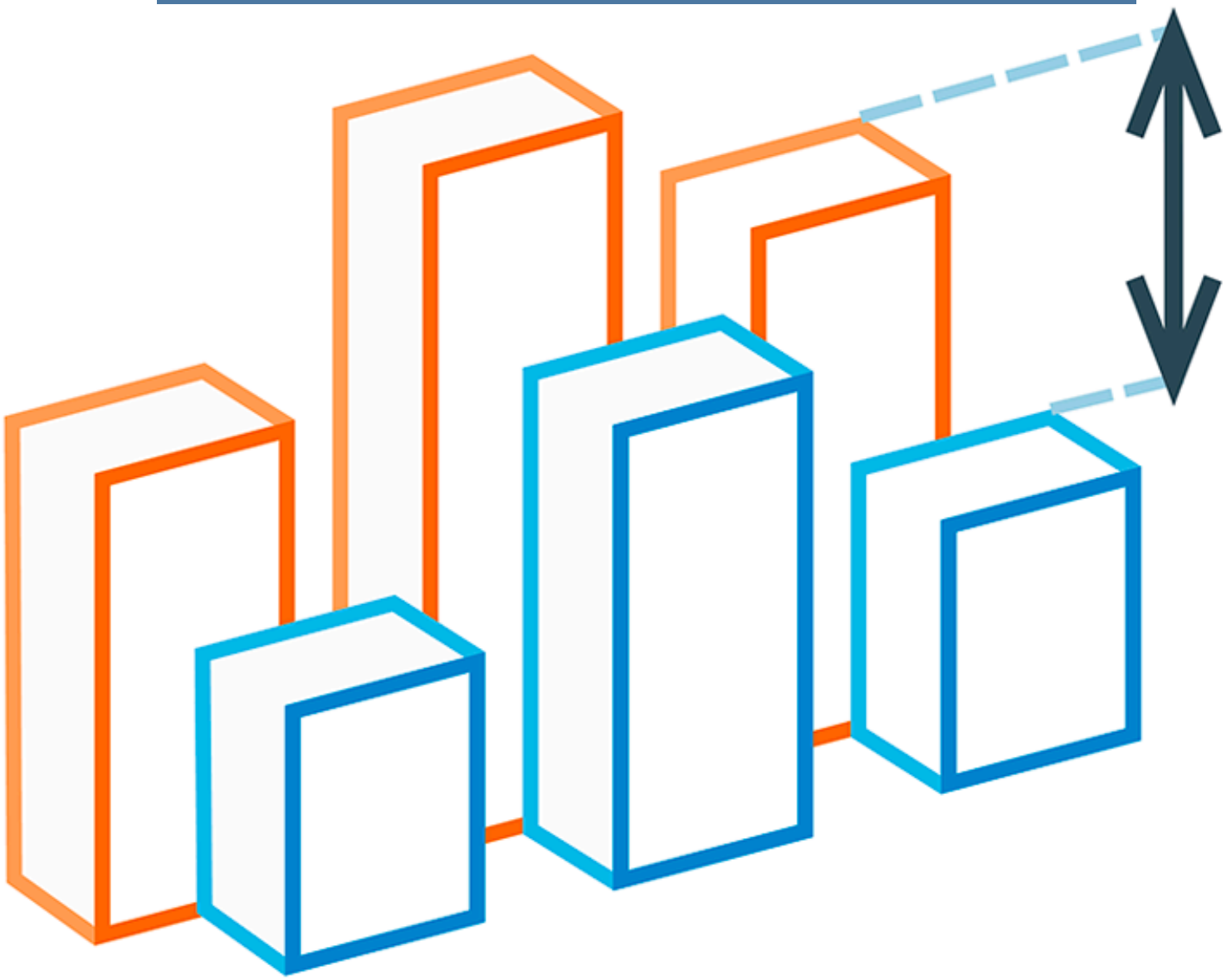
Data Compression



Prepared to :

DR. Mahmoud Gamal

ENG. Hossam El-Sokkary

01 سلمى عفيفي علي
20221453713

02 مريم علي حسن موسى
20221462924

03 امنية نصر عبد العاطي
20221457157

04 هدى أحمد السيد حافظ
20221442691

05 نادين السيد السيد احمد
20221377541

Libraries and Probability calculation

Step 1: Import Libraries needed:

This code imports modules necessary for a GUI application using Tkinter, including message boxes for user interaction, mathematical operations, heap queue algorithms, and counting elements in collections

```
import tkinter as tk
from tkinter import messagebox
import math
import heapq
from collections import Counter
```

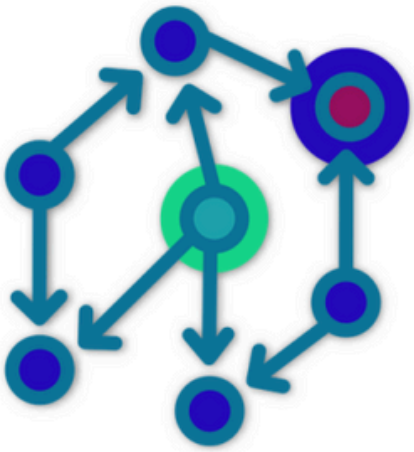
- Tkinter: Used for GUI creation.
- Messagebox: Displays dialog boxes for user interaction.
- Math Module: Provides mathematical functions.
- Heapq Module: Implements heap queue algorithms.
- Counter Class: Counts occurrences of elements in collections.

Step 2: Calculate the Probability:

This function calculates the probability of each character occurring in a given text. It initializes an empty dictionary to store character probabilities, counts occurrences of each character, calculates their probabilities, and returns the resulting dictionary.

```
def calculate_probability(text):
    probabilities = {}
    total_chars = len(text)
    for char in text:
        if char in probabilities:
            probabilities[char] += 1
        else:
            probabilities[char] = 1
    for char, count in probabilities.items():
        probabilities[char] = count / total_chars
    return probabilities
```

Huffman coding



Step 1: Define the HuffmanNode class :

The algorithm starts by defining the HuffmanNode class, which represents a node in the Huffman tree. Each node contains a character, its frequency, and references to its left and right child nodes. The class also defines a comparison method to compare nodes based on their frequencies.

```
class HuffmanNode:
    def __init__(self, char, frequency):
        self.char = char
        self.frequency = frequency
        self.left = None
        self.right = None

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

Step 2: Calculate the entropy :

The algorithm calculates the entropy of the input text by using the calculate_entropy function. It takes a dictionary of probabilities as input and applies the entropy formula: entropy = Σ(p * log2(1/p)) to compute the entropy value.

```
def calculate_entropy(probabilities):
    entropy = 0
    for prob in probabilities.values():
        entropy += prob * math.log2(1 / prob)
    return entropy
```

Step 3: Build the Huffman tree :

The algorithm builds the Huffman tree using the build_huffman_tree function takes a dictionary of probabilities as input. It initializes a priority queue (priority_queue) with HuffmanNode objects, one for each character and its frequency. The priority queue is then converted into a heap data structure. The function repeatedly extracts the two nodes with the lowest frequencies from the priority queue, merges them into a new node, and inserts the new node back into the priority queue. This process continues until only one node remains in the priority queue, which represents the root of the Huffman tree. The function returns the root node of the Huffman tree.

```
def build_huffman_tree(probabilities):
    priority_queue = [HuffmanNode(char, freq) for char, freq in probabilities.items()]
    heapq.heapify(priority_queue)
    while len(priority_queue) > 1:
        left = heapq.heappop(priority_queue)
        right = heapq.heappop(priority_queue)
        merged = HuffmanNode(None, left.frequency + right.frequency)
        merged.left = left
        merged.right = right
        heapq.heappush(priority_queue, merged)
    return priority_queue[0]
```

Step 4: Generate Huffman codes :

The algorithm generates Huffman codes for each character in the input text using the generate_huffman_codes function. It traverses the Huffman tree recursively and assigns a binary code to each character based on its position in the tree. The function maintains a current code during the traversal, appending "0" for a left branch and "1" for a right branch. When a leaf node (character node) is reached, the code is stored in a dictionary with the character as the key.

```
def generate_huffman_codes(root, current_code, codes):
    if root is None:
        return
    if root.char is not None:
        codes[root.char] = current_code
        return
    generate_huffman_codes(root.left, current_code + "0", codes)
    generate_huffman_codes(root.right, current_code + "1", codes)
```

Step 5: Encode the text using Huffman codes :

The algorithm encodes the input text using the generated Huffman codes. It iterates over each character in the text and checks if there is a corresponding code in the Huffman codes dictionary. If a code exists, it appends the code to the encoded text. If a character does not have a code, an error is raised, indicating that the character cannot be encoded using Huffman coding.

```
def encode_huffman(text, codes):
    encoded_text = ""
    for char in text:
        if char in codes:
            encoded_text += codes[char]
        else:
            # If the character is not in the Huffman codes, raise an error or handle it accordingly
            raise ValueError(f"Character '{char}' cannot be encoded using Huffman coding.")
    return encoded_text
```

Step 6: Decode the Huffman-encoded text :

The algorithm decodes the Huffman-encoded text using the Huffman tree. It iterates over each bit in the encoded text and follows the corresponding path in the Huffman tree. When a leaf node (character node) is reached, the corresponding character is added to the decoded text. The algorithm continues this process until all bits in the encoded text are processed.

```
def decode_huffman(encoded_text, huffman_tree):
    decoded_text = ""
    current_node = huffman_tree
    for bit in encoded_text:
        if bit == "0":
            current_node = current_node.left
        else:
            current_node = current_node.right
        if current_node.char is not None:
            decoded_text += current_node.char
            current_node = huffman_tree
    return decoded_text
```

Step 7: Calculate the average length of the encoded text :

The algorithm calculates the average length of the encoded text by multiplying each character's probability with the length of its Huffman code. This is done using the calculate_average_length function.

```
def calculate_average_length(probabilities, codes):
    average_length = sum(prob * len(codes[char]) for char, prob in probabilities.items())
    return average_length
```

Step 8: Calculate the efficiency of the compression :

The algorithm calculates the efficiency of the compression by dividing the entropy by the average length and multiplying by 100. This is done using the calculate_efficiency function.

```
def calculate_efficiency(average_length, entropy):
    return (entropy / average_length) * 100
```

Step 9: Process the Huffman text :

The algorithm processes the Huffman text based on the specified operation (compress or decompress). It retrieves the input text and checks for empty input. It then calculates the probabilities of each character in the text and builds the Huffman tree. Huffman codes are generated, and the input text is encoded using these codes. If the operation is compression, the encoded text is decoded as well.

```
def process_huffman_text(operation):
    text = get_input_text()
    if not text:
        messagebox.showwarning("Warning", "Please enter some text.")
        return

    probabilities = calculate_probability(text)

     # Check if there are characters in the text that cannot be compared using Huffman coding
    invalid_chars = [char for char in text if char not in probabilities]
    if invalid_chars:
        messagebox.showwarning("Warning", f"The following characters cannot be compared using Huffman coding: {', '.join(invalid_chars)}")
        return

    huffman_tree = build_huffman_tree(probabilities)
    huffman_codes = {}
    generate_huffman_codes(huffman_tree, "", huffman_codes)
    encoded_text = encode_huffman(text, huffman_codes)

    if operation == "compress":
        decoded_text = decode_huffman(encoded_text, huffman_tree)
        additional_info = f"Decoded text: {decoded_text}\n"
    else:
        additional_info = ""
```

Step 10: Calculate compression metrics :

The algorithm calculates various metrics such as the number of bits before and after encoding, reduction in bits, compression ratio, compression ratio percentage, space usage ratio, entropy, average length, and efficiency.

```
bits_before, bits_after = len(text) * 8, len(encoded_text)
reduction_in_bits = bits_before - bits_after
compression_ratio = bits_before / bits_after
compression_ratio_percentage = (1 - (bits_after / bits_before)) * 100 if bits_before != 0 else 0
space_usage_ratio = (bits_after / bits_before) * 100
entropy = calculate_entropy(probabilities)
average_length = calculate_average_length(probabilities, huffman_codes)
efficiency = calculate_efficiency(average_length, entropy)
efficiency_message = "Compression was effective." if bits_after < bits_before else "Compression was not effective."
```

Step 11: Display the output :

The algorithm constructs an output string containing all the relevant information, including the original text, encoded text, additional information (if applicable), compression metrics, probabilities, entropy, average length, and efficiency. The output text is updated in the output display.

```
    output_text = f"Original text: {text}\n"
    output_text += f"Encoded text: {encoded_text}\n"
    output_text += additional_info
    output_text += f"Bits before encoding: {bits_before}\n"
    output_text += f"Bits after encoding: {bits_after}\n"
    output_text += f"Reduction in bits: {reduction_in_bits}\n"
    output_text += f"Compression ratio : {compression_ratio}\n"
    output_text += f"Compression ratio percentage(%): {compression_ratio_percentage:.2f}%\n"
    output_text += f"Space usage ratio (%): {space_usage_ratio:.2f}%\n"
    output_text += "Probability of occurrence for each character:\n"
    for char, prob in probabilities.items():
        output_text += f"- {char}: {prob}\n"
    output_text += f"Entropy: {entropy}\n"
    output_text += f"Average length: {average_length}\n"
    output_text += f"Efficiency of this message: {efficiency}\n"
    output_text += f"Efficiency message: {efficiency_message}\n"

    update_output_text(output_text)

def get_input_text():
    return text_input.get("1.0", tk.END).strip()

def update_output_text(text):
    text_output.delete("1.0", tk.END)
    text_output.insert("1.0", text)
```

Golomb Coding

Step 1: Global Variable Declaration:

This line initializes the efficiency_golomb variable as None. This variable will be used to store the efficiency of Golomb-Rice encoding.

```
efficiency_golomb = None  # Define efficiency as a global variable
```

Step 2: Golomb-Rice Encoding Function :

This function takes a sequence of symbols and an integer parameter M and encodes the sequence using Golomb-Rice encoding. Golomb-Rice encoding is a universal entropy coding technique suitable for integer-valued data with a geometric distribution of small values.

Implementation:

- It first generates a run-length encoding (RLE) of the input sequence to reduce redundancy.
- For each symbol and its count in the RLE sequence, it computes the Golomb-Rice codeword for the count using golomb_rice_encode_count function.
- It returns the RLE sequence and a dictionary of codewords.

```
def golomb_rice_encode(sequence, M):
    # Generate RLE from the sequence
    rle_sequence_golomb = []
    char_sequence = []
    count = 1
    for i in range(1, len(sequence)):
        if sequence[i] == sequence[i - 1]:
            count += 1
        else:
            rle_sequence_golomb.append((sequence[i-1], count))  # Storing both symbol and count
            count = 1
    rle_sequence_golomb.append((sequence[-1], count))  # Storing the last symbol and its count

    if not rle_sequence_golomb:
        raise ValueError("Empty sequence.")

    codewords = {}
    for char, count in rle_sequence_golomb:
        # Generate Golomb-Rice codeword for count
        count_codeword = golomb_rice_encode_count(count, M)
        # Store the Golomb-Rice codeword along with the character
        codewords[(char, count)] = count_codeword

    # Return the encoded sequence and the codewords
    return rle_sequence_golomb, codewords
```

Step 3: Golomb-Rice Decoding Function:

This function decodes a Golomb-Rice encoded sequence using the provided codewords and the parameter M.

Implementation:

- Retrieve the Golomb-Rice codeword from the dictionary.
- Decode the count using golomb_rice_decode_count.
- Append the decoded symbols to reconstruct the original sequence.

```
def golomb_rice_decode(encoded, codewords, M):
    decoded_sequence = []
    for char, count in encoded:
        # Decode the count using the Golomb-Rice codeword
        count_codeword = codewords[(char, count)]
        decoded_value = golomb_rice_decode_count(count_codeword, M)
        # Append the decoded value to the sequence
        decoded_sequence.extend([char] * decoded_value)
    # Return the decoded sequence
    return decoded_sequence
```

Step 4: Golomb-Rice Encoding of Counts:

This function encodes a single count value using Golomb-Rice encoding with the parameter M.

Implementation:

- Divide the count by M to get quotient and remainder.
- Encode the quotient in unary and the remainder in binary.
- Concatenate the unary and binary representations to form the Golomb-Rice codeword.

```
def golomb_rice_encode_count(value, M):
    quotient = value // M
    remainder = value % M

    # Encode quotient in unary; a sequence of 1s followed b
    unary = '1' * quotient + '0'

    # Encode remainder in binary. The length of the binary
    binary_length = math.ceil(math.log2(M))
    binary = format(remainder, f'0{binary_length}b')

    return unary + binary
```

Step 5: Golomb-Rice Dencoding of Counts:

This function decodes a Golomb-Rice encoded count value using the provided parameter M.

Implementation:

- Extract the quotient from the unary representation.
- Skip over the separator '0'.
- Extract the binary part and convert it to an integer.
- Reconstruct the original count using the quotient and remainder.

```
def golomb_rice_decode_count(encoded, M):
    decoded_value = 0
    # Extract the quotient
    while encoded[0] == '1':
        decoded_value += 1
        encoded = encoded[1:]
    # Skip over the separator '0'
    encoded = encoded[1:]
    # Extract the binary part
    remainder = ''
    while len(remainder) < math.ceil(math.log2(M)):
        remainder += encoded[0]
        encoded = encoded[1:]
    # Convert the binary part to integer
    remainder_value = int(remainder, 2)
    # Decode the value using the quotient and remainder
    decoded_value = decoded_value * M + remainder_value
    return decoded_value
```

Step 6: Average Length Calculation:

This function calculates the average length of Golomb-Rice encoded symbols based on their probabilities. It takes three inputs: rle_sequence_golomb (the run-length encoded sequence), codewords (dictionary mapping symbols to Golomb-Rice codewords), and probabilities (dictionary mapping symbols to their probabilities).

```
def calculate_average_length_golomb(rle_sequence_golomb, codewords, probabilities):
    total_length = 0

    for char, count in rle_sequence_golomb:
        probability = probabilities[char]
        codeword_length = len(codewords[(char, count)])
        total_length += probability * codeword_length

    return total_length
```

Step 7: Input Retrieval:

The code segment manages the initialization and validation process before Golomb encoding. It starts by clearing any existing content on the canvas. Subsequently, it collects user input, ensuring the sequence is stripped of any leading or trailing whitespace. Additionally, it acquires the parameter M, validating its positivity.

```
def process_golomb_text(operation):
    global efficiency_golomb
    canvas.delete("all")
    try:
        sequence = text_input.get("1.0", tk.END).strip()
        M = int(entry_M.get())
        if M <= 0:
            raise ValueError("M should be a positive integer.")
```

Step 8: Bits Calculation:

the code snippet generates a run-length encoding (RLE) of the input sequence using Golomb-Rice encoding, and then checks if the resulting RLE sequence is empty.
Then, it calculates the number of bits before and after Golomb-Rice encoding. It determines the number of bits before encoding based on whether the input sequence consists only of '0's and '1's or includes other characters. For the bits after encoding, it computes the total number of bits needed to represent the Golomb-Rice encoded sequence by summing the lengths of Golomb-Rice codewords for each symbol and its count in the run-length encoded Golomb sequence. Additionally, it concatenates the Golomb-Rice codewords to form the encoded code. These calculations help evaluate the compression achieved by Golomb-Rice encoding.

```
    # Generate RLE from the sequence
    rle_sequence_golomb, codewords = golomb_rice_encode(sequence, M)

    if not rle_sequence_golomb:
        raise ValueError("Empty sequence.")

   # Calculate bits before encoding
    if all(char in '01' for char in sequence):
        original_bits = len(sequence)  # Each character is considered as one bit
    else:
        original_bits = len(sequence) * 8  # Each character is considered as one byte (eight bits)

    # Calculate bits after encoding
    encoded_bits = sum(len(codewords[(char, count)]) for char, count in rle_sequence_golomb)
    encoded_code = ''.join(codewords[(char, count)] for char, count in rle_sequence_golomb)
```

Step 9: Encoding or Decoding:

Depending on the specified operation:

- For compression:
  - Generate the Golomb-Rice encoded sequence and codewords.
- For decompression:
  - Decode the Golomb-Rice encoded sequence.

```
if operation == "compress":
    decoded_text = golomb_rice_decode(rle_sequence_golomb, codewords, M)
    decoded_sequence = ''.join(decoded_text)
    additional_info = f"Decoded text: {decoded_sequence}\n"
else:
    additional_info = ""
```

Step 10: Compression Metrics Calculation:

Calculate compression metrics such as compression ratio, probabilities, entropy, average length, and efficiency.

```
# Calculate compression ratio
compression_ratio = (original_bits/encoded_bits)*100

# Calculate probabilities
probabilities = calculate_probability(sequence)

# Calculate entropy
# Calculate entropy of the original text
entropy = calculate_entropy(probabilities)

# Calculate average length
average_length = calculate_average_length_golomb(rle_sequence_golomb, codewords, probabilities)

# Calculate efficiency
efficiency_golomb = entropy / average_length * 100
```

Step 11: Output Preparation:

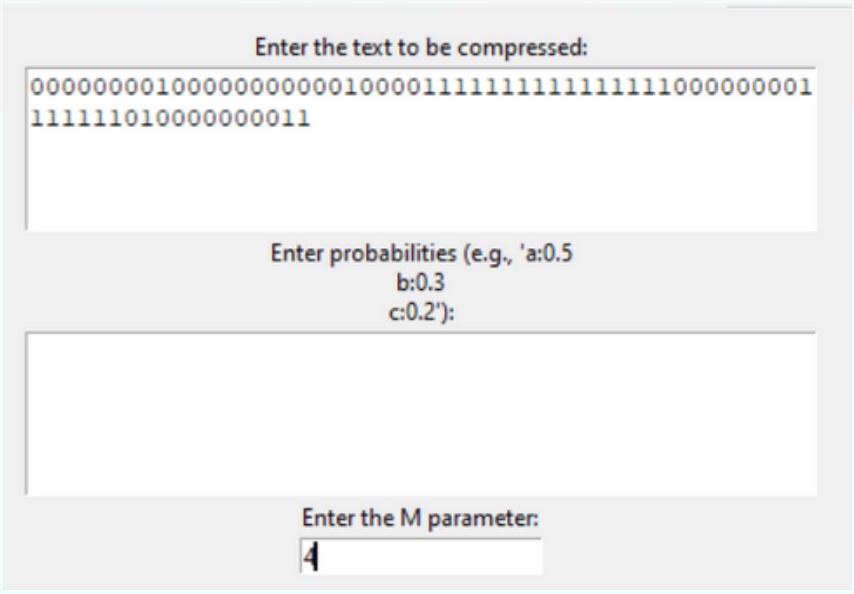
```
        output_text = f"Original sequence: {sequence}\n"
        output_text += f"Encoded code:\n{encoded_code}\n"
        output_text += additional_info
        output_text += f"Character Probabilities:\n"
        # Calculate probabilities of characters in the original text
        for char, prob in probabilities.items():
            output_text += f"- {char}: {prob}\n"
        output_text += f"Codewords:\n"
        for (char, count), codeword in codewords.items():
            output_text += f"- {char} ({count}): {codeword}\n"
        output_text += f"Bits before encoding: {original_bits}\n"
        output_text += f"Bits after encoding: {encoded_bits}\n"
        output_text += f"Compression ratio: {compression_ratio:.2f}%\n"
        output_text += f"Entropy: {entropy:.2f} bits/character\n"
        output_text += f"Average Length: {average_length:.2f} bits/character\n"
        output_text += f"Efficiency: {efficiency_golomb:.2f}%\n"

        text_output.delete("1.0", tk.END)
        text_output.insert("1.0", output_text)

    except ValueError as e:
        messagebox.showerror("Error", str(e))
```

Prepare the output text with relevant information

Output




```
Original sequence: 000000001000000000001000011111111111111110000000011111111010000000011
Encoded code:
110000011101100110001111000110001011001001110000010
Decoded text: 000000001000000000001000011111111111111110000000011111111010000000011
Character Probabilities:
- 0: 0.5882352941176471
- 1: 0.4117647058823529
Codewords:
- 0 (8): 11000
- 1 (1): 001
- 0 (11): 11011
- 0 (4): 1000
- 1 (16): 1111000
- 1 (7): 1011
```

```
- 0 (1): 001
- 1 (2): 010
Bits before encoding: 68
Bits after encoding: 50
Compression ratio: 1.36
Entropy: 0.98 bits/character
Average Length: 25.35 bits/character
Efficiency: 3.86%
```

lzw

Step 1: intialize the dictionary:

- Initialize dictionary_size to 256, representing the number of ASCII characters.
- Create a dictionary where each ASCII character is mapped to its ASCII value.

```
# Initialize dictionary with ASCII characters
dictionary_size = 256
dictionary = {chr(i): i for i in range(dictionary_size)}
```

Step 2: Compression Process(encoding function) :

result: A list to store the compressed codes generated during compression.
current_string: A string variable to keep track of the current substring being considered.
Iterate through each character (char) in the input text (text).
Construct a new string (new_string) by appending the current character to the current_string.
Check if the constructed substring (new_string) is already present in the dictionary:
If it is present, update the current_string to include the new character.
If it is not present:
Append the code corresponding to the current substring (current_string) in the dictionary to the result.
Add the new substring (new_string) to the dictionary with the next available index (dictionary_size).
Increment dictionary_size to maintain the uniqueness of dictionary entries.
Reset the current_string to the current character.
If there are remaining characters in current_string after the loop, append their corresponding code to the result.

```
result = []
current_string = ""
for char in text:
    new_string = current_string + char
    if new_string in dictionary:
        current_string = new_string
    else:
        result.append(dictionary[current_string])
        # Add new string to dictionary
        dictionary[new_string] = dictionary_size
        dictionary_size += 1
        current_string = char
if current_string:
    result.append(dictionary[current_string])
return result
```

Step 3: calc entropy function:

- initialize **entropy** to 0.
- Iterate through each probability in the **probabilities** dictionary.
- For each probability, calculate the term $-p \cdot \log2(p)$-p · log2(p), where pp is the probability of the event.
- Add each calculated term to the **entropy**.

```
def calculate_entropy(probabilities):
    entropy = 0
    for prob in probabilities.values():
        entropy -= prob * math.log2(prob)
    return entropy
```

Step 4: Compress text function:

- The function compresses the input text using the LZW algorithm and evaluates the efficiency of compression by calculating entropy and average length. It then provides insights into compression effectiveness by displaying compression statistics and other relevant information in the GUI text box.

implementation

- **Input**:
  - Retrieves the input text from a GUI text box (**text_input**).
- **Process**:
  a. **Input Validation**:
     - Checks if there is input text. If not, it displays a warning message and stops the compression process.
  b. **Compression**:
     - Uses the **LZW_compress** function to compress the input text.
  c. **Entropy Calculation**:
     - Calculates the probabilities of characters in the original text.
     - Calculates the entropy of the original text using the **calculate_entropy** function.
  d. **Average Length Calculation**:
     - Calculates the average length of the compressed LZW output. This is typically done by multiplying the number of compressed codes by the assumed number of bits per code (e.g., 8 bits).
  e. **Efficiency Calculation**:
     - Calculates the efficiency of the compressed output using the **calculate_efficiency** function.
  f. **Compression Ratio Calculation**:
     - Calculates the compression ratio, which is the ratio of bits before and after encoding. This helps in evaluating the effectiveness of compression.
  g. **Output Construction**:
     - Constructs an output text containing details about the original text, compressed text, compression statistics (bits before and after encoding, compression ratio), probabilities of characters, entropy, average length, and efficiency.

```
def compress_text_lzw():
    input_text = text_input.get("1.0", tk.END).strip()
    if not input_text:
        messagebox.showwarning("Warning", "Please enter some text.")
        return

    compressed_text = LZW_compress(input_text)

    # Calculate probabilities of characters in the original text
    probabilities = calculate_probability(input_text)

    # Calculate entropy of the original text
    entropy = calculate_entropy(probabilities)

    # Calculate average length of the compressed LZW output
    average_length = len(compressed_text) *8  # Assuming each code takes 12 bits

    # Calculate efficiency of the compressed LZW output
    efficiency = calculate_efficiency(average_length, entropy)

    bits_before = len(input_text) * 8
    bits_after = average_length
    compression_ratio = bits_before / bits_after
```

Step 5: display the out put:

```
output_text = f"Original text: {input_text}\n"
output_text += f"Compressed text: {compressed_text}\n"
output_text += f"Bits before encoding: {bits_before}\n"
output_text += f"Bits after encoding: {bits_after}\n"
output_text += f"Compression ratio (%): {compression_ratio}\n"
output_text += "Probability of occurrence for each character:\n"
for char, prob in probabilities.items():
    output_text += f"- {char}: {prob}\n"
output_text += f"Entropy: {entropy}\n"
output_text += f"Average length: {average_length}\n"
output_text += f"Efficiency of this message: {efficiency}\n"
```

Prepare the output text with relevant information

Arithmetic Coding

Step 1: ArithmeticCoding class:

- This class implements the arithmetic coding algorithm for data compression.
- In the constructor __init__, it takes a dictionary of symbol probabilities as input and initializes various attributes such as low_range, high_range, and cumulative_prob.
- The initialize_ranges method calculates the low and high ranges for each symbol based on their probabilities.

```
class ArithmeticCoding:
    def __init__(self, probabilities):
        self.probabilities = probabilities
        self.low_range = {}
        self.high_range = {}
        self.cumulative_prob = {}
        self.initialize_ranges()

    def initialize_ranges(self):
        low = 0.0
        for symbol, prob in self.probabilities.items():
            self.low_range[symbol] = low
            self.high_range[symbol] = low + prob
            self.cumulative_prob[symbol] = low
            low += prob
```

Step 2: encode method:

- This method encodes a sequence of symbols using arithmetic coding.
- It iterates through each symbol in the sequence and updates the range based on the probabilities of the symbols.
- Finally, it returns the encoded value, which is the midpoint of the final range.

```
def encode(self, sequence):
    low = 0.0
    high = 1.0
    for symbol in sequence:
        range_width = high - low
        high = low + range_width * self.high_range[symbol]
        low = low + range_width * self.low_range[symbol]
    return (low + high) / 2
```

Step 3: decode method:

- This method decodes a compressed code back into the original sequence of symbols.
- It iteratively finds the symbol corresponding to the range containing the given code and updates the code and range for the next iteration.
- It raises a ValueError if the code is invalid.

```
def decode(self, code, length):
    sequence = []
    for _ in range(length):
        found_symbol = False
        for symbol, low in self.low_range.items():
            if low <= code < self.high_range[symbol]:
                sequence.append(symbol)
                range_width = self.high_range[symbol] - low
                code = (code - low) / range_width
                found_symbol = True
                break
        if not found_symbol:
            raise ValueError("Invalid compression code")
    return ''.join(sequence)
```

Step 4: encode_length method:

- This method calculates the length of the encoded code for a given symbol.
- It encodes the symbol using the encode method, converts the result to a string, and returns the length of this string.
- This method allows us to determine the number of bits required to represent the compressed code for a symbol.

```
def encode_length(self, symbol):
    # Implement logic to calculate the length of the encoded code for the symbol
    encoded_code = self.encode(symbol)
    # Example: return the length of the string representation of the float
    return len(str(encoded_code))
```

Step 5: get_probabilities function:

- This function retrieves symbol probabilities, possibly from user input.
- It parses the input and returns a dictionary of symbol-probability pairs.

```
def get_probabilities():
    probabilities = {}
    prob_input = text_probabilities.get("1.0", tk.END).strip()
    prob_lines = prob_input.split("\n")
    for line in prob_lines:
        symbol, prob = line.split(":")
        probabilities[symbol.strip()] = float(prob)
    return probabilities
```

Step 6: calculate_metrics function:

- This function calculates various metrics related to compression.
- It calculates original text size, compressed code size, compression ratio, entropy of the original text, average length of the encoded symbols, and efficiency.
- It relies on calculate_entropy_arith and calculate_average_length_arith functions to compute entropy and average length.

```
def calculate_metrics(original_text, compressed_code, arithmetic_coder):
    original_text_bits = len(original_text) * 8  # Assuming 8 bits per character
    compressed_bits = len(compressed_code)
    compression_ratio = (original_text_bits / compressed_bits) * 100
    probabilities = get_probabilities()
    entropy = calculate_entropy_arith(probabilities)

    average_length = calculate_average_length_arith(probabilities, arithmetic_coder)
    efficiency = entropy / average_length
    return original_text_bits, compressed_bits, compression_ratio, entropy, average_length, efficiency
```

Step 7: calculate_entropy_arith function:

- This function calculates the entropy of the original text using Shannon's entropy formula.
- It counts the frequency of each symbol in the text, calculates probabilities, and then computes entropy using the probabilities.

```
def calculate_entropy_arith(probabilities):
    # Calculate the entropy using the probabilities
    entropy = -sum(prob * math.log2(prob) for prob in probabilities.values())
    return entropy
```

Step 8:calculate_average_length_arith function:

- This function calculates the average length of the encoded symbols using the given probabilities and the encode_length method of ArithmeticCoding.
- It sums up the product of the length of each encoded symbol and its probability.

```
def calculate_average_length_arith(probabilities, arithmetic_coder):
    # Calculate the average length of the encoded symbols
    average_length = sum(arithmetic_coder.encode_length(symbol) * prob for symbol, prob in probabilities.items())
    return average_length
```

Step 9: compress_text_arithmetic function:

- This function compresses the input text using arithmetic coding.
- It retrieves symbol probabilities, initializes an instance of the ArithmeticCoding class, and then encodes the input text.
- It calculates various metrics using calculate_metrics and formats them into a human-readable output string.
- Finally, it updates the output area with the compression results.

```
def compress_text_arithmetic():
    input_text = text_input.get("1.0", tk.END).strip()
    if not input_text:
        messagebox.showwarning("Warning", "Please enter some text.")
        return

    probabilities = get_probabilities()

    arithmetic_coder = ArithmeticCoding(probabilities)
    compressed_code = str(arithmetic_coder.encode(input_text))  # Convert to string

    original_text_bits, compressed_bits, compression_ratio, entropy, average_length, efficiency = calculate_metrics(

    output_text = f"Original text: {input_text}\n"
    output_text += f"Compressed code: {compressed_code}\n"
    output_text += f"Original text bits: {original_text_bits}\n"
    output_text += f"Compressed bits: {compressed_bits}\n"
    output_text += f"Compression Ratio: {compression_ratio:.2f}%\n"
    output_text += f"Entropy: {entropy:.2f}\n"
    output_text += f"Average Length: {average_length:.2f}\n"
    output_text += f"Efficiency: {efficiency:.2f}\n"

    text_output.delete("1.0", tk.END)
    text_output.insert("1.0", output_text)
```

RLE Encoding

Step 1: encode_rle function

This function performs Run-Length Encoding (RLE) on a given text.

1. Initialize Variables: Initialize variables for tracking encoded text, character count, and previous character.
2. Edge Case Handling: Return default values if the input text is empty.
3. Iterate Through Text:
   - Iterate through each character in the input text.
   - If the current character is different from the previous one:
     - Append the count of the previous character and the character itself to the encoded text.
     - Reset the character count to 1 and update the previous character.
   - If the current character is the same as the previous one, increment the character count.
4. Encode Remaining Characters: Append the count of the last character encountered to the encoded text.
5. Calculate Compression Metrics: Calculate compression metrics such as bits before and after encoding, compression ratio, and efficiency.
6. Return Results: Return the encoded text and compression metrics.

```python
def encode_rle(text):
    encoded_text = ''
    count = 1
    prev_char = ''
    no_of_vec = 0
    max_count = 0
    if not text:
        return '', 0, 0, {}, 0, 0, 0

    for char in text:
        if char != prev_char:
            no_of_vec += 1
            if prev_char:
                encoded_text += str(count) + prev_char
            count = 1
            prev_char = char
        else:
            count += 1
        if max_count < count:
            max_count = count
    encoded_text += str(count) + prev_char

    bits_before = len(text) * 8
    bits_after = no_of_vec * ( 8 + (math.ceil(math.log(max_count+1, 2))) )
    compression_ratio = bits_before / bits_after
    probabilities = calculate_probability(text)
    entropy = calculate_entropy(probabilities)
    average_length = calculate_average_length_rle(probabilities, encoded_text)
    efficiency = calculate_efficiency(average_length, entropy)

    return encoded_text, bits_before, bits_after, compression_ratio, probabilities, entropy, average_length, efficiency
```

Step 2: calculate_average_length_rle function

This function calculates the average length of the encoded text in Run-Length Encoding (RLE). In RLE, each character is followed by its count, so the average length is simply the sum of the lengths of the encoded characters.

```
def calculate_average_length_rle(probabilities, encoded_text):
    # In RLE, each character is followed by its count, so the average length
    # is the sum of the lengths of the encoded characters.
    average_length = len(encoded_text)
    return average_length
```

Step 3: decode_rle function

This function decodes a text that has been encoded using RLE. It iterates through the encoded text, rebuilding the original text based on the counts of consecutive characters.

```
def decode_rle(encoded_text):
    decoded_text = ''
    count = ''
    for char in encoded_text:
        if char.isdigit():
            count += char
        else:
            decoded_text += char * int(count)
            count = ''
    return decoded_text
```

Step 4: compress_text_rle function

This function is likely part of a GUI-based application. It retrieves the input text from a text input widget, then calls the encode_rle function to compress the text. It then decodes the compressed text using the decode_rle function. Finally, it displays various compression metrics along with the original, encoded, and decoded texts in a text output widget.

```
def compress_text_rle():
    input_text = text_input.get("1.0", tk.END).strip()
    if not input_text:
        messagebox.showwarning("Warning", "Please enter some text.")
        return

    encoded_text, bits_before, bits_after, compression_ratio, probabilities, entropy, average_length, efficiency= encode_rle(input_text)
    decoded_text = decode_rle(encoded_text)

    output_text = f"Original text: {input_text}\n"
    output_text += f"Encoded text: {encoded_text}\n"
    output_text += f"Decoded text: {decoded_text}\n"
    output_text += f"Bits before encoding: {bits_before}\n"
    output_text += f"Bits after encoding: {bits_after}\n"
    output_text += f"Compression ratio (%): {compression_ratio}\n"
    output_text += "Probability of occurrence for each character:\n"
    for char, prob in probabilities.items():
        output_text += f"- {char}: {prob}\n"
    output_text += f"Entropy: {entropy}\n"
    output_text += f"Average length: {average_length}\n"
    output_text += f"Efficiency of this message: {efficiency}\n"

    text_output.delete("1.0", tk.END)
    text_output.insert("1.0", output_text)

    # Clear the canvas
```

The Output

Compression/Decompression Details:

```
Original text: nnnnnnnnaaaaaddddddddd
Encoded text: 8n5a9d
Decoded text: nnnnnnnnaaaaaddddddddd
Bits before encoding: 176
Bits after encoding: 36
Compression ratio (%): 4.888888888888889
Probability of occurrence for each character:
- n: 0.36363636363636365
- a: 0.22727272727272727
- d: 0.4090909090909091
Entropy: 1.544024096481951
Average length: 6
Efficiency of this message: 0.2573373494136585
```

Optimal Technique

Choosing Optimal Technique

The function choose_optimal_technique() is designed to determine the most efficient data compression technique for a given input text. It analyzes the input text using Huffman Encoding and Run Length Encoding (RLE), calculating metrics such as probabilities, entropy, average length, and efficiency for both techniques. Then, it compares these metrics with other encoding techniques like Golomb-Rice Encoding, Arithmetic Encoding, and LZW Compression. Finally, it recommends the technique with the highest efficiency for the given text.

```
def choose_optimal_technique():
    input_text = text_input.get("1.0", tk.END).strip()
    if not input_text:
        messagebox.showwarning("Warning", "Please enter some text.")
        return

    # Calculate probabilities and other metrics for Huffman encoding
    probabilities_huffman = calculate_probability(input_text)
    huffman_tree = build_huffman_tree(probabilities_huffman)
    huffman_codes = {}
    generate_huffman_codes(huffman_tree, "", huffman_codes)
    average_length_huffman = calculate_average_length(probabilities_huffman, huffman_codes)
    entropy_huffman = calculate_entropy(probabilities_huffman)
    efficiency_huffman = calculate_efficiency(average_length_huffman, entropy_huffman)

    # Calculate probabilities and other metrics for RLE encoding
    _, _, _, _, probabilities_rle, entropy_rle, average_length_rle, efficiency_rle = encode_rle(input_text)

    # Choose the optimal technique based on efficiency
    optimal_technique = max(
        ("Golomb-Rice Encoding", efficiency_golomb),
        ("Huffman Encoding", efficiency_huffman),
        ("Run Length Encoding (RLE)", efficiency_rle),
        ("Arithmetic Encoding", efficiency_arith),
        ("LZW Compression", efficiency_lzw),
        key=lambda x: x[1]
    )

    messagebox.showinfo("Optimal Technique", f"{optimal_technique[0]} is recommended for this text.")
```

Thank You