|  |  |
| --- | --- |
|  |  |
| DATA COMPRESSION |  |
|  |  |
|  |  |
|  | Prof. Mahmoud Gamal Eng. Mariem Nagi |

**Project Overview**

Data compression is a vital process for reducing data size while preserving its integrity. This project explores four compression algorithms: Run-Length Encoding (RLE), Arithmetic Coding, Huffman Coding, and Golomb Coding. Each algorithm is implemented in Python, showcasing their unique efficiencies and use cases. A graphical user interface (GUI) was developed to facilitate compression and decompression operations, making the program accessible and user-friendly. The GUI allows users to input data, select algorithms, and view results seamlessly. This report highlights the algorithms' implementation, their comparative performance, and the practical utility of the GUI.

**Code Details**

**Step 1 : Importing the required libraries**

**A white background with black text

Description automatically generated**

Why we used those libraries ?

1. **tkinter:**
   * **Used to create the graphical user interface (GUI) for the program.**
   * **Provides a robust and easy-to-use framework for building windows, buttons, input fields, and other UI elements.**
   * **Includes modules like messagebox, filedialog, and ttk to handle user interactions, file selection, and enhanced styling.**
2. **heapq:**
   * **Utilized for implementing priority queues, which are critical for algorithms like Huffman Coding.**
   * **Efficiently manages the construction of Huffman trees by maintaining the smallest elements at the top of the heap.**
3. **math:**
   * **Provides mathematical functions such as logarithms and ceiling operations, which are essential for algorithms like Golomb Coding and Arithmetic Coding.**
   * **Ensures accuracy in computations required for compression techniques.**
4. **collections.Counter:**
   * **Facilitates counting occurrences of symbols or characters in the input data.**
   * **Simplifies the process of frequency analysis, which is a fundamental step in Huffman Coding and other compression methods.**

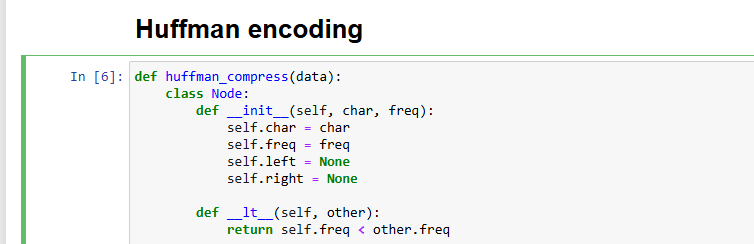
**These libraries were chosen for their efficiency, compatibility with Python, and ability to simplify the implementation of complex algorithms and user interactions.**

**Step 2 : Compression & Decompression Codes**

**2.1 Huffman Compression:**

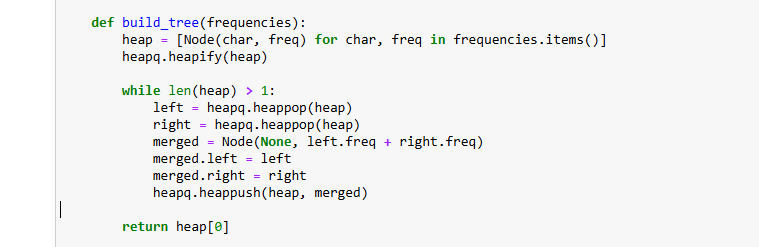
**Node Class:**

* **A custom class is defined to represent the nodes of a Huffman tree.**
* **Each node stores a character (char) and its frequency (freq) along with pointers to its left and right children (left and right).**
* **The \_\_lt\_\_ method is overridden to enable comparison based on frequency, which is essential for constructing the priority queue.**

****

**Building the Huffman Tree:**

* **The build\_tree function creates a binary tree using a priority queue (min-heap) implemented with the heapq library.**
* **Each character and its frequency are first converted into a Node object and added to the heap.**
* **The two nodes with the smallest frequencies are repeatedly merged until a single root node remains, representing the Huffman tree.**

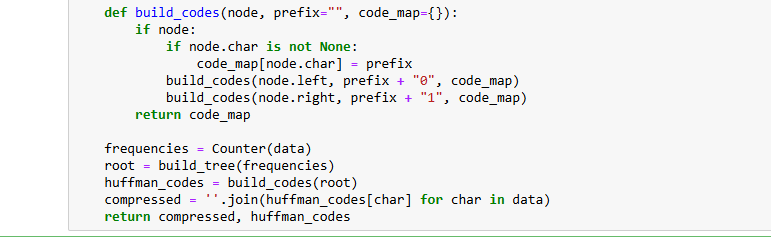


**Generating Huffman Codes:**

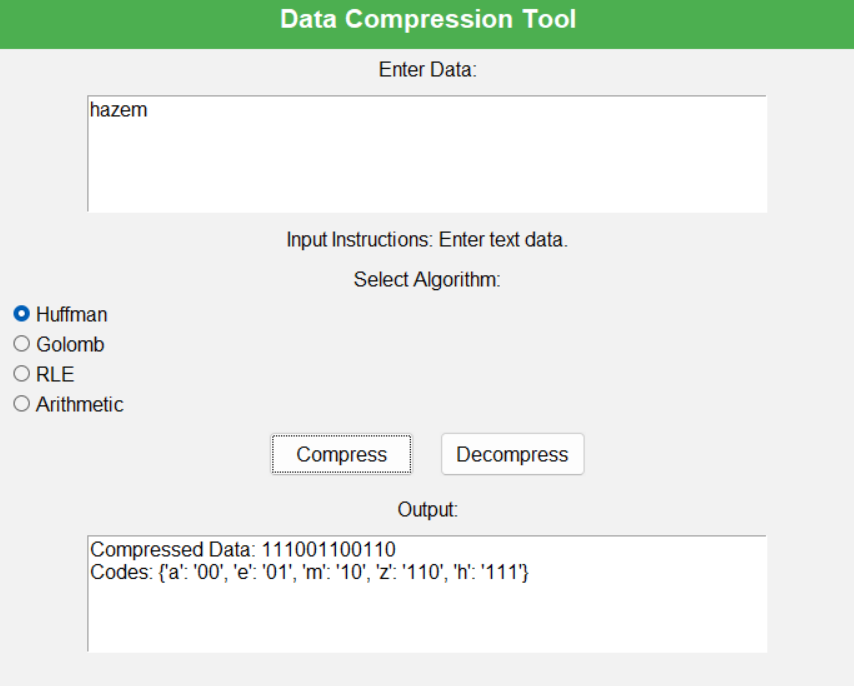
* **The build\_codes function traverses the Huffman tree recursively to assign binary codes to each character.**
* **A prefix ("0" or "1") is appended as the traversal progresses, building unique codes for each character.**
* **The resulting mapping of characters to their binary codes is stored in code\_map.**

**Compression Process:**

* **The frequencies of characters in the input data are calculated using collections.Counter.**
* **The Huffman tree is constructed, and the codes are generated.**
* **The input data is then encoded into a compressed binary string (compressed) by replacing each character with its corresponding Huffman code.**

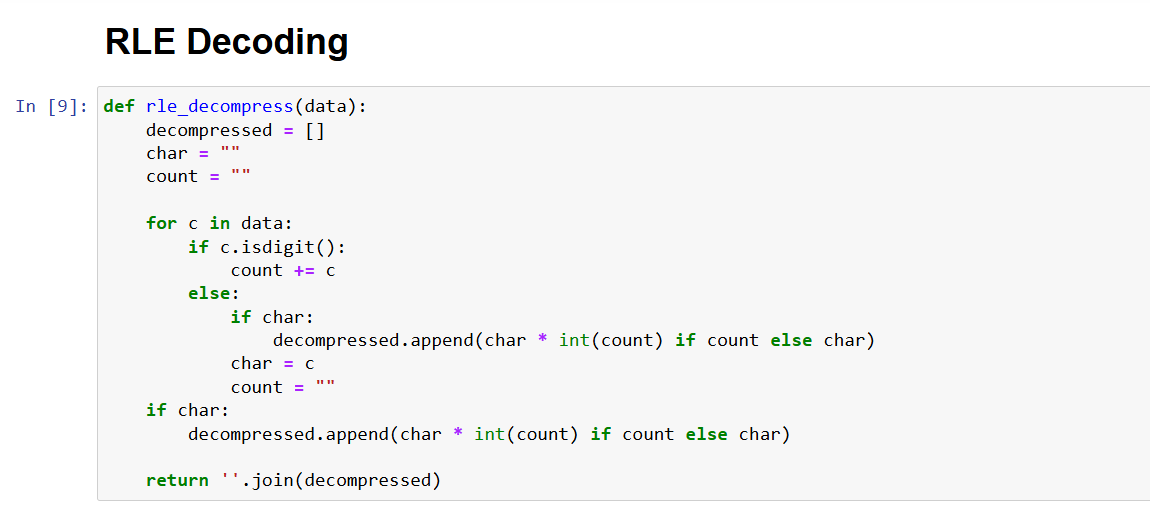


**Huffman compress output example :**

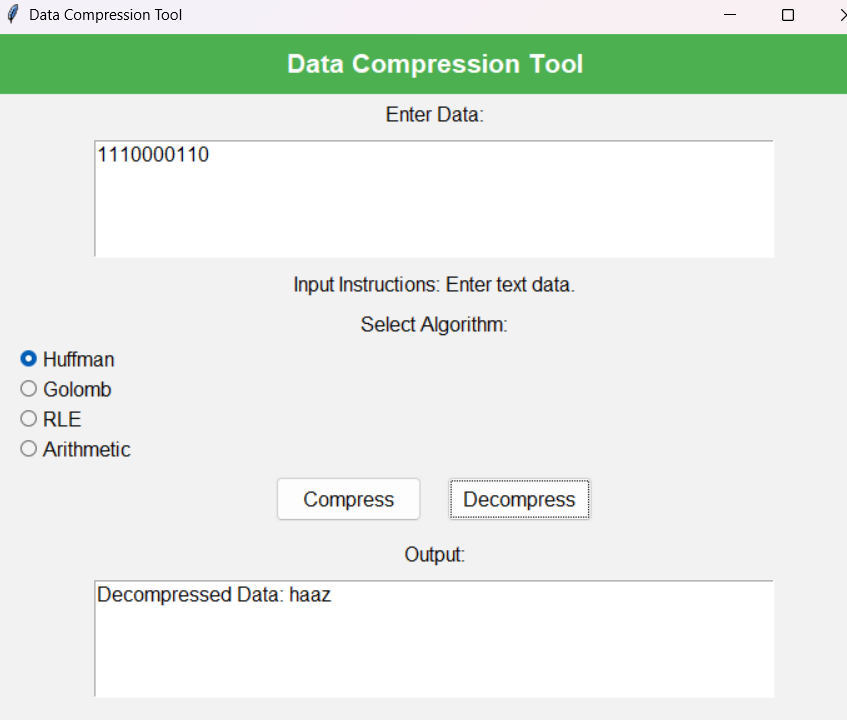
****

**2.2 Huffman Decompression:**

* **Reverse Code Mapping:**
  + **A reverse dictionary is created to map binary codes back to their respective characters, simplifying the decoding process.**
* **Decompression Process:**
  + **Iterates through the compressed binary data, building binary strings one bit at a time.**
  + **Once a complete code matches an entry in the reverse dictionary, the corresponding character is appended to the decompressed output.**
  + **This process continues until all bits are decoded.**
* **Output:**
  + **Returns the fully decompressed data, identical to the original input.**

****

**Huffman compress output example :**



**2.3 RLE Compression:**

Logic of Compression:

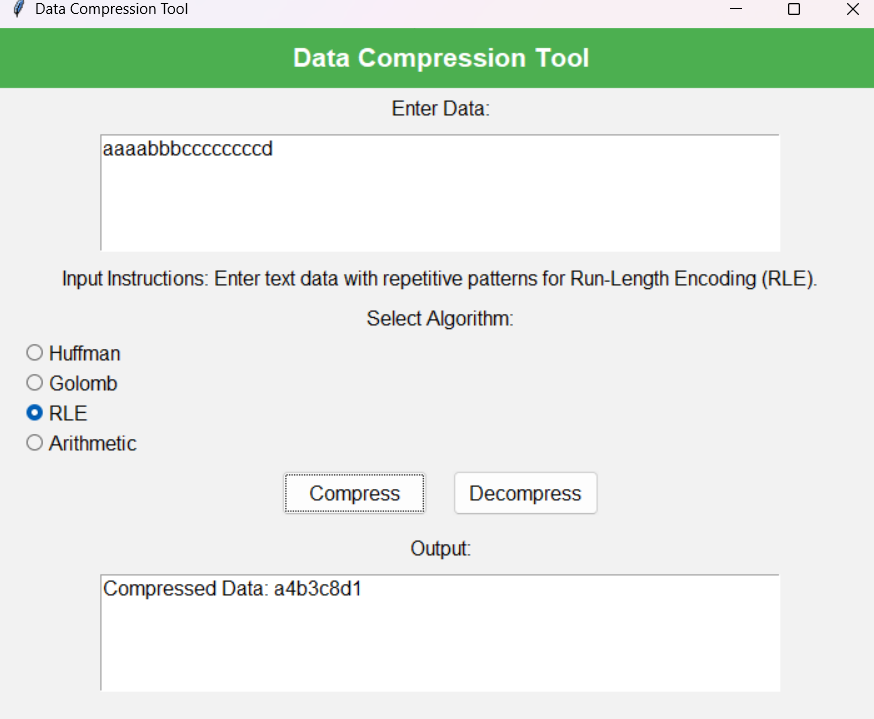
* The function iterates through the input data to identify sequences of consecutive, identical characters (runs).
* For each run, the character and its frequency are combined into a single string (e.g., "A5" for 5 consecutive 'A's).
* Runs are stored in a list (compressed), which is later concatenated into a single compressed string.

Implementation Details:

* Initialization: A list compressed is created to store the encoded runs, and a counter (count) is initialized to track the length of each run.
* Iteration: The loop starts from the second character and compares each character with the previous one:
  + If they match, the counter is incremented.
  + If they differ, the run (character + count) is added to the compressed list, and the counter resets to 1 .
* Finalization: After the loop, the last run is appended to the list, ensuring no data is left unprocessed.

****

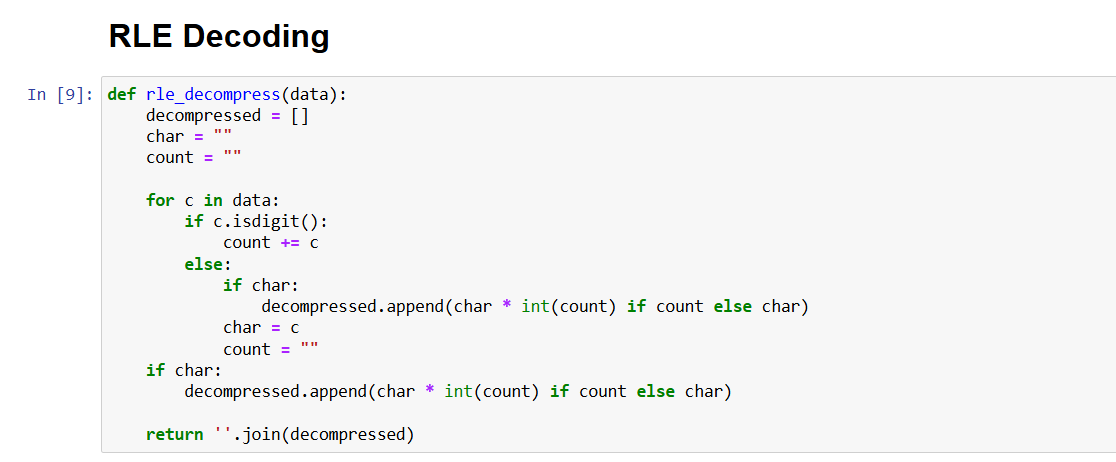
**RLE compress output example :**



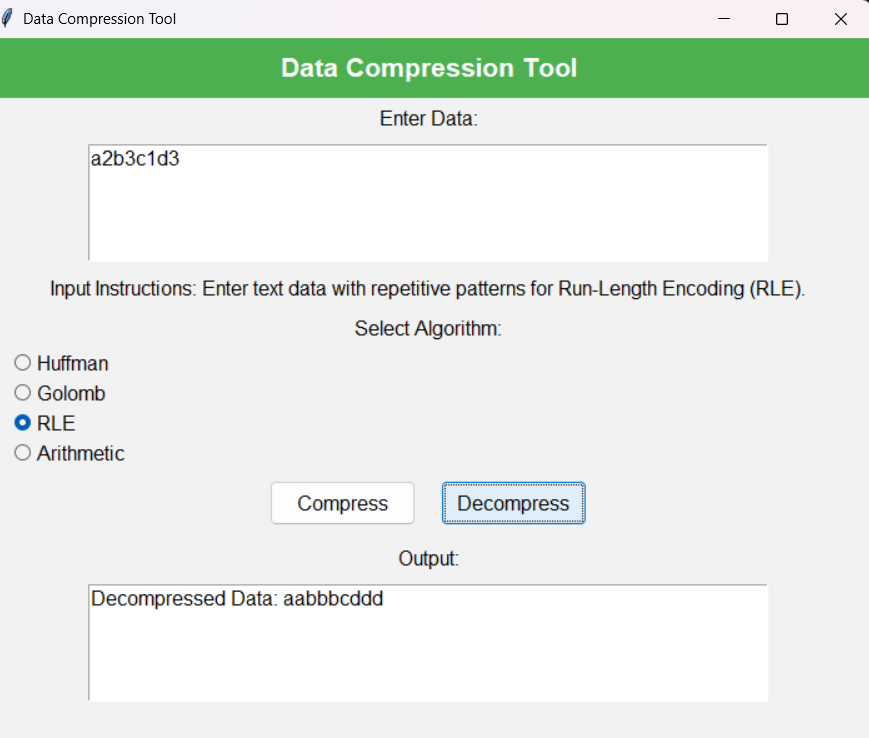
**2.4 RLE Decompression:**

**Implementation Details**:

* **Initialization**:
  + decompressed: A list to store the reconstructed sequence.
  + char: Tracks the current character being processed.
  + count: Tracks the numerical value (count) of repetitions for the current character.
* **Iteration**:
  + For each character in the input string:
    - If the character is a digit, it is added to the count.
    - If it is not a digit, it indicates the start of a new character. At this point:
      * The previous character (char) and its count (count) are processed to expand the sequence using repetition (char \* int(count)).
      * The expanded sequence is appended to the decompressed list.
      * The variables char and count are reset for the next character.
* **Finalization**:
  + After the loop, the last character and its count are processed to ensure no data is left unexpanded.



**RLE decompress output example :**

****

**2.5 Golomb Compression:**

**Function Definition**: The function golomb\_compress(data, m) takes two inputs:

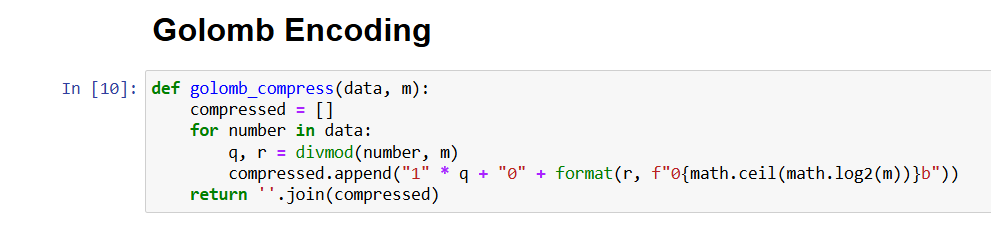
* data: A list of integers that needs to be compressed.
* m: The Golomb parameter, which determines how the numbers will be split into quotient and remainder for compression.

**Compressed Data Storage**: The function initializes an empty list compressed to store the binary representations of the compressed numbers.

**Processing Each Number**: The code then loops through each number in the data list. For every number:

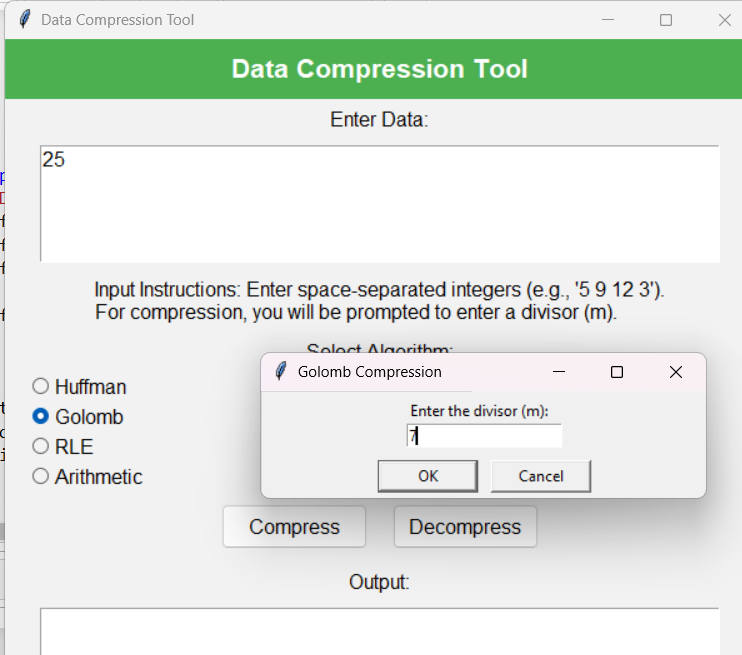
* It divides the number by m to get two values: the quotient (q) and the remainder (r). The quotient represents how many times m fits into the number, and the remainder is what is left over.
* The quotient is encoded using a unary code, which is a series of q ones followed by a zero. This is done by repeating the digit 1 for q times and appending a single 0 after that.
* The remainder is encoded as a binary number. The number of bits used to represent the remainder is determined by the size of m. Specifically, it is calculated by taking the logarithm of m, rounding it up to the nearest integer, and then formatting the remainder to fit that many bits.

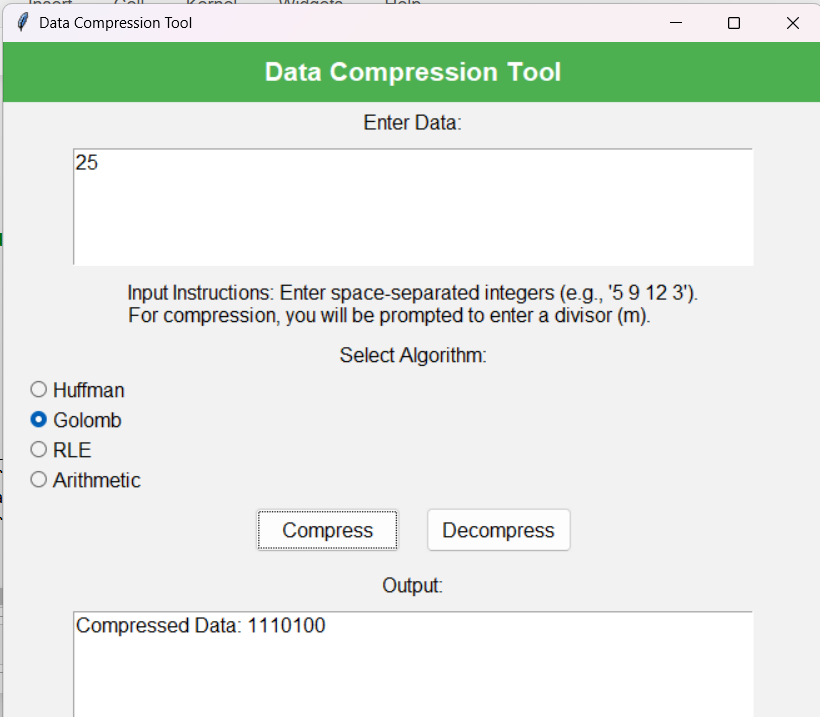
**Combining the Encodings**: For each number, the unary-encoded quotient and the binary-encoded remainder are combined into a single binary string. This string is added to the compressed list.



**Golomb compress output example :**

**User should give M (Divisor) as input first**

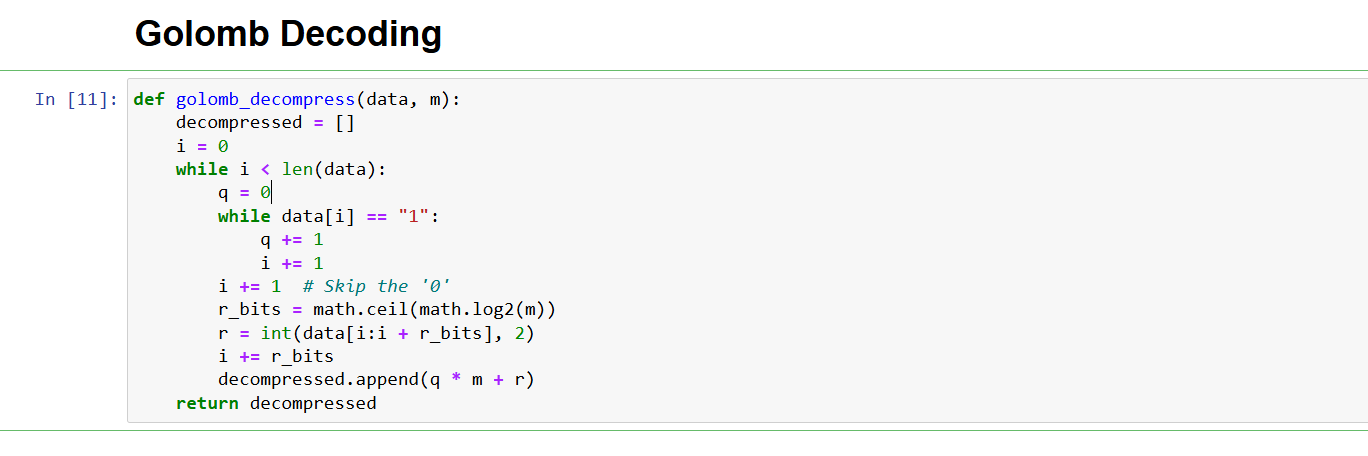
****



**2.6 Golomb Decompression:**

The Golomb decompress function reverses the Golomb compression process. It takes a binary string data and a parameter m as inputs. The function reconstructs the original integers by decoding the binary data in the following way:

1. Quotient Extraction: The unary-encoded quotient is extracted by counting consecutive 1s in the binary string. This quotient represents how many times m fits into the original number.
2. Remainder Extraction: After skipping a separator 0, the binary-encoded remainder is extracted, and its value is determined by converting the binary slice into an integer. The number of bits used for the remainder is dynamically determined based on m.
3. Reconstruction: The original integer is reconstructed using the formula q \* m + r, where q is the quotient and r is the remainder.
4. Output: The reconstructed integers are stored in a list, which is returned as the decompressed data.



**Golomb decompress output example :**

**User should give M (Divisor) as input first**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**2.7 Arithmetic Compression:**

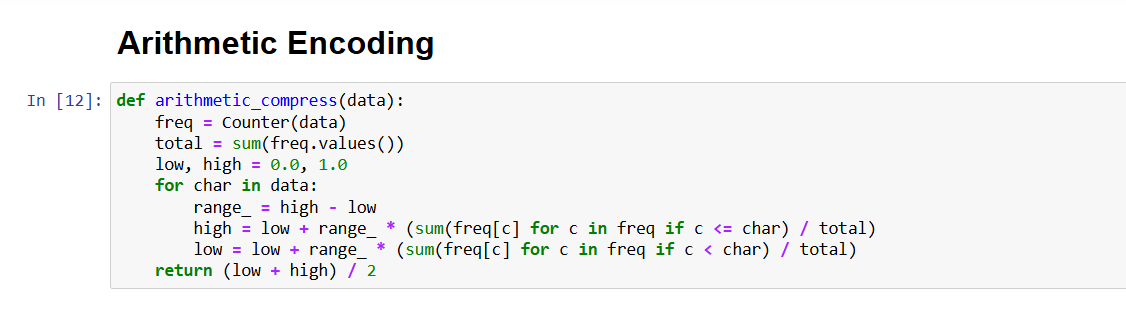
Frequency Calculation:  
The function first calculates the frequency of each character in the input data using Counter(data). It then calculates the total number of characters in the data by summing the frequencies.

Initialize Low and High:  
The range for the compression is initialized with low set to 0.0 and high set to 1.0. This range will represent all possible values for the compressed output.

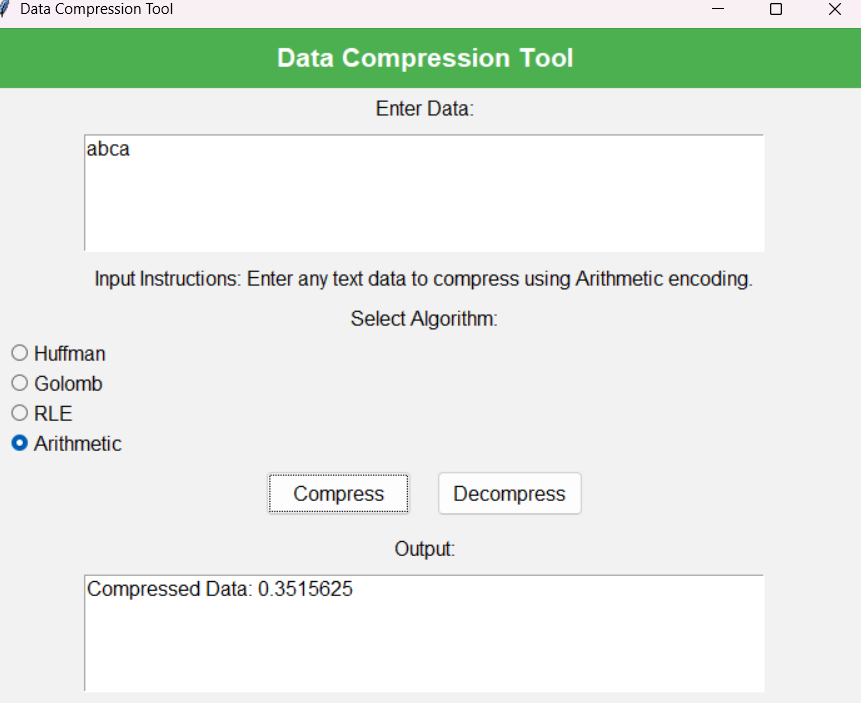
Iterate Through Data:  
The function processes each character in the input data one at a time. For each character:

* The range between low and high is adjusted based on the cumulative probability of the characters in the data up to and including the current character.
* The cumulative probability of characters less than the current character updates the low value.
* The cumulative probability of characters less than or equal to the current character updates the high value.

Final Output:  
After processing all characters in the data, the function calculates the midpoint of the final range ((low + high) / 2). This midpoint represents the compressed value, which uniquely identifies the entire sequence of characters in the input.



**Arithmetic compress output example :**



**2.8 Arithmetic Decompression:**

**Initialization**:  
The function starts by calculating the total frequency of all characters. It sets the initial range for decoding between 0.0 and 1.0 (low and high), and initializes an empty string result to store the decompressed data.

**Iterate for Each Character**:  
A loop runs for the number of characters in the original data (data\_length). For each character:

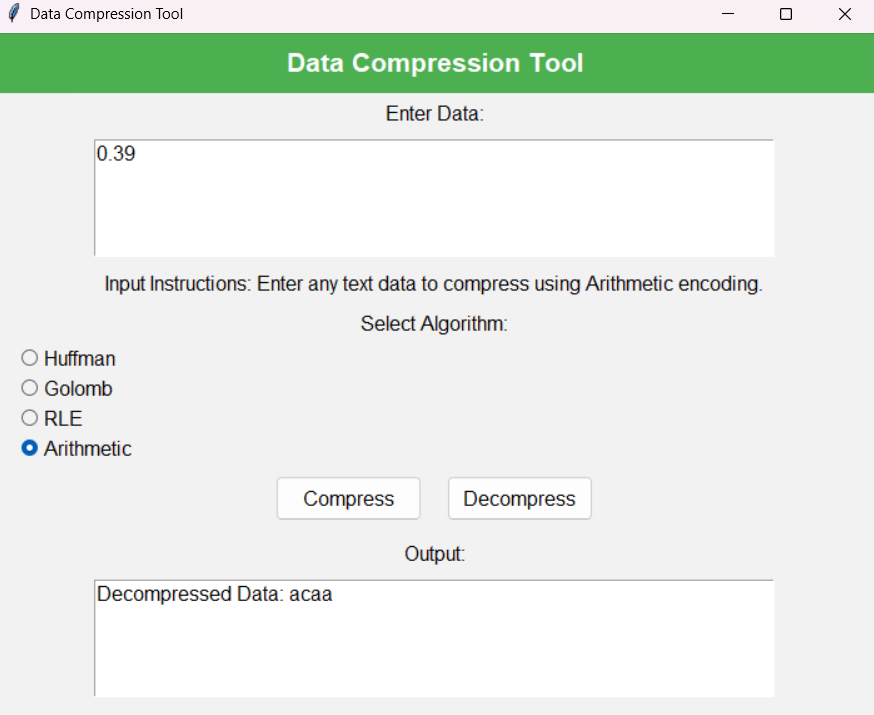
* The width of the current range (range\_) is calculated.
* The normalized position of the code within the range (value) is determined.

**Identify the Character**:  
The function checks which character corresponds to the current range by comparing value with the cumulative probabilities of characters. It iterates through the sorted frequency list of characters, updating the cumulative probability as it goes. Once it finds the character whose cumulative probability encompasses value, it adds that character to the result string.

**Update the Range**:  
After identifying the character, the function updates the low and high values to narrow the range to the one corresponding to that character. This adjustment reflects the next segment of the compressed number that will represent the next character.

**Return the Decompressed Data**:  
After processing all characters, the function returns the fully decompressed string.

**Arithmetic decompress output example :**

****

**Step 3:Graphical user interface (GUI)**

**1..Class Initialization and GUI Setup**

The CompressionApp class is the main application that handles the GUI and compression/decompression logic. Upon initialization, it sets up the window, styles, input fields, and various widgets.

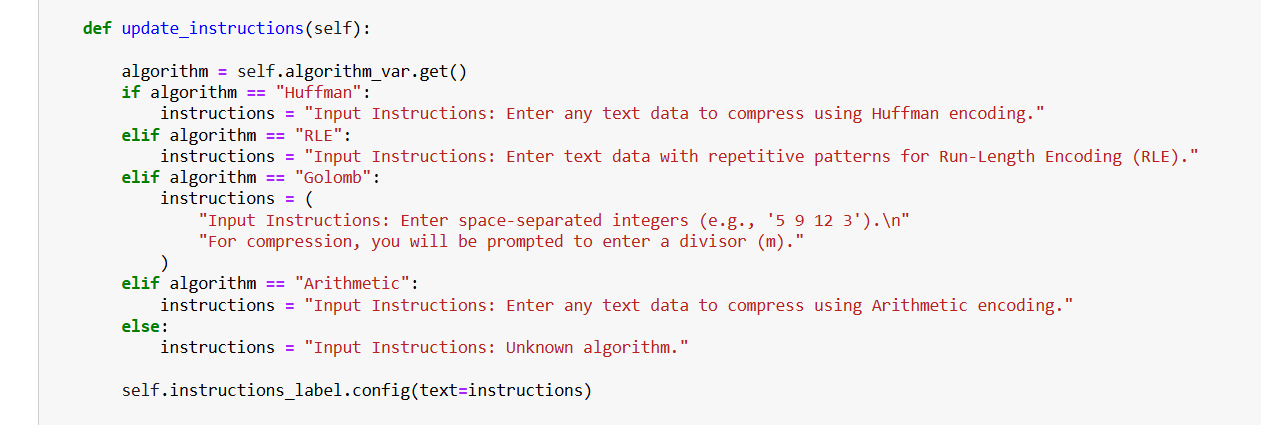
* **Window Setup**: The root window (root) is created, with a title "Data Compression Tool" and a size of 600x500 pixels. The background color is set to a light gray (#f2f2f2).
* **Algorithm Selection**: A StringVar is used to store the selected compression algorithm (Huffman by default).
* **Styles**: The ttk.Style is configured to set the font and background colors for the labels, buttons, and radio buttons.
* **Header**: A header label is created at the top with the text "Data Compression Tool" styled in bold, white text, and a green background.
* **Input Section**:
  + A Text widget is used to allow the user to input data (text or integers).
  + An instruction label guides the user on how to input data based on the selected algorithm.
* **Algorithm Selection**: Four algorithms (Huffman, Golomb, RLE, Arithmetic) are provided as radio buttons for the user to select from. The instructions change based on the selected algorithm, updating dynamically through the update\_instructions method.
* **Buttons**: Two buttons—"Compress" and "Decompress"—are provided to trigger compression or decompression. They call the respective methods when clicked.
* **Output Section**: A disabled Text widget is used to display the output (compressed or decompressed data) so that the user can view the results.



**2.. Updating Instructions**

This method updates the instructions based on the selected compression algorithm:

* **Huffman**: Informs the user to input any text data.
* **RLE**: Encourages the user to input repetitive text data.
* **Golomb**: Instructs the user to input space-separated integers and prompts for a divisor (m) during compression.
* **Arithmetic**: Informs the user to input text data for Arithmetic encoding.

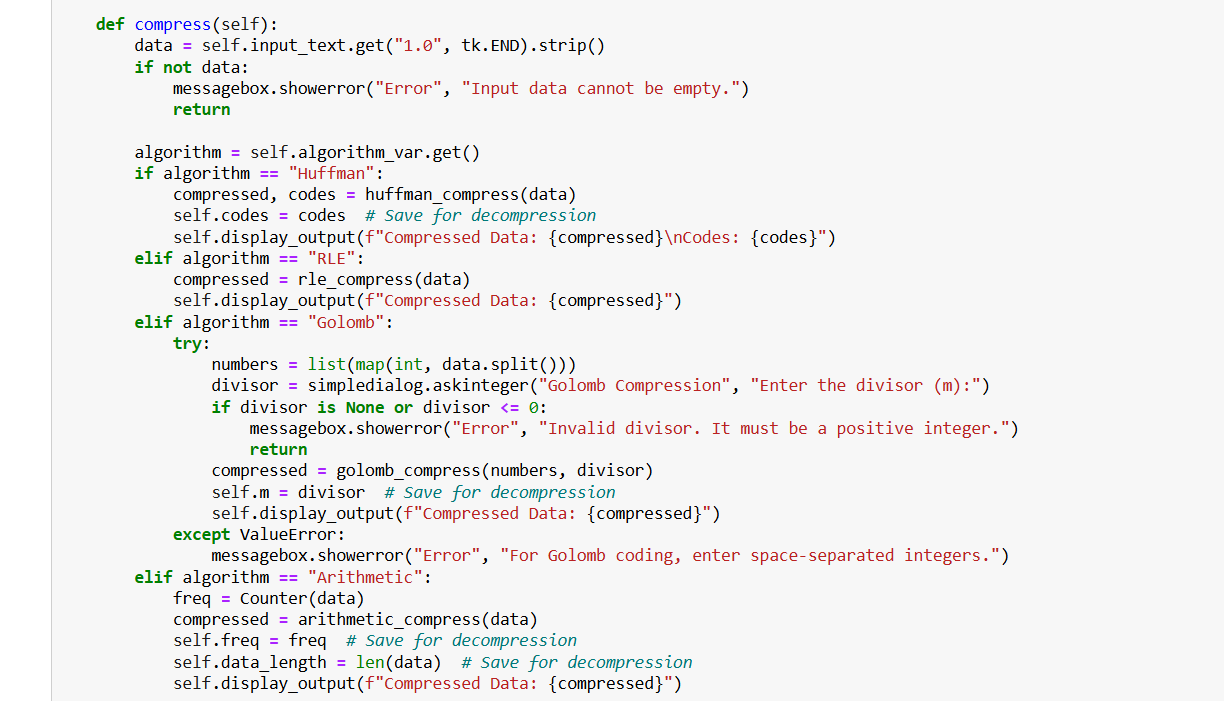


**3. Compression Method**

The compress method is called when the user clicks the "Compress" button. It performs the following steps:

* Retrieves the input data from the Text widget.
* Verifies that the input data is not empty. If it is, an error message is displayed.
* Based on the selected algorithm, the appropriate compression function is called:
  + **Huffman**: Calls huffman\_compress(data), which returns the compressed data and encoding codes.
  + **RLE**: Calls rle\_compress(data) for Run-Length Encoding compression.
  + **Golomb**: Prompts the user for a divisor (m) and calls golomb\_compress(numbers, m).
  + **Arithmetic**: Calls arithmetic\_compress(data) and uses the frequency distribution of the input data for compression.

After compression, the results (compressed data and, in some cases, the encoding codes or frequency table) are displayed in the output section.



1. **Decompression Method (decompress method)**

The decompress method is called when the user clicks the "Decompress" button. It performs the following steps:

* Retrieves the compressed data from the Text widget.
* Verifies that the input data is not empty. If it is, an error message is displayed.
* Based on the selected algorithm, the appropriate decompression function is called:
  + **Huffman**: Calls huffman\_decompress(compressed\_data, self.codes) (using previously stored codes).
  + **RLE**: Calls rle\_decompress(compressed\_data) for decompression.
  + **Golomb**: Prompts the user for the divisor (m) and calls golomb\_decompress(compressed\_data, m).
  + **Arithmetic**: Uses previously stored frequency and data length to call arithmetic\_decompress(code, data\_length, freq) for decompression.

Once decompressed, the resulting data is displayed in the output section.



1. **Displaying Output (display\_output method)**

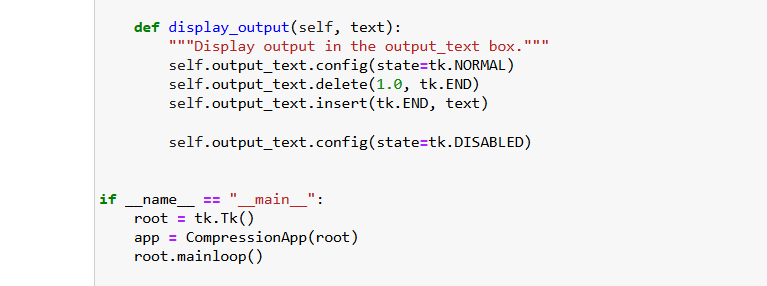
This method is used to display either the compressed or decompressed data in the output section:

* The Text widget is first set to be editable (state=tk.NORMAL), allowing text to be inserted.
* The existing text is cleared using delete(1.0, tk.END).
* The result text is inserted into the Text widget using insert(tk.END, text).
* Finally, the Text widget is set back to read-only mode (state=tk.DISABLED).

1. **Main Program Execution**

In the if \_\_name\_\_ == "\_\_main\_\_": block:

* The root window (root) is created.
* The CompressionApp class is instantiated with the root window as an argument.
* The mainloop() method is called to start the Tkinter event loop, allowing the app to run and interact with the user.



And this is the final GUI output

