key).rjust(3) + " " + str(bin(key).replace("0b",
"")).rjust(head_bits - 1, '0') + " " * 7)
        content.append(str(value).rjust(15) + " " * 8)
        codeword.append("<" + str(int(codewords[value][:len(codewords[value])
- 1], 2)) + "," + codewords[value][
```

```
            len(codewords[value]) - 1] + ">")
        data = {
            "Dictionary Location": locations,
            "Dictionary Contents": content,
            "Codeword": codeword
        }
        df = pd.DataFrame(data)
    print(df.to_string(index=False))
    print("-" * 80)
    print("# of phrases: " + str(len(lz_dict.items())))
    print("# of bits per phrase: " + str(head_bits + 8) + ' -> [' +
str(head_bits) + ' + 8]')
    print("Nb: " + str(Nb) + ' -> [' + str(head_bits + 8) + ' * ' +
str(len(lz_dict.items())) + ']')
    print('\n\n')
    summary_output += str(i + 1).rjust(5) + "\t" + str(N).rjust(5) + "\t" +
str(Nb).rjust(5) + "\n"
    locations = []
    content = []
    codeword = []
print(summary_output)
print("Average NB = " + str(Nb_avg / 5))
```

5.4. Generating sequences with different number of symbols

```
from LimpelZiv import *

symbols = ['a', 'b', 'c', 'd']
probabilities = [0.4, 0.3, 0.2, 0.1]
summary_output = "*********************************************\n\tN\t
Nb\t\tNb/8*N\t\t\tNb/N\n"
Ns = [20, 50, 100, 200, 400, 800, 1000, 2000]
for N in Ns:
    lz_dict = {}
    generated_sequence = generate_sequence(symbols, probabilities, N)
    parse_sequence(generated_sequence, lz_dict)
    head_bits = math.ceil(math.log2(len(lz_dict.items())))
    codewords = encode_phrases(head_bits, lz_dict)
    encoded_sequence = ''.join([str(item) for item in codewords.values()])
    Nb = calculate_nb(head_bits, 8, len(lz_dict.items()))
    print("N = " + str(N) + "\n" + '-' * len("N = " + str(N)))
    print(
        "Generated sequence: " + generated_sequence + '\n' + "Encoded
sequence: " + encoded_sequence + '\n' + '-' * len(
            "Encoded sequence: " + encoded_sequence))
    print("# of phrases: " + str(len(lz_dict.items())))
    print("# of bits per phrase: " + str(head_bits + 8) + ' -> [' +
str(head_bits) + ' + 8]')
    print("Nb: " + str(Nb) + ' -> [' + str(head_bits + 8) + ' * ' +
str(len(lz_dict.items())) + ']')
    print("-"*len("Nb: " + str(Nb) + ' -> [' + str(head_bits + 8) + ' * ' +
str(len(lz_dict.items())) + ']') + '\n')
    summary_output += str(N).rjust(5) + "\t" + str(Nb).rjust(5) + "\t\t" +
str(
        "{:.2f}".format(round((Nb / (N * 8)) * 100, 2))) + "%\t\t\t" +
str("{:.2f}".format(Nb / N)) + "\n"
print(summary_output)
```

5.5. Huffman Code

```
import heapq
import random
from LimpelZiv import *


class Node:
    def __init__(self, symbol, frequency):
        self.symbol = symbol
        self.frequency = frequency
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.frequency < other.frequency

def build_huffman_tree(symbols, probabilities):
    # Step 2: Build the Huffman tree
    heap = []
    for symbol, probability in zip(symbols, probabilities):
        node = Node(symbol, probability)
        heapq.heappush(heap, node)

    while len(heap) > 1:
        node1 = heapq.heappop(heap)
        node2 = heapq.heappop(heap)
        merged_frequency = node1.frequency + node2.frequency
        merged_node = Node(None, merged_frequency)
        merged_node.left = node1
        merged_node.right = node2
        heapq.heappush(heap, merged_node)

    return heap[0]

def build_codewords(node, current_code, codewords):
    if node.symbol:
        codewords[node.symbol] = current_code
        return

    build_codewords(node.left, current_code + "0", codewords)
    build_codewords(node.right, current_code + "1", codewords)

print("Huffman part" + "\n" + "-"*len("Huffman part") + "\n")

# Step 1: Define the symbols and their respective probabilities
symbols = ['a', 'b', 'c', 'd']
probabilities = [0.4, 0.3, 0.2, 0.1]

# Step 2: Build the Huffman tree
huffman_tree = build_huffman_tree(symbols, probabilities)

# Step 3: Build the codewords dictionary
codewords = {}
build_codewords(huffman_tree, "", codewords)

# Step 4: Print the codewords
print("Huffman Codewords:")
for symbol in codewords:
```

```
    print(symbol + ": " + codewords[symbol])

# Step 5: Calculate average bits per character
average_bits_per_character = sum(probabilities[symbols.index(symbol)] *
len(codewords[symbol]) for symbol in codewords)

print("Average bits per character (Huffman):", average_bits_per_character)

# Step 6: Generate 100 random symbols
random_symbols = random.choices(symbols, probabilities, k=100)
print(random_symbols)
# Step 7: Calculate the number of bits needed using ASCII code
ascii_bits = 8 * len(random_symbols)

print("Number of bits needed using ASCII code:", ascii_bits)

# Step 8: Encode the sequence using Huffman coding
encoded_sequence = ''.join(codewords[symbol] for symbol in random_symbols)

# Step 9: Calculate the size of the encoded sequence (NB)
encoded_bits = len(encoded_sequence)
encoded_bytes = (encoded_bits + 7) // 8  #The addition of 7 in the expression
(encoded_bits + 7) // 8 is to ensure that any remaining bits, after dividing
by 8, are properly accounted for.

print("Size of encoded sequence (NB):", encoded_bits, "bits (",
encoded_bytes, "bytes)")

# Step 10: Calculate the compression ratio NB/(8*N)
compression_ratio = encoded_bits / (8 * len(random_symbols))

print("Compression ratio (NB/(8*N)):", compression_ratio)

# Step 11: Calculate the number of bits per symbol (NB/N)
bits_per_symbol = encoded_bits / len(random_symbols)

print("Number of bits per symbol (NB/N):", bits_per_symbol)

# ------------------------------------------------------
print("\nLimpel-ziv part" + "\n" + "-"*len("Limpel-ziv part") + "\n")
N = 100
lz_dict = {}
generated_sequence = random_symbols
parse_sequence(generated_sequence, lz_dict)
head_bits = math.ceil(math.log2(len(lz_dict.items())))
codewords = encode_phrases(head_bits, lz_dict)
encoded_sequence = ''.join([str(item) for item in codewords.values()])
Nb = calculate_nb(head_bits, 8, len(lz_dict.items()))
print("Size of encoded sequence (NB):", Nb, "bits")
print("Compression ratio (NB/(8*N)):", str("{:.2f}".format(round((Nb / (N *
8)) * 100, 2))))
print("Number of bits per symbol (NB/N):", str("{:.2f}".format(Nb / N)))
```