AIM:

To study and analyse different multimedia file formats (text, video, image, and audio), covering their structure, header information, and differences between raw and compressed formats.

FILE FORMAT:

A file format is a standard way that information is encoded for storage in a computer file. It specifies how bits are used to encode information in digital storage medium.

Example: Image, Video, Audio, Text file formats.

Text file formats:

Raw:

- **TXT (Text)**:
  Structure: Plain text file without any metadata.
  Header: No specific header; it is a simple sequence of characters.

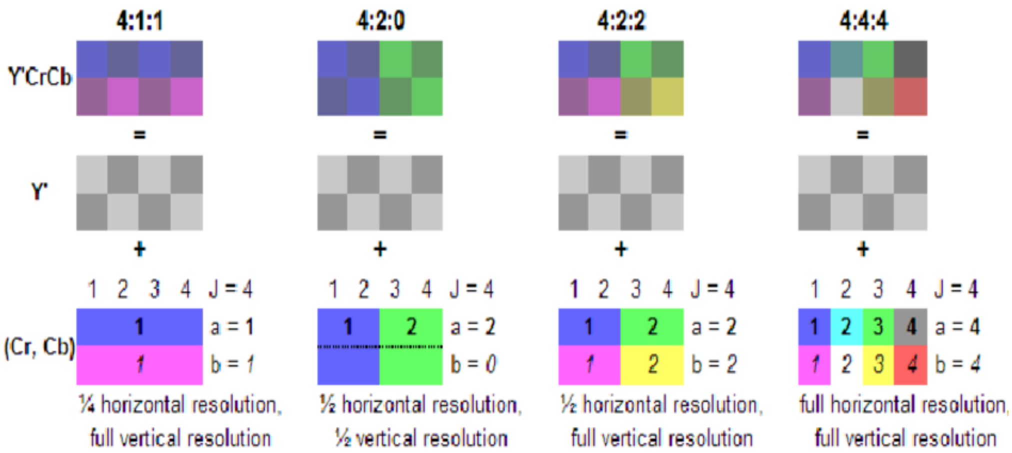
Compressed:

- **PDF (Portable Document Format)**:
  Structure: Compressed text and images.
  Header: Starts with %PDF-1.x, followed by metadata and content structure.
- **DOCX (Microsoft Word Open XML)**:
  Structure: ZIP-compressed XML files and resources.
  Header: Contains the content in XML and various resources in a zipped folder.
- **RTF (Rich Text Format)**:
  Structure: Includes both text and formatting.
  Header: Starts with {\rtf1, followed by information on fonts, formatting, etc.

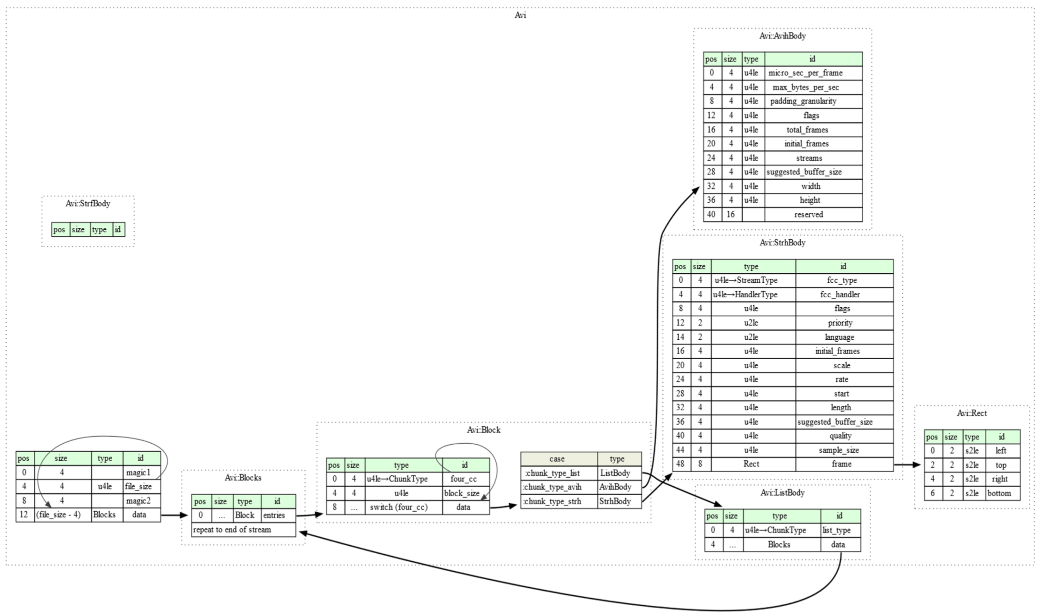
Video file formats:

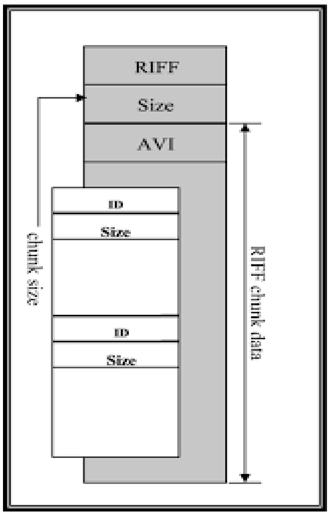
Raw:

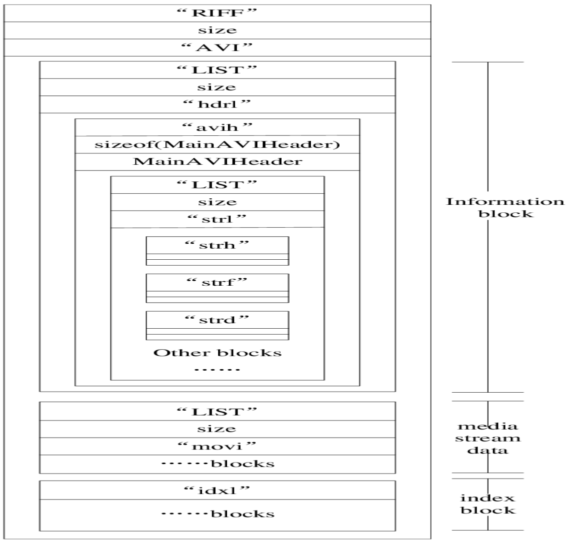
- **YUV (Raw Video Format)**:
  Structure: Stores raw video data, uncompressed.
  Header: No specific header in raw YUV format, just raw pixel values.



- **AVI (Audio Video Interleave)**:
  Structure: Contains audio and video in interleaved format.
  Header: Contains RIFF header followed by format information.







Compressed:

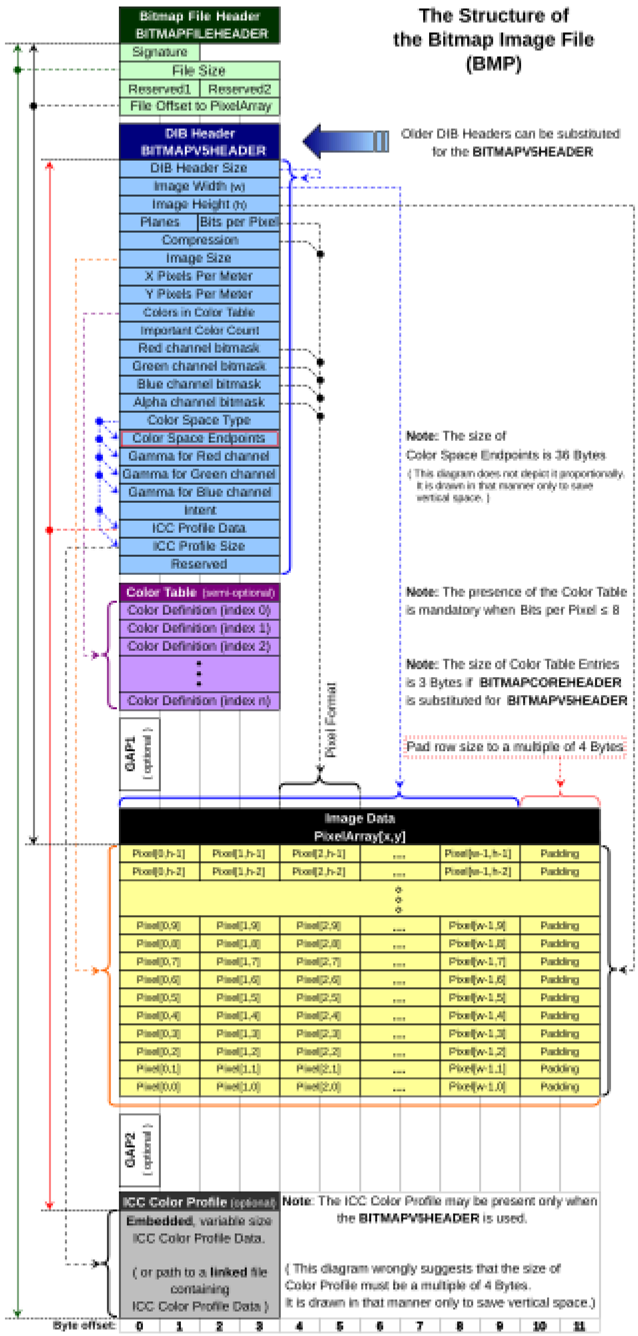
- **MP4 (MPEG-4 Part 14)**:
  Structure: Compressed video and audio streams.
  Header: Begins with ftyp, specifying the type of media and version.
- **MKV (Matroska Video)**:
  Structure: Can contain various video, audio, subtitle tracks.
  Header: EBML-based format starts with 1A45DFA3.

Image file formats:

Raw:

- **BMP (Bitmap Image File)**:
  Structure: Uncompressed raster graphics.
  Header: Begins with a 14-byte file header followed by a 40-byte DIB header.

BMP STRUCTURE:



Compressed:

- **JPEG (Joint Photographic Experts Group)**:
  Structure: Compressed raster image using lossy compression.
  Header: Starts with FFD8, followed by metadata and the image data.

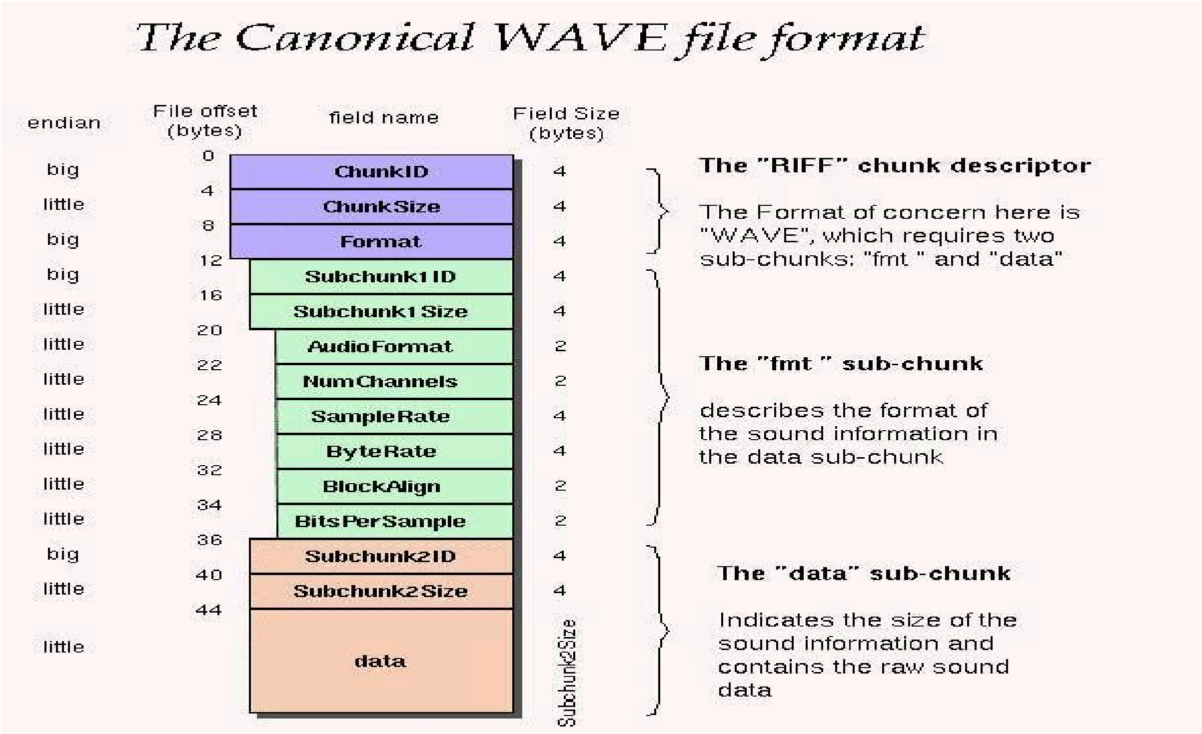
- **PNG (Portable Network Graphics)**:
  Structure: Lossless compressed image format.
  Header: Begins with an 8-byte signature 89 50 4E 47 0D 0A 1A 0A.
- **GIF (Graphics Interchange Format)**:
  Structure: Compressed image file supporting animations.
  Header: Starts with GIF87a or GIF89a, indicating the version.

Audio file formats:

Raw:

- **WAV (Waveform Audio File Format)**:
  Structure: Raw, uncompressed audio data.
  Header: RIFF header begins with 52494646, followed by format and chunk size.

RAW STRUCTURE:



Compressed:

- **MP3 (MPEG Layer 3 Audio)**:
  Structure: Lossy compressed audio.
  Header: Begins with an MP3 frame header, sync word FFF.

- **AAC (Advanced Audio Coding)**:
  Structure: Lossy compressed audio format.
  Header: Begins with 1111 1010 0111 (12-bit sync word), followed by format details.
- **OGG (Ogg Vorbis)**:
  Structure: Lossy compressed container format.
  Header: Begins with OggS, followed by details like version, granule position, etc.

RESULT:

The study provided insights into 4 formats for each media type, focusing on file structure and headers. This understanding aids in identifying and processing multimedia data effectively across various applications.

AIM:

To study the basic programming concepts in Python language like flow content , functions , data types , file handling , classes and objects.

INTRODUCTION:

Python Keywords

Keywords are the reserved words in Python that have predefined meanings and cannot be used as identifiers (like variable or function names). They form the syntax rules for the language.

List of Keywords

False, None, True, and, as, assert, async, await, break, class, continue, def, del, elif, else, except, false, finally, for, from, global, if, import, in, is, lambda, nonlocal, not, or, pass, raise, return, try, while, with, yield

Python Identifiers

Identifiers are names used to identify variables, functions, classes, modules, and other objects. They must start with a letter (A-Z, a-z) or an underscore (_) and can be followed by letters, digits (0-9), or underscores.

Example:

```
my_variable = 10
MyClass = "Example"
```

Variables

Variables are symbolic names that are used to store data. A variable's value can be changed throughout the program. Python is dynamically typed, so you do not need to declare the variable type explicitly.

Example:

```
x = 5
name = "Alice"
```

Constants

Constants are variables that are meant to stay unchanged throughout the program. While Python does not have built-in constant types, it is a convention to use uppercase letters to indicate that a variable should not be modified.

Example:

```
PI = 3.14
```

```
MAX_LIMIT = 100
```

Data Types

Python supports various built-in data types, allowing for the storage of different kinds of data. These include numeric types, sequences, and mappings. Common data types are integers, floats, strings, lists, tuples, and dictionaries.

Example:

```
integer_var = 10        # Integer
float_var = 10.5        # Float
string_var = "Hello"    # String
list_var = [1, 2, 3]    # List
```

Strings

Strings are sequences of characters enclosed in single quotes (`'`) or double quotes (`"`). Strings can be manipulated with various built-in methods, allowing for concatenation, slicing, and formatting.

Example:

```
greeting = "Hello, World!"
substring = greeting[0:5]  # "Hello"
```

Import

The `import` statement is used to include modules (external Python files) into your script. This allows you to utilize functions and classes defined in those modules, promoting code reusability.

Example:

```
import math
result = math.sqrt(16)  # result = 4.0
```

Flow Control

Flow control statements manage the execution of code based on certain conditions. Common flow control statements include `if`, `elif`, `else`, `for`, and `while`. They enable branching and looping in your code.

Example:

```
if x > 10:
    print("x is greater than 10")
elif x == 10:
```

```
    print("x is 10")
else:
    print("x is less than 10")
```

Functions

Functions are blocks of code designed to perform a specific task. They can take parameters and return values, promoting modularity and reusability in your code. Functions are defined using the `def` keyword.

Example:

```
def greet(name):
    return f"Hello, {name}!"
message = greet("Alice")  # message = "Hello, Alice!"
```

File I/O

File input/output (I/O) operations allow you to read from and write to files. This is essential for data persistence. The `open()` function is used to access files, and you can specify the mode (e.g., read, write).

Example:

```
with open("example.txt", "w") as file:
    file.write("Hello, World!")
with open("example.txt", "r") as file:
    content = file.read()  # Read the content of the file
```

Directory

A directory is a file system structure that contains files and other directories. You can manipulate directories (like creating, deleting, or navigating) using the `os` module in Python.

Example:

```
import os
current_directory = os.getcwd()  # Get current directory
files = os.listdir(current_directory)  # List all files in the directory
```

Class

A class is a blueprint for creating objects. It defines attributes (data) and methods (functions) that the created objects can use. Classes promote object-oriented programming principles such as encapsulation and inheritance

Example:

```
class Dog:
    def __init__(self, name):
        self.name = name
```

```
    def bark(self):
        return "Woof!"
my_dog = Dog("Rex")
print(my_dog.bark())  # Output: "Woof!"
```

PYTHON LIBRARIES IN ICT:

NumPy

A fundamental library for numerical computing in Python. It provides support for multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

SciPy

SciPy is a free and open-source Python library used for scientific computing and technical computing. SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering.

PyPi

The Python Package Index (PyPI) is a repository of software for the Python programming language.

Pandas

Pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series. It is free software released under the three-clause BSD license.

pycrate

A Python library to ease the development of encoders and decoders for various protocols and file formats.

binascii

The binascii module contains a number of methods to convert between binary and various ASCII-encoded binary representations. Normally, you will not use these functions directly but use wrapper modules like base64 instead. The binascii module contains low-level functions written in C for greater speed that are used by the higher-level modules.

JSON

JSON is a data interchange format that uses human-readable text to store and transmit data. It's based on JavaScript object syntax and is often used in web applications.

MATLAB TOOLS:

MATLAB Coder

MATLAB Coder is a tool that automatically converts MATLAB code into C and C++ It allows users to generate code for real-time applications, embedded systems performance-critical.

MATLAB Toolboxes

Offers a comprehensive set of functions for image processing, analysis, and visualization. It allows users to enhance and manipulate images efficiently.

RESULT:

Thus ,this study provides a foundational understanding of Python programming, covering essential concepts that enable effective coding, code organization, and the use of object-oriented programming principles.

AIM:

To implement Shannon-Fano encoding and decoding to compress and decompress messages by generating unique binary codes for each character based on their frequency, achieving efficient data representation.

ALGORITHM:

Step 1: Read the original message from a file (input_message.txt).

Step 2: Calculate and sort the frequency of each character in descending order.

Step 3: Generate Shannon-Fano codes by recursively dividing the sorted list of charactersinto two parts with nearly equal frequencies, assigning 0 to the left subset and 1 to the right subset until each character has a unique code.

Step 4: Encode the message by replacing each character in the original message with its Shannon-Fano code.

Step 5: Save the encoded binary message to a file (encoded_message.fano).

Step 6: Decode the message by reconstructing the original message from the encoded binary message by matching binary codes to characters.

Step 7: Save the decoded message to a file (decoded_message.txt).

Step 8: Verify that the decoded message matches the original message, confirming Successful encoding and decoding.

SOURCE CODE:

```
import os
def shannon_fano_recursive(symbols_freq, code=''):
if len(symbols_freq) == 1:
    return {symbols_freq[0][0]: code}
    total_freq = sum([freq for symbol, freq in symbols_freq])
    cumulative_freq = 0
    split_point = 0
    for i in range(len(symbols_freq)):
        cumulative_freq += symbols_freq[i][1]
        if cumulative_freq >= total_freq / 2:
            split_point = i + 1
            break
        left_part = shannon_fano_recursive(symbols_freq[:split_point], code + '0')
        right_part = shannon_fano_recursive(symbols_freq[split_point:], code + '1')
        left_part.update(right_part)
        return left_part
def encode_message(message, codes):
    return ''.join(codes[char] for char in message)
def decode_message(encoded_message, codes):
    reverse_codes = {v: k for k, v in codes.items()}
    current_code = ''
    decoded_message = ''
    for bit in encoded_message:
        current_code += bit
    if current_code in reverse_codes:
        decoded_message += reverse_codes[current_code]
        current_code = ''
        return decoded_message
def main():
    input_file = input("Enter the text file name: ")
    with open(input_file, 'r') as file:
    message = file.read().strip()
```

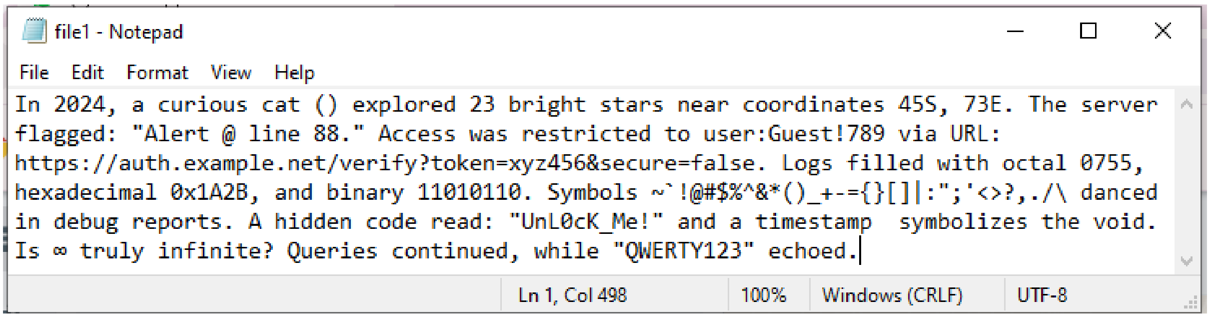
```
        print("Original message successfully loaded.")
        symbols_freq = {char: message.count(char) for char in set(message)}
        symbols_freq = sorted(symbols_freq.items(), key=lambda x: x[1], reverse=True)
        codes = shannon_fano_recursive(symbols_freq)
        folder_name = "output_files"
        if not os.path.exists(folder_name):
                os.makedirs(folder_name)
                encoded_message = encode_message(message, codes)
                encoded_file = os.path.join(folder_name, 'encoded_message.fano')
                with open(encoded_file, 'w') as file:
                file.write(encoded_message)
                print(f"Encoded message saved to {encoded_file}")
                decoded_message = decode_message(encoded_message, codes)
                decoded_file = os.path.join(folder_name, 'decoded_message.txt')
                with open(decoded_file, 'w') as file:
                file.write(decoded_message)
                print(f"Decoded message saved to {decoded_file}")
                input_size = len(message)
                output_size = len(decoded_message)
                if input_size == output_size and message == decoded_message:
                        print(f"Success! The input file size ({input_size} bytes) matches the
                        output file size ({output_size} bytes).")
                        print("SUCCESS!!!THE ORIGINAL MESSAGE AND THE
                        DECODED MESSAGE ARE **EQUAL**.")
        else:
                print(f"Error! The input file size ({input_size} bytes) does not match the
                output file size ({output_size} bytes).")
                print("THE ORIGINAL MESSAGE AND THE DECODED MESSAGE ARE
                NOT **EQUAL**.")
if __name__ == "__main__":
   main()
```

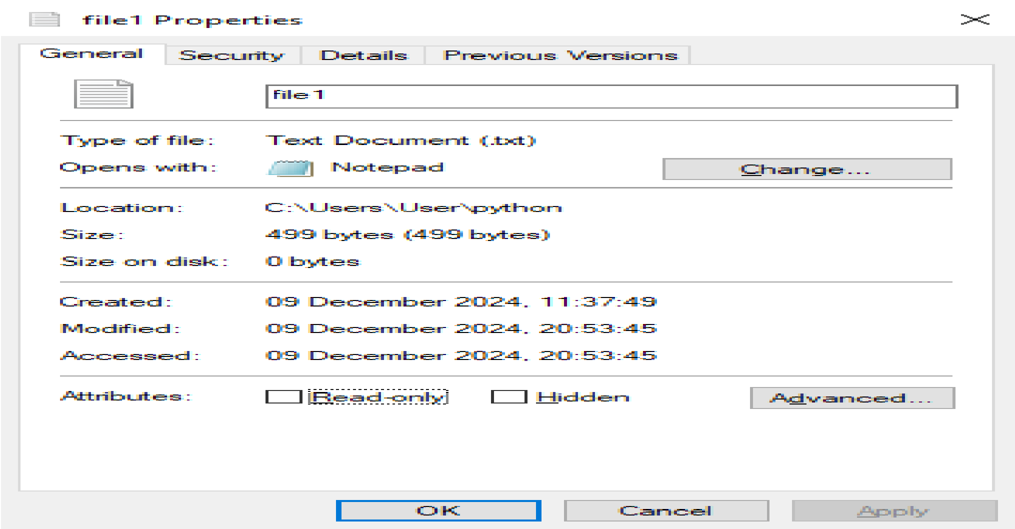
INPUT AND OUTPUT:

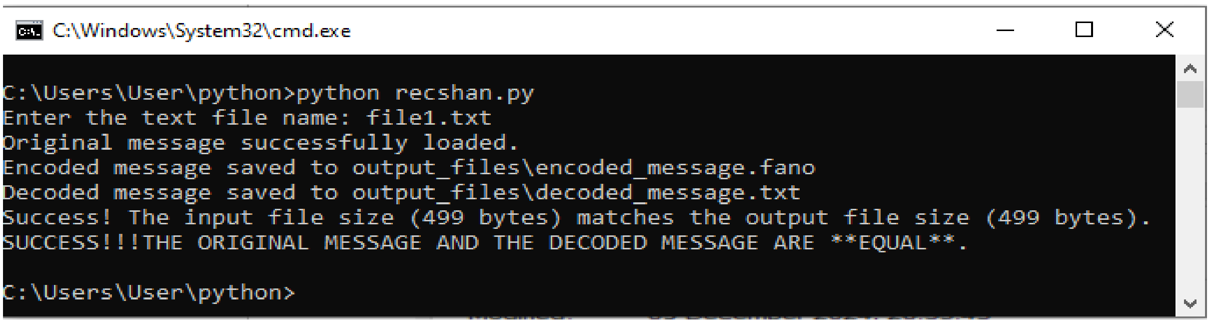
INPUT TEXT FILE CONTENT:

In 2024, a curious cat () explored 23 bright stars near coordinates 45S, 73E. The server flagged: "Alert @ line 88." Access was restricted to user:Guest!789 via URL: https://auth.example.net/verify?token=xyz456&secure=false. Logs filled with octal 0755, hexadecimal 0x1A2B, and binary 11010110. Symbols ~`!@#$%^&*()_+-={}[]|:";'<>?,./\ danced in debug reports. A hidden code read: "UnL0cK_Me!" and a timestamp symbolizes the void. Is ∞ truly infinite? Queries continued, while "QWERTY123" echoed.

INPUT FILE: – file1.txt







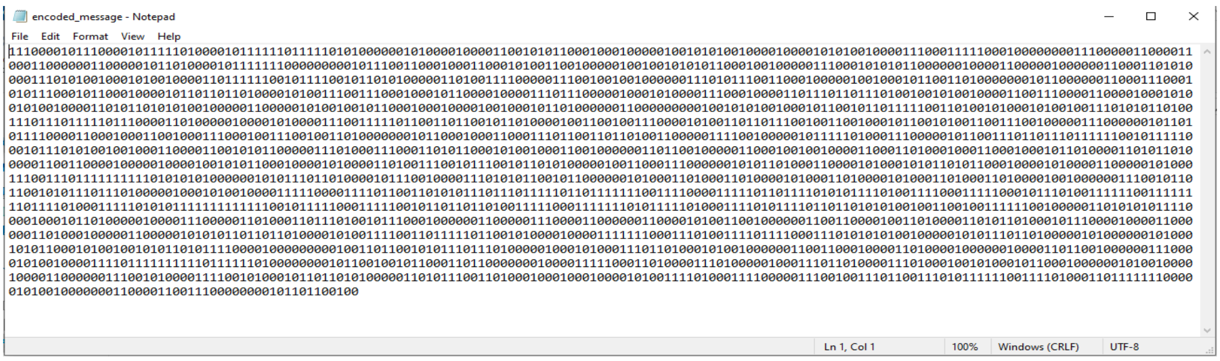
OUTPUT FILES:

Encoded message:

1110000101110000101111101000010111111011111010100000010100001000011001010110
0010001000000100101010010000100001010100100001110001111100010000000011100000
1100001100011000000110000010110100001011111110000000001011100110001000110001
0100110010000001001001010101100010010000011100010101011000000100001100000100
0000110001101010000111010100100010100100001101111110010111100101101010000011
0100111100000111001001001000000111010111001100010000010010001011001101000000
0101100000011000111000101011100010110001000010110110110100001010011100111000
1000101100001000011101110000010001010000111000100001101110110111010010010100
10000110011100001100001000101001010010000110101101010100100000110000010100 1

001011000100010000100100010110100000011000000000100101010010001011001011011
11100110100101000101001001110101011010011101110111110111000011010000001000010
1000011100111110110011011001011010000100110010011100001010011011011100100110
0100010110010100110011100100000111000000101101011110000110001000110010001110
0010011100100110100000001011000100011000111011001101101001100000111100100000
10111111010001110000010110011101101110111111001011111000101110101001001000110
0001100101011000001110100011100011010110001010010001100100000011011001000001
1000100100100001100011010001000110001000101101000011010110100000011001100001
0000010000100101011000100001010000110100111001011100101101010000001001100011
10000000101011010001100001010001010110101100010000101000011000000101000111001
11011111111110101010100000001010111011010000101110010000111010101100101100000
0101000110100011010000101000110100001010001101000110100001001000000011100101
1011001010111011101000001000101001000011111000011110110011010101110111011111
01101111111001111000011111011011110101011110100111100011111000101110100111111
001111111101111010001111101010111111111111100101111100011111001011011011010011
11100011111110101111101000111101011110110110101010010011001001111110010000011
01010101111000010001011010000001000011100000110100011011101001011100010000001
1000001110000110000001100001010011001000000011001100001001101000011010110100
0101110000100001100000011010001000001100000101010110110110100001010011110011
0111110110010100001000011111110001110100111101111000111010101010010000010101
110110100000010100000001010001010110001010010010101101011110000100000000001001
10110010101110111010000010001010001110110100010100100000001100110001000011010
0001000000010000110110010000000111000010100100001111011111111111011111101000000
001011001001011000110110000000010000111110001101000011101000000100011101101000
0111010001001010001011000100000010100100001000011000000011100101000011110010
1000101101101010000011010111001101000100010001000010100111101000111100000111
001001110110011101011111100111101000110111111110000010100100000000110000110011
100000000101101100100

Encoded file:

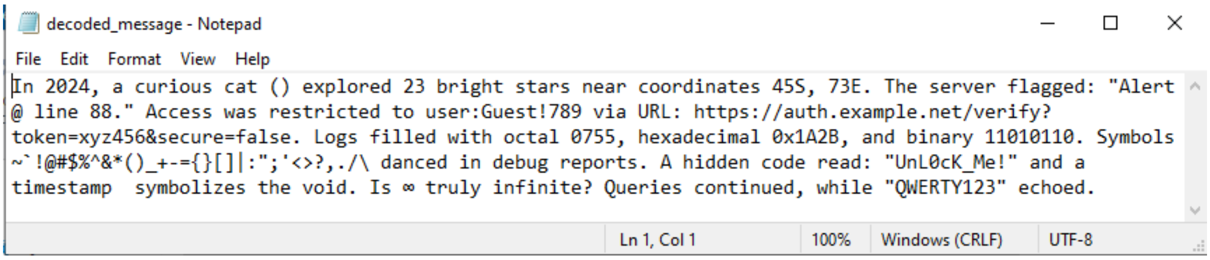


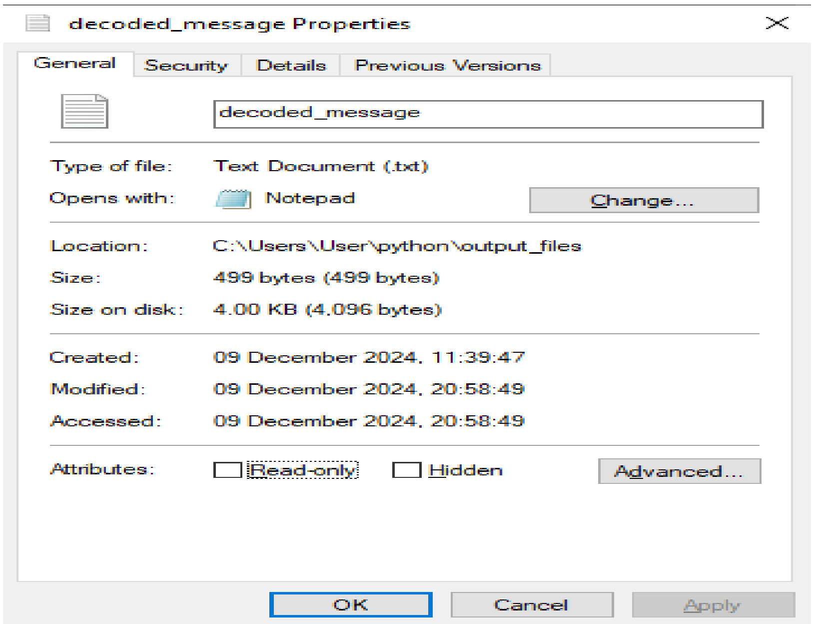
Decoded message:

In 2024, a curious cat () explored 23 bright stars near coordinates 45S, 73E. The server flagged: "Alert @ line 88." Access was restricted to user:Guest!789 via URL: https://auth.example.net/verify?token=xyz456&secure=false. Logs filled with octal 0755, hexadecimal 0x1A2B, and binary 11010110. Symbols ~`!@#$%^&*()_+-={}[]|:";'<>?,./\

danced in debug reports. A hidden code read: "UnL0cK_Me!" and a timestamp symbolizes the void. Is ∞ truly infinite? Queries continued, while "QWERTY123" echoed.

Decoded file:





RESULT:

Thus,The Shannon-Fano encoding and decoding to compress and decompress text messages is implemented and output verified after successful execution.

AIM:

The aim of this code is to implement Huffman encoding and decoding, a lossless data compression algorithm that assigns variable-length codes to input characters based on their frequencies, resulting in efficient data representation.

ALGORITHM:

Step 1: Read the original message from a file (input_message.txt).

Step 2: Calculate and sort the frequency of each character in descending order.

Step 3: Create a priority queue (min-heap) to build the Huffman tree by:

  i. Inserting each character and its frequency as a node.
  ii. Repeatedly extracting the two nodes with the lowest frequency and combining them into a new node until only one node remains (the root of the Huffman tree).

Step 4: Generate Huffman codes by traversing the Huffman tree, assigning 0 for left branches and 1 for right branches.

Step 5: Encode the message by replacing each character in the original message with its Huffman code.

Step 6: Save the encoded binary message to a file (encoded_message.huff).

Step 7: Decode the message by reconstructing the original message from the encoded binary message using the Huffman tree.

Step 8: Save the decoded message to a file (decoded_message.txt).

Step 9: Verify that the decoded message matches the original message, confirming successful encoding and decoding.

SOURCE CODE:

```
import os
import heapq
class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None
 def __lt__(self, other):
        return self.freq < other.freq
def build_huffman_tree(symbols_freq):
    heap = [Node(char, freq) for char, freq in symbols_freq]
    heapq.heapify(heap)
        while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
return heap[0]
def generate_huffman_codes(node, code='', codes=None):
    if codes is None:
        codes = {}
    if node is not None:
        if node.char is not None:
            codes[node.char] = code
        generate_huffman_codes(node.left, code + '0', codes)
        generate_huffman_codes(node.right, code + '1', codes)
    return codes
def encode_message(message, codes):
    return ''.join(codes[char] for char in message)
def decode_message(encoded_message, root):
    decoded_message = ''
    current_node = root
```

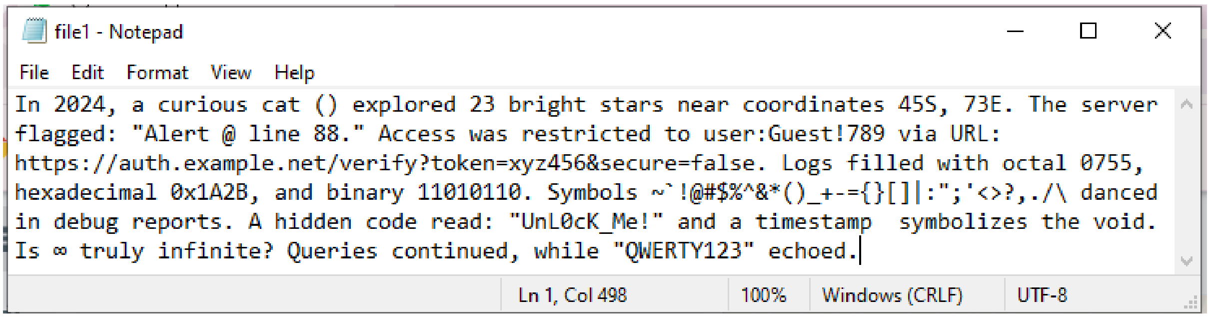
```
    for bit in encoded_message:
        current_node = current_node.left if bit == '0' else current_node.right
        if current_node.char is not None:
            decoded_message += current_node.char
            current_node = root
    return decoded_message
def main():
    input_file = input("Enter the text file name: ")
    with open(input_file, 'r') as file:
        message = file.read().strip()
    print("Original message successfully loaded.")
    symbols_freq = {char: message.count(char) for char in set(message)}
    symbols_freq = sorted(symbols_freq.items(), key=lambda x: x[1], reverse=True)
    huffman_tree_root = build_huffman_tree(symbols_freq)
    codes = generate_huffman_codes(huffman_tree_root)
    folder_name = "output_files"
    if not os.path.exists(folder_name):
        os.makedirs(folder_name)
    encoded_message = encode_message(message, codes)
    encoded_file = os.path.join(folder_name, 'encoded_message.huff')
    with open(encoded_file, 'w') as file:
        file.write(encoded_message)
    print(f"Encoded message saved to {encoded_file}")
    decoded_message = decode_message(encoded_message, huffman_tree_root)
    decoded_file = os.path.join(folder_name, 'decoded_message.txt')
    with open(decoded_file, 'w') as file:
        file.write(decoded_message)
    print(f"Decoded message saved to {decoded_file}")
    input_size = len(message)
    output_size = len(decoded_message)
    if input_size == output_size and message == decoded_message:
        print(f"Success! The input file size ({input_size} bytes) matches the output file size ({output_size} bytes).")
        print("SUCCESS!!! THE ORIGINAL MESSAGE AND THE DECODED MESSAGE ARE **EQUAL**.")
    else:
        print(f"Error! The input file size ({input_size} bytes) does not match the output file size ({output_size} bytes).")
        print("THE ORIGINAL MESSAGE AND THE DECODED MESSAGE ARE NOT **EQUAL**.")
if __name__ == "__main__":
    main()
```

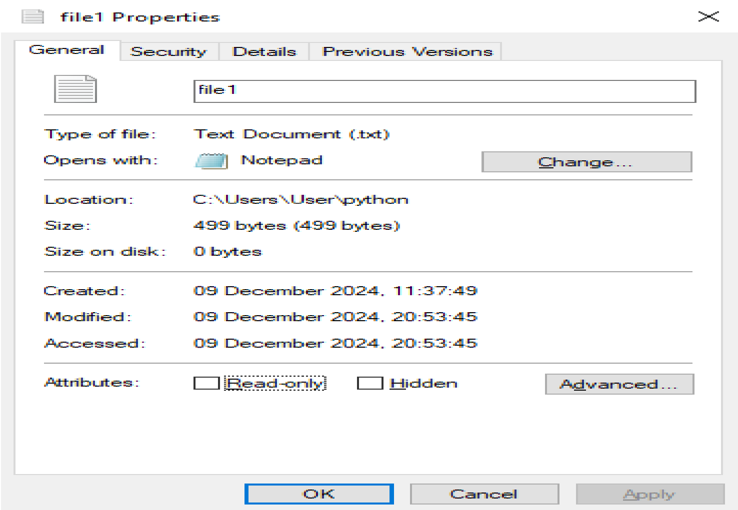
INPUT AND OUTPUT:
INPUT TEXT FILE – file1.txt

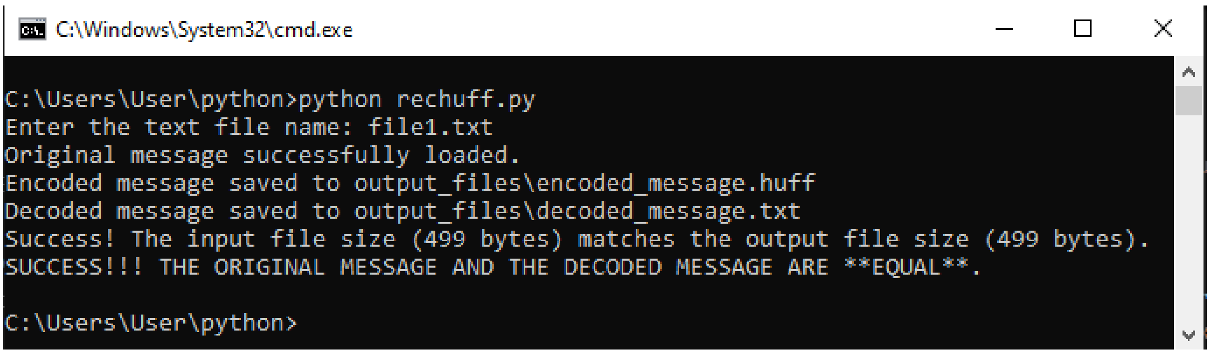
CONTENT:

In 2024, a curious cat () explored 23 bright stars near coordinates 45S, 73E. The server flagged: "Alert @ line 88." Access was restricted to user:Guest!789 via URL: https://auth.example.net/verify?token=xyz456&secure=false. Logs filled with octal 0755, hexadecimal 0x1A2B, and binary 11010110. Symbols ~`!@#$%^&*()_+-={}[]|:";'<>?,./\ danced in debug reports. A hidden code read: "UnL0cK_Me!" and a timestamp symbolizes the void. Is ∞ truly infinite? Queries continued, while "QWERTY123" echoed.

INPUT FILE: – file1.txt





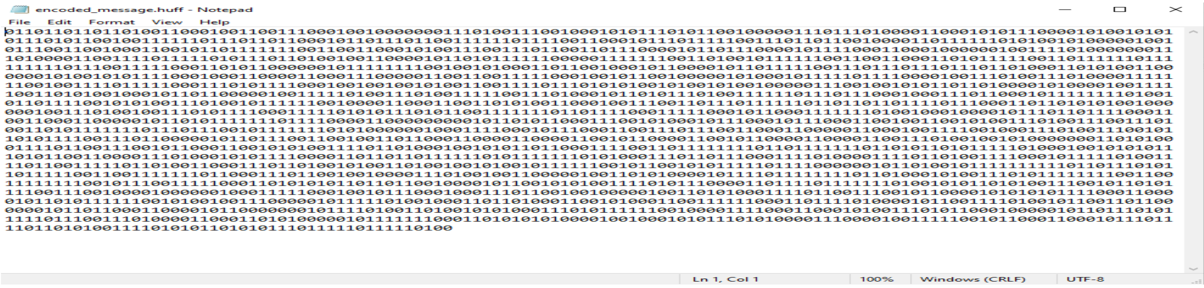


OUTPUT FILES:

Encoded message:

0111110010110111010011111011111010101110010010110011111011111010010111101001011011000110111011101010010111010111000110101111011010111110110110010111100110
0001001000011000111001110101010011110010110110101000010110110110110101111101
1111011001101110000110110001111100000000100101100101111101010101100111011011110100100011110101100011011010110000110111011011110111011011111011010010111100011011000101011000111000101101111101111000000110111110010001101110111010001101100011101010111000011110101101000010110110001110001100001101100111101011111
0001011010111110100000101101101101111100000000011010001100111101111001011010
1010100101100101000110111011011101011011111111000001010101000001101101110001
1000111100110000111001011011101110011000000010010111010001010010110101101100
1011111010001110000101101111100010010000100101100001011011011011111010000010
1010111001111000000101101000000101010001110101000111111000010001010100011110111101010001001100110100110011011010110011010011011101111001010110010011101001100110001010111001011111001100001101001000010111010111001010111111001110111
0000101001000100011010000000010100110001111011011010111001100001111100111101
1101001001000100101001001111111001110010111110101110111010100011011110101100
1111100010010111100110001110111011110101100000101011110100110001100010101001
0110110110111110111110010011011001110111000110001110011101010001011011110101100101010101001101000111101110111111000001110001111100101011100000111110011000110000000110100011111000111011111111011100001111010011010100011010001001101
1000111100001001101001000101110110111100000000100101101100010001111010101101111011011111110111001011100000110100100011110111011011110011011011110101100011
0110001111110000100000110110110010110101111110101010000010111111101100110000
1111100000000100100111000100011101001010000111100000100001111010110111111101010010111101101111111100100100101011000011000111000101101111000101111001101001111010010100001010010001101110111101011000001010111110101100111010001001010
0001111101111100110111000011001010010001000110100000000101001100011110101110
10000101000111010111000011000011110111100111000101111100000111110011101011100
1011001110110100110010000010101111010100000001111000010101001100101111000001011101000000110100001110101111111110101000010010110110010011011000111111001010
1110000011010011111000010001010100111110100001101101011001111110111100010101
1100111110001110000110100110101101110100110110001100010001110001011011101011
0111110000100110010110011111000111110000000111011000010101010010110111010011011000101111

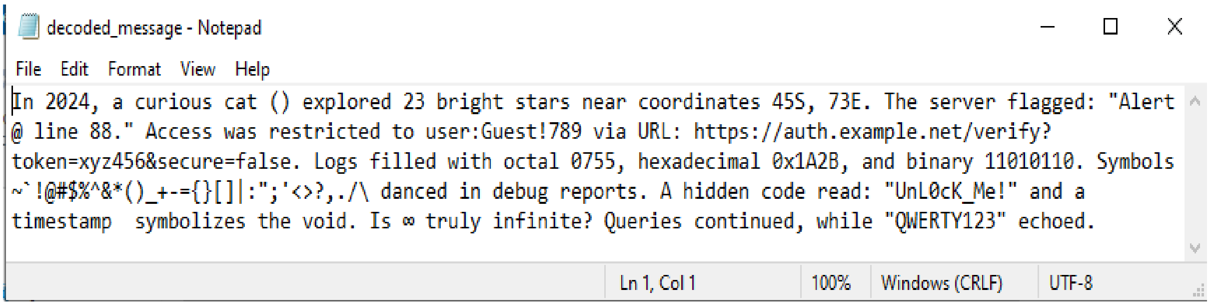
Encoded file:



Decoded message:

In 2024, a curious cat () explored 23 bright stars near coordinates 45S, 73E. The server flagged: "Alert @ line 88." Access was restricted to user:Guest!789 via URL: https://auth.example.net/verify?token=xyz456&secure=false. Logs filled with octal 0755, hexadecimal 0x1A2B, and binary 11010110. Symbols ~`!@#$%^&*()_+-={}[]|:";'<>?,./\ danced in debug reports. A hidden code read: "UnL0cK_Me!" and a timestamp symbolizes the void. Is ∞ truly infinite? Queries continued, while "QWERTY123" echoed.

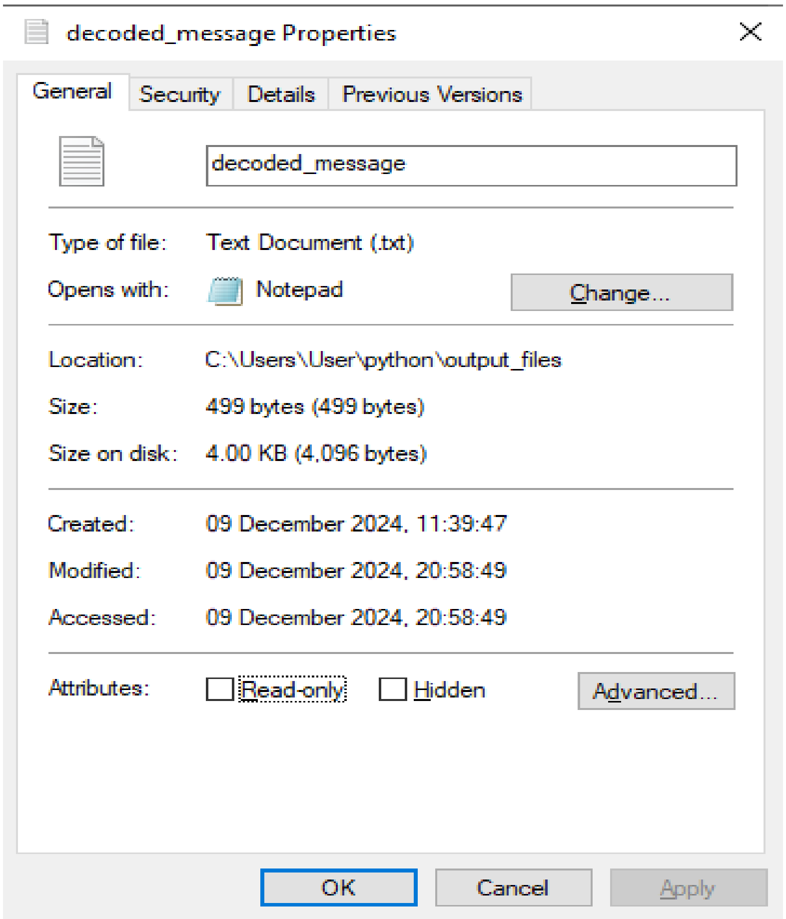
Decoded file:


decoded_message - Notepad

File Edit Format View Help

In 2024, a curious cat () explored 23 bright stars near coordinates 45S, 73E. The server flagged: "Alert @ line 88." Access was restricted to user:Guest!789 via URL: https://auth.example.net/verify?token=xyz456&secure=false. Logs filled with octal 0755, hexadecimal 0x1A2B, and binary 11010110. Symbols ~`!@#$%^&*()_+-={}[]|:";'<>?,./\ danced in debug reports. A hidden code read: "UnL0cK_Me!" and a timestamp symbolizes the void. Is ∞ truly infinite? Queries continued, while "QWERTY123" echoed.

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8



RESULT:

Thus, The Huffman encoding and decoding to compress and decompress text messages is implemented and output verified after successful execution.

AIM:

The aim of this code is to implement arithmetic encoding, a lossless data compression technique that represents an entire message as a single number between 0 and 1, based on the cumulative frequencies of characters, achieving efficient data representation.

ALGORITHM:

Step 1: Read the original message from a file (input_message.txt).

Step 2: Calculate the frequency of each character in the message.

Step 3: Compute the cumulative frequencies of characters to create intervals:

i. Normalize the frequencies to get probabilities.
ii. Calculate cumulative probabilities to determine the range of each character.

Step 4: Initialize lower and upper bounds for the interval (e.g., low = 0.0, high = 1.0).

Step 5: For each character in the message:

i. Update the bounds based on the character's cumulative probability:
ii. Calculate the new upper and lower bounds using the character's interval.

Step 6: The final interval after processing all characters represents the encoded value.Choose a number within this interval as the encoded output.

Step 7: Save the encoded value to a file (encoded_message.arithmetic).

Step 8: Decode the message by using the encoded value to determine the characters:

i. Initialize the same cumulative frequency intervals.
ii. Use the encoded value to find which character’s interval it falls into, iteratively reconstructing the original message.

Step 9: Save the decoded message to a file (decoded_message.txt).

Step 10: Verify that the decoded message matches the original message, confirming successful encoding and decoding.

SOURCE CODE:

```
from collections import defaultdict
from decimal import Decimal, getcontext
import os
getcontext().prec = 500
def calculate_ranges(message):
```

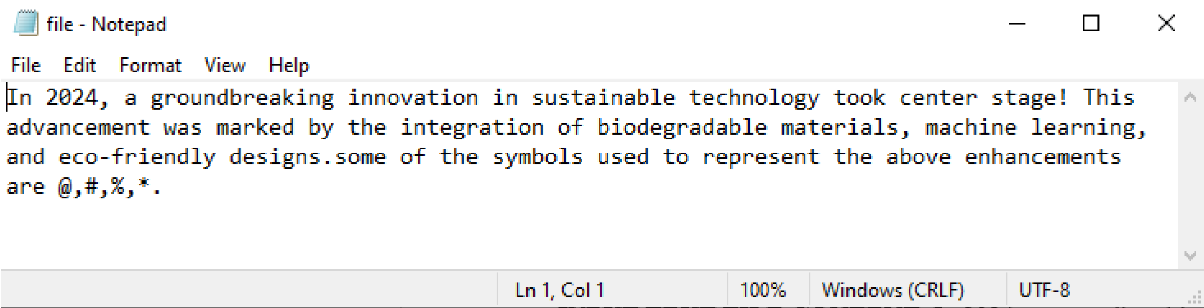
```
    frequency = defaultdict(int)
    for char in message:
        frequency[char] += 1
    total_chars = len(message)
    ranges = {}
    lower_bound = Decimal(0)
    for char, count in frequency.items():
        ranges[char] = (lower_bound / total_chars, (lower_bound + count) / total_chars)
        lower_bound += count
    return ranges
def arithmetic_encode(message):
    ranges = calculate_ranges(message)
    low = Decimal(0.0)
    high = Decimal(1.0)
    for char in message:
        range_width = high - low
        high = low + range_width * Decimal(ranges[char][1])
        low = low + range_width * Decimal(ranges[char][0])
    return (low + high) / 2
def arithmetic_decode(encoded_value, message, ranges):
    low = Decimal(0.0)
    high = Decimal(1.0)
    decoded_message = ""
    for _ in range(len(message)):
        range_width = high - low
        value = (encoded_value - low) / range_width
        for char, (low_range, high_range) in ranges.items():
            if Decimal(low_range) <= value < Decimal(high_range):
                decoded_message += char
                high = low + range_width * Decimal(high_range)
                low = low + range_width * Decimal(low_range)
                break
    return decoded_message
def main():
    input_file = input("Enter the text file name: ").strip()
    if not os.path.isfile(input_file):
        print("Error: File not found.")
        return
    with open(input_file, 'r') as file:
        message = file.read().strip()
    print("Original message successfully loaded.")
    ranges = calculate_ranges(message)
    encoded_value = arithmetic_encode(message)
```

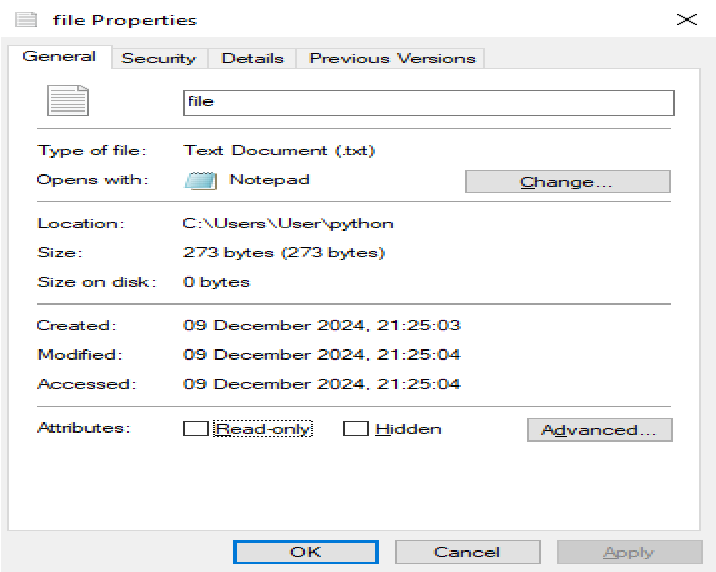
```
    folder_name = "output_files"
    if not os.path.exists(folder_name):
        os.makedirs(folder_name)
    encoded_file = os.path.join(folder_name, 'encoded_value.arith')
    with open(encoded_file, 'w') as file:
        file.write(str(encoded_value))
    print(f"Encoded message saved to {encoded_file}")
    decoded_message = arithmetic_decode(Decimal(encoded_value), message, ranges)
    decoded_file = os.path.join(folder_name, 'decoded_message.txt')
    with open(decoded_file, 'w') as file:
        file.write(decoded_message)
    print(f"Decoded message saved to {decoded_file}")
    input_size = len(message.encode('utf-8'))
    output_size = len(decoded_message.encode('utf-8'))
    if message == decoded_message:
        print(f"Success! The input file size ({input_size} bytes) matches the output file size
({output_size} bytes).")
        print("SUCCESS!!! THE ORIGINAL MESSAGE AND THE DECODED MESSAGE
ARE **EQUAL**.")
    else:
        print(f"Error! The input file size ({input_size} bytes) does not match the output file size
({output_size} bytes).")
        print("ERROR!!! THE ORIGINAL MESSAGE AND THE DECODED MESSAGE
ARE NOT **EQUAL**.")
if __name__ == "__main__":
    main()
```

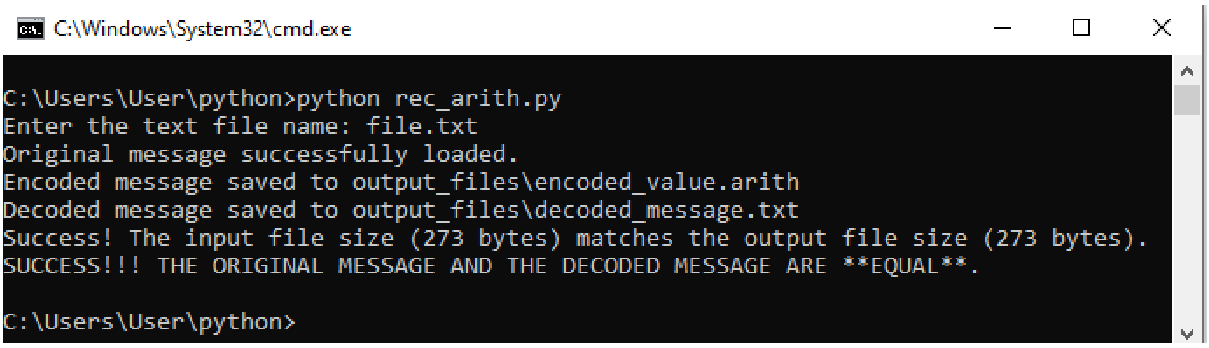
INPUT AND OUTPUT:

INPUT TEXT FILE CONTENT: In 2024, a groundbreaking innovation in sustainable technology took center stage! This advancement was marked by the integration of biodegradable materials, machine learning, and eco-friendly designs.some of the symbols used to represent the above enhancements are @,#,%,*.

INPUT FILE: – file2.txt







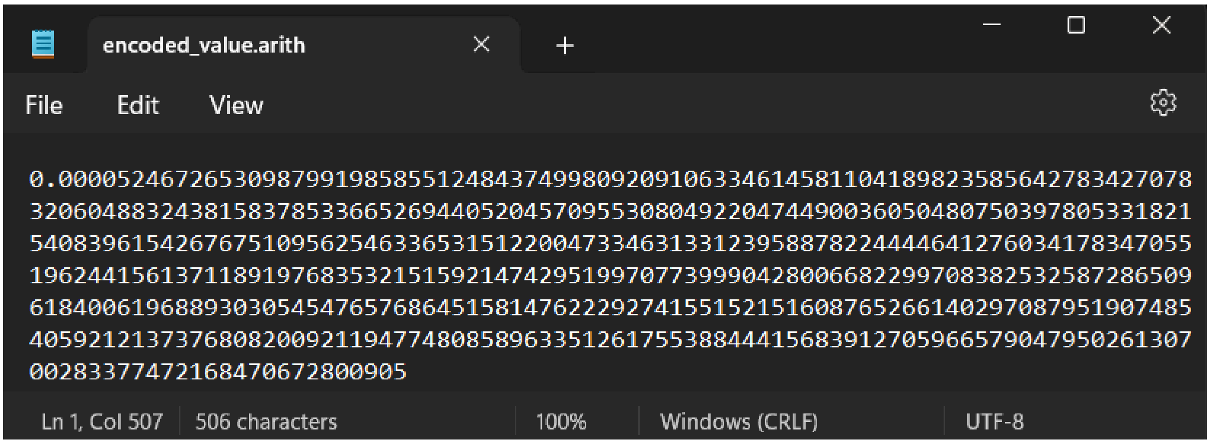
OUTPUT FILES:

Encoded message:

0.000052467265309879919858551248437499809209106334614581104189823585642783427078320604883243815837853366526944052045709553080492204744900360504807503978053318215408396154267675109562546336531512200473346313312395887822444641

27603417834705519624415613711891976835321515921474295199707739990428006682299708382532587286509618400619688930305454765768645158147622292741551521516087652661402970879519074854059212137376808200921194774808589633512617553884441568391270596657904795026130700283377472168470672800905

Encoded file:



encoded_value.arith

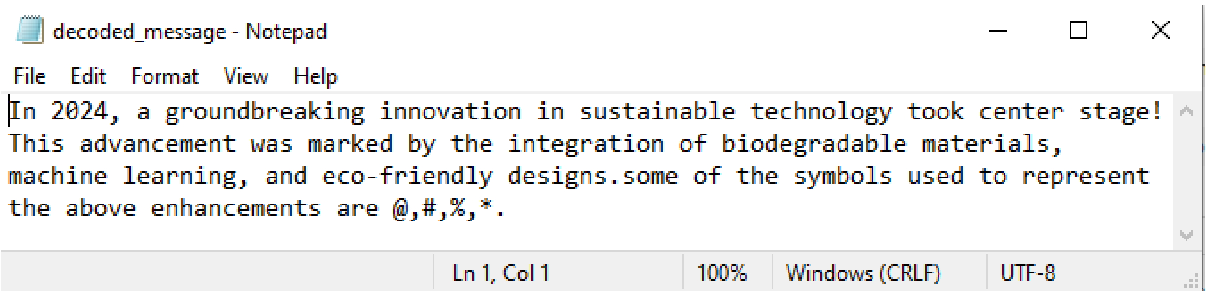
File Edit View

```
0.000052467265309879919858551248437499809209106334614581104189823585642783427078
320604883243815837853366526944052045709553080492204744900360504807503978053318215
408396154267675109562546336531512200473346313312395887822444464127603417834705
519624415613711891976835321515921474295199707739990428006682299708382532587286509
618400619688930305454765768645158147622292741551521516087652661402970879519074854
059212137376808200921194774808589633512617553884441568391270596657904795026130700
283377472168470672800905
```

Ln 1, Col 507 | 506 characters | 100% | Windows (CRLF) | UTF-8

Decoded message: In 2024, a groundbreaking innovation in sustainable technology took center stage! This advancement was marked by the integration of biodegradable materials, machine learning, and eco-friendly designs.some of the symbols used to represent the above enhancements are @,#,%,*.

Decoded file:

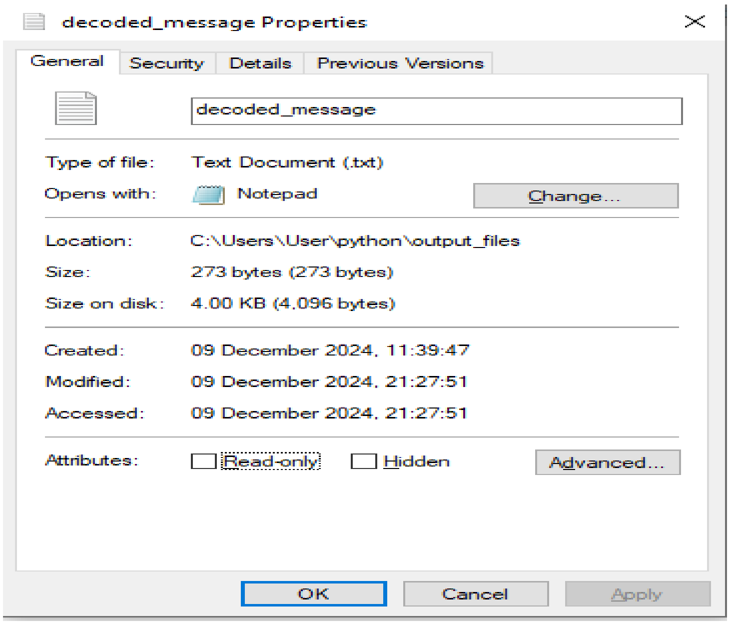


decoded_message - Notepad

File Edit Format View Help

```
In 2024, a groundbreaking innovation in sustainable technology took center stage!
This advancement was marked by the integration of biodegradable materials,
machine learning, and eco-friendly designs.some of the symbols used to represent
the above enhancements are @,#,%,*.
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8



RESULT:

Thus, The Arithmetic encoding and decoding to compress and decompress text messages is implemented and output verified after successful execution.

AIM:

To implement the Run-Length Encoding (RLE) compression algorithm. The goal is to take an input string and compress it by replacing consecutive repeated characters with a single character followed by the count of its repetitions.

ALGORITHM:

Step 1: Initialize an empty list called encoding.

Step 2: Set an index variable i to 0.

Step 3: While i is less than the length of the data, do the following:

i. Set a count variable to 1.

ii. While the next character is the same as the current character, increment count and i.

iii. Append a tuple of the current character and its count to encoding.

iv. Increment i.

Step 4: Return the encoding list.

SOURCE CODE:

```
import os
def run_length_encoding(data):
    encoding = []
    i = 0
    while i < len(data):
        count = 1
        while i + 1 < len(data) and data[i] == data[i + 1]:
            count += 1
            i += 1
        encoding.append((data[i], count))
        i += 1
    return encoding
def save_encoded_data(encoded_data, output_folder):
    output_file = os.path.join(output_folder, 'encoded_message.txt')
    with open(output_file, 'w', encoding='utf-8') as file:
        encoded_string = ', '.join(f"{char}:{count}" for char, count in encoded_data)
        file.write(encoded_string)
    return output_file
def main():
    file_path = input("Enter the path of the input file: ").strip()
    try:
        if not os.path.isfile(file_path):
            print(f"Error: The file '{file_path}' was not found.")
            return
        with open(file_path, 'r') as file:
            data = file.read().strip()
        print("Original message successfully loaded.")
        encoded_data = run_length_encoding(data)
        output_folder = "output_files"
        if not os.path.exists(output_folder):
            os.makedirs(output_folder)
        encoded_file = save_encoded_data(encoded_data, output_folder)
        print(f"Encoded message saved to {encoded_file}")
        decoded_message = ''.join(char * count for char, count in encoded_data)
        decoded_file = os.path.join(output_folder, 'decoded_message.txt')
        with open(decoded_file, 'w') as file:
            file.write(decoded_message)
        print(f"Decoded message saved to {decoded_file}")
        input_size = len(data.encode('utf-8'))
        output_size = len(decoded_message.encode('utf-8'))
```

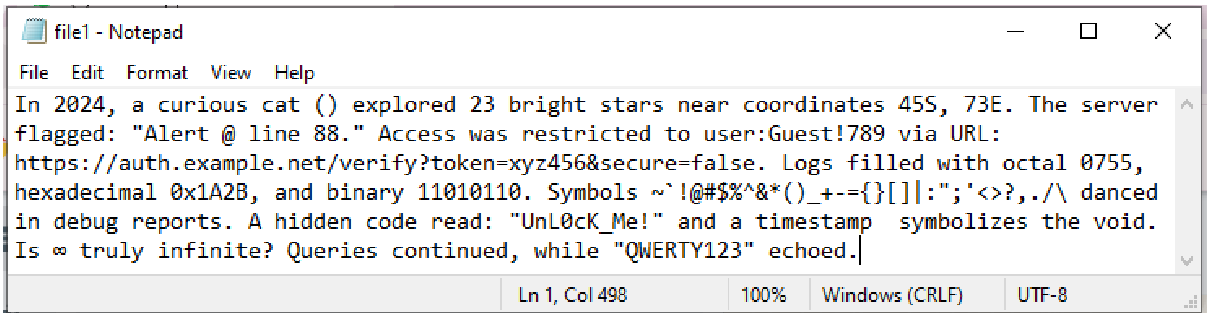
```
        if data == decoded_message:
            print(f"Success! The input file size ({input_size} bytes) matches the output file size
({output_size} bytes).")
            print("SUCCESS!!! THE ORIGINAL MESSAGE AND THE DECODED
MESSAGE ARE **EQUAL**.")
        else:
            print(f"Error! The input file size ({input_size} bytes) does not match the output file
size ({output_size} bytes).")
            print("ERROR!!! THE ORIGINAL MESSAGE AND THE DECODED MESSAGE
ARE NOT **EQUAL**.")
    except FileNotFoundError:
        print(f"Error: The file '{file_path}' was not found.")
    except Exception as e:
        print(f"An error occurred: {e}")
if __name__ == "__main__":
    main()
```

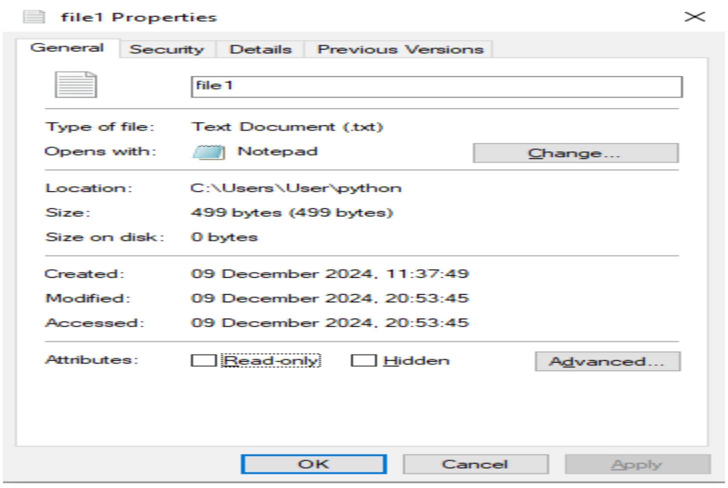
INPUT AND OUTPUT:

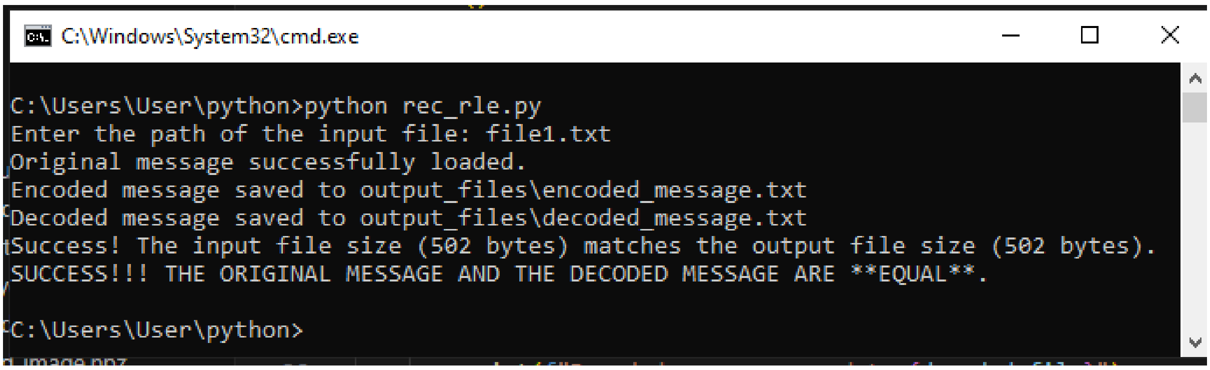
INPUT TEXT FILE CONTENT:

In 2024, a curious cat () explored 23 bright stars near coordinates 45S, 73E. The server flagged: "Alert @ line 88." Access was restricted to user:Guest!789 via URL: https://auth.example.net/verify?token=xyz456&secure=false. Logs filled with octal 0755, hexadecimal 0x1A2B, and binary 11010110. Symbols ~`!@#$%^&*()_+-={}[]|:";'<>?,./\ danced in debug reports. A hidden code read: "UnL0cK_Me!" and a timestamp symbolizes the void. Is ∞ truly infinite? Queries continued, while "QWERTY123" echoed.

INPUT FILE: – file1.txt







OUTPUT FILES:

Encoded message:

I:1, n:1, :1, 2:1, 0:1, 2:1, 4:1, ,:1, :1, a:1, :1, c:1, u:1, r:1, i:1, o:1, u:1, s:1, :1, c:1, a:1, t:1, :1, (:1,):1, :1, e:1, x:1, p:1, l:1, o:1, r:1, e:1, d:1, :1, 2:1, 3:1, :1, b:1, r:1, i:1, g:1, h:1, t:1, :1, s:1, t:1, a:1, r:1, s:1, :1, n:1, e:1, a:1, r:1, :1, c:1, o:2, r:1, d:1, i:1, n:1, a:1, t:1, e:1, s:1, :1, 4:1, 5:1, S:1, ,:1, :1, 7:1, 3:1, E:1, .:1, :1, T:1, h:1, e:1, :1, s:1, e:1, r:1, v:1, e:1, r:1, :1, f:1, l:1, a:1, g:2, e:1, d:1, ::1, :1, ":1, A:1, l:1, e:1, r:1, t:1, :1, @:1, :1, l:1, i:1, n:1, e:1, :1, 8:2, .:1, ":1, :1, A:1, c:2, e:1, s:2, :1, w:1, a:1, s:1, :1, r:1, e:1, s:1, t:1, r:1, i:1, c:1, t:1, e:1, d:1, :1, t:1, o:1, :1, u:1, s:1, e:1, r:1, ::1, G:1, u:1, e:1, s:1, t:1, !:1, 7:1, 8:1, 9:1, :1, v:1, i:1, a:1, :1, U:1, R:1, L:1, ::1, :1, h:1, t:2, p:1, s:1, ::1, /:2, a:1, u:1, t:1, h:1, .:1, e:1, x:1, a:1, m:1, p:1, l:1, e:1, .:1, n:1, e:1, t:1, /:1, v:1, e:1, r:1, i:1, f:1, y:1, ?:1, t:1, o:1, k:1, e:1, n:1, =:1, x:1, y:1, z:1, 4:1, 5:1, 6:1, &:1, s:1, e:1, c:1, u:1, r:1, e:1, =:1, f:1, a:1, l:1, s:1, e:1, .:1, :1, L:1, o:1, g:1, s:1, :1, f:1, i:1, l:2, e:1, d:1, :1, w:1, i:1, t:1, h:1, :1, o:1, c:1, t:1, a:1, l:1, :1, 0:1, 7:1, 5:2, ,:1, :1, h:1, e:1, x:1, a:1, d:1, e:1, c:1, i:1, m:1, a:1, l:1, :1, 0:1, x:1, 1:1, A:1, 2:1, B:1, ,:1, :1, a:1, n:1, d:1, :1, b:1, i:1, n:1, a:1, r:1, y:1, :1, 1:2, 0:1, 1:1, 0:1, 1:2, 0:1, .:1, :1, S:1, y:1, m:1, b:1, o:1, l:1, s:1, :1, ~:1, `:1, !:1, @:1, #:1, $:1, %:1, ^:1, &:1, *:1, (:1,):1, _:1, +:1, -:1, =:1, {:1, }:1, [:1,]:1, |:1, ::1, ":1, ;:1, ':1, <:1, >:1, ?:1, ,:1, .:1, /:1, \:1, :1, d:1, a:1, n:1, c:1, e:1, d:1, :1, i:1, n:1, :1, d:1, e:1, b:1, u:1, g:1, :1, r:1, e:1, p:1, o:1, r:1, t:1, s:1, .:1, :1, A:1, :1, h:1, i:1, d:2, e:1, n:1, :1, c:1, o:1, d:1, e:1, :1, r:1, e:1, a:1, d:1, ::1, :1, ":1, U:1, n:1, L:1, 0:1, c:1, K:1, _:1, M:1, e:1, !:1, ":1, :1, a:1, n:1, d:1, :1, a:1, :1, t:1, i:1, m:1, e:1, s:1, t:1, a:1, m:1, p:1, :2, s:1, y:1, m:1, b:1, o:1, l:1, i:1, z:1, e:1, s:1, :1, t:1, h:1, e:1, :1, v:1, o:1, i:1, d:1, .:1, :1, I:1, s:1, :1, â:1, ˆ:1, ž:1, :1, t:1, r:1, u:1, l:1, y:1, :1, i:1, n:1, f:1, i:1, n:1, i:1, t:1, e:1, ?:1, :1, Q:1, u:1, e:1, r:1, i:1, e:1, s:1, :1, c:1, o:1, n:1, t:1, i:1, n:1, u:1, e:1, d:1, ,:1, :1, w:1, h:1, i:1, l:1, e:1, :1, ":1, Q:1, W:1, E:1, R:1, T:1, Y:1, 1:1, 2:1, 3:1, ":1, :1, e:1, c:1, h:1, o:1, e:1, d:1, .:1

Encoded file:

encoded_message - Notepad

File Edit Format View Help

```
I:1, n:1,  :1, 2:1, 0:1, 2:1, 4:1, ,:1,  :1, a:1,  :1, c:1, u:1, r:1, i:1, o:1, u:1, s:1,  :1, c:1, a:1, t:1,  :1, (:1,
):1,  :1, e:1, x:1, p:1, l:1, o:1, r:1, e:1, d:1,  :1, 2:1, 3:1,  :1, b:1, r:1, i:1, g:1, h:1, t:1,  :1, s:1, t:1, a:1,
r:1, s:1,  :1, n:1, e:1, a:1, r:1,  :1, c:1, o:2, r:1, d:1, i:1, n:1, a:1, t:1, e:1, s:1,  :1, 4:1, 5:1, S:1, ,:1,  :1,
7:1, 3:1, E:1, .:1,  :1, T:1, h:1, e:1,  :1, s:1, e:1, r:1, v:1, e:1, r:1,  :1, f:1, l:1, a:1, g:2, e:1, d:1, ::1,  :1,
":1, A:1, l:1, e:1, r:1, t:1,  :1, @:1,  :1, l:1, i:1, n:1, e:1,  :1, 8:2, .:1, ":1,  :1, A:1, c:2, e:1, s:2,  :1, w:1,
a:1, s:1,  :1, r:1, e:1, s:1, t:1, r:1, i:1, c:1, t:1, e:1, d:1,  :1, t:1, o:1,  :1, u:1, s:1, e:1, r:1, ::1, G:1, u:1,
e:1, s:1, t:1, !:1, 7:1, 8:1, 9:1,  :1, v:1, i:1, a:1,  :1, U:1, R:1, L:1, ::1,  :1, h:1, t:2, p:1, s:1, ::1, /:2, a:1,
u:1, t:1, h:1, .:1, e:1, x:1, a:1, m:1, p:1, l:1, e:1, .:1, n:1, e:1, t:1, /:1, v:1, e:1, r:1, i:1, f:1, y:1, ?:1, t:1,
o:1, k:1, e:1, n:1, =:1, x:1, y:1, z:1, 4:1, 5:1, 6:1, &:1, s:1, e:1, c:1, u:1, r:1, e:1, =:1, f:1, a:1, l:1, s:1, e:1,
.:1,  :1, L:1, o:1, g:1, s:1,  :1, f:1, i:1, l:2, e:1, d:1,  :1, w:1, i:1, t:1, h:1,  :1, o:1, c:1, t:1, a:1, l:1,  :1,
0:1, 7:1, 5:2, ,:1,  :1, h:1, e:1, x:1, a:1, d:1, e:1, c:1, i:1, m:1, a:1, l:1,  :1, 0:1, x:1, 1:1, A:1, 2:1, B:1, ,:1,
:1, a:1, n:1, d:1,  :1, b:1, i:1, n:1, a:1, r:1, y:1,  :1, 1:2, 0:1, 1:1, 0:1, 1:2, 0:1, .:1,  :1, S:1, y:1, m:1, b:1,
o:1, l:1, s:1,  :1, ~:1, `:1, !:1, @:1, #:1, $:1, %:1, ^:1, &:1, *:1, (:1, ):1, _:1, +:1, -:1, =:1, {:1, }:1, [:1, ]:1,
|:1, ::1, ":1, ;:1, ':1, <:1, >:1, ?:1, ,:1, .:1, /:1, \:1,  :1, d:1, a:1, n:1, c:1, e:1, d:1,  :1, i:1, n:1,  :1, d:1,
e:1, b:1, u:1, g:1,  :1, r:1, e:1, p:1, o:1, r:1, t:1, s:1, .:1,  :1, A:1,  :1, h:1, i:1, d:2, e:1, n:1,  :1, c:1, o:1,
d:1, e:1,  :1, r:1, e:1, a:1, d:1, ::1,  :1, ":1, U:1, n:1, L:1, 0:1, c:1, K:1, _:1, M:1, e:1, !:1, ":1,  :1, a:1, n:1,
d:1,  :1, a:1,  :1, t:1, i:1, m:1, e:1, s:1, t:1, a:1, m:1, p:1,  :2, s:1, y:1, m:1, b:1, o:1, l:1, i:1, z:1, e:1, s:1,
:1, t:1, h:1, e:1,  :1, v:1, o:1, i:1, d:1, .:1,  :1, I:1, s:1,  :1, â:1, ˆ:1, ž:1,  :1, t:1, r:1, u:1, l:1, y:1,  :1,
i:1, n:1, f:1, i:1, n:1, i:1, t:1, e:1, ?:1,  :1, Q:1, u:1, e:1, r:1, i:1, e:1, s:1,  :1, c:1, o:1, n:1, t:1, i:1, n:1,
u:1, e:1, d:1, ,:1,  :1, w:1, h:1, i:1, l:1, e:1,  :1, ":1, Q:1, W:1, E:1, R:1, T:1, Y:1, 1:1, 2:1, 3:1, ":1,  :1, e:1,
c:1, h:1, o:1, e:1, d:1, .:1
```

Decoded message:

In 2024, a curious cat () explored 23 bright stars near coordinates 45S, 73E. The server flagged: "Alert @ line 88." Access was restricted to user:Guest!789 via URL: https://auth.example.net/verify?token=xyz456&secure=false. Logs filled with octal 0755, hexadecimal 0x1A2B, and binary 11010110. Symbols ~`!@#$%^&*()_+-={}[]|:";'<>?,./\ danced in debug reports. A hidden code read: "UnL0cK_Me!" and a timestamp symbolizes the void. Is ∞ truly infinite? Queries continued, while "QWERTY123" echoed.

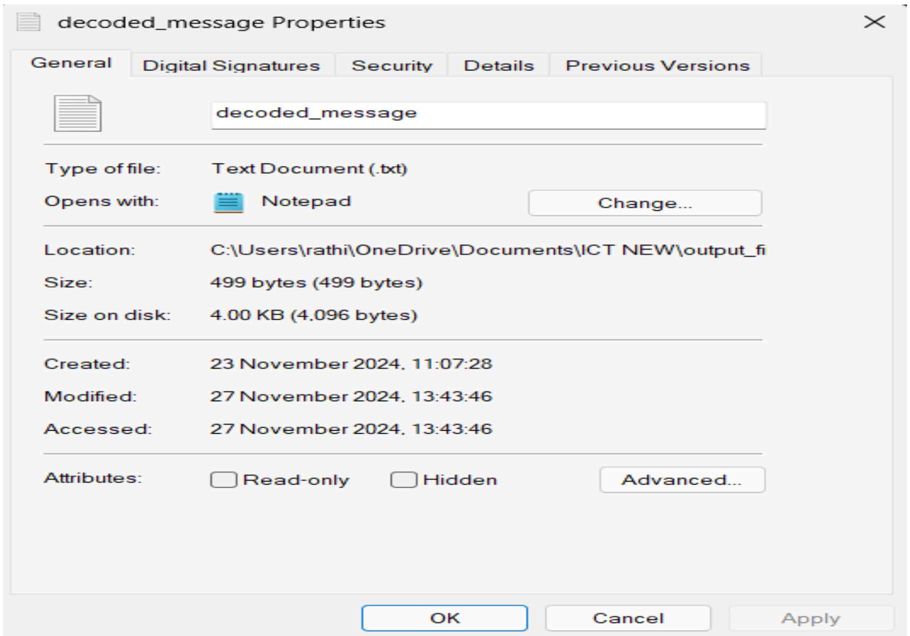
Decoded file:

decoded_message - Notepad

File Edit Format View Help

```
In 2024, a curious cat () explored 23 bright stars near coordinates 45S, 73E. The server flagged:
"Alert @ line 88." Access was restricted to user:Guest!789 via URL:
https://auth.example.net/verify?token=xyz456&secure=false. Logs filled with octal 0755,
hexadecimal 0x1A2B, and binary 11010110. Symbols ~`!@#$%^&*()_+-={}[]|:";'<>?,./\ danced in debug
reports. A hidden code read: "UnL0cK_Me!" and a timestamp  symbolizes the void. Is ∞ truly
infinite? Queries continued, while "QWERTY123" echoed.
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8



RESULT:

Thus, the implementation efficiently compresses the input string using the Run-Length Encoding algorithm, and the output can be useful for reducing the size of data with repetitive sequences, making it suitable for applications like image compression and simple data transmission.

AIM:

To implement the Dictionary based coding algorithm, which is a lossless data compression method which take an input string, compress it using the algorithm, and output a list of encoded values that represent the compressed data.

ALGORITHM:

Step 1: Initialize the dictionary with ASCII characters (0-255).

Step 2: Set current_str to an empty string.

Step 3: Initialize compressed_data as an empty list.

Step 4: Set dict_size to 256.

Step 5: Read the input data character by character.

Step 6: For each character, combine it with current_str to form combined_str.

Step 7: Check if combined_str exists in the dictionary:

i. If yes, update current_str.

ii. If no, append the value of current_str to compressed_data, add combined_str to the dictionary, and update current_str.

Step 8: If current_str is not empty, append its value to compressed_data.

SOURCE CODE:

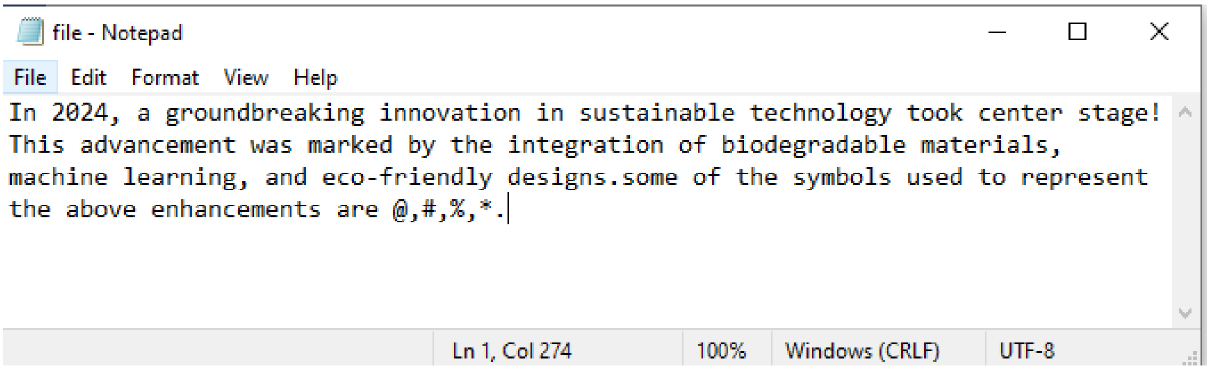
```
import os
def lzw_encode(data):
    """Compress a string to a list of output symbols using LZW algorithm."""
    dictionary = {chr(i): i for i in range(256)}
    dict_size = 256
    result = []
    w = ""
    for c in data:
        wc = w + c
        if wc in dictionary:
            w = wc
        else:
            result.append(dictionary[w])
            dictionary[wc] = dict_size
            dict_size += 1
            w = c
    if w:
        result.append(dictionary[w])
    return result
def lzw_decode(compressed):
    """Decompress a list of output symbols to a string using LZW algorithm."""
    dictionary = {i: chr(i) for i in range(256)}
    dict_size = 256
    w = chr(compressed.pop(0))
    result = w
    for k in compressed:
        if k in dictionary:
            entry = dictionary[k]
        elif k == dict_size:
            entry = w + w[0]
        else:
            raise ValueError("Bad compressed k: %s" % k)
        result += entry
        dictionary[dict_size] = w + entry[0]
        dict_size += 1
        w = entry
    return result
def main():
    input_file = input("Enter the text file name: ")
    with open(input_file, 'r') as file:
        message = file.read().strip()
    print("Original message successfully loaded.")
```

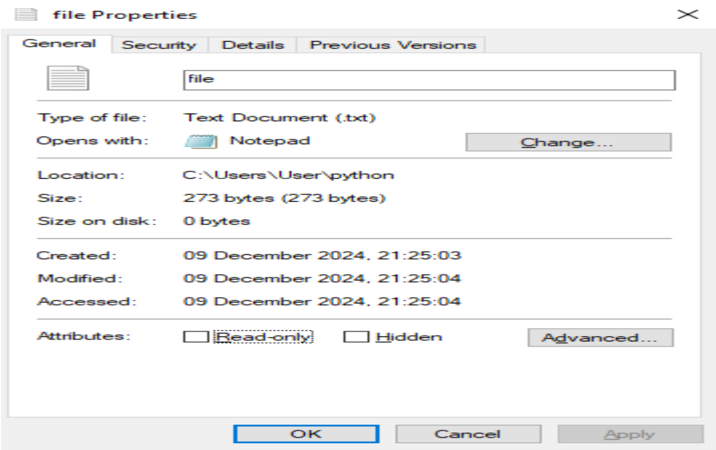
```
    compressed = lzw_encode(message)
    folder_name = "output_files"
    if not os.path.exists(folder_name):
        os.makedirs(folder_name)
    encoded_file = os.path.join(folder_name, 'encoded_message.lzw')
    with open(encoded_file, 'w') as file:
        file.write(' '.join(map(str, compressed)))
    print(f"Encoded message saved to {encoded_file}")
   decoded_message = lzw_decode(compressed)
    decoded_file = os.path.join(folder_name, 'decoded_message.txt')
    with open(decoded_file, 'w') as file:
        file.write(decoded_message)
    print(f"Decoded message saved to {decoded_file}")
    input_size = len(message)
    output_size = len(decoded_message)
    if input_size == output_size and message == decoded_message:
        print(f"Success! The input file size ({input_size} bytes) matches the output file size
({output_size} bytes).")
        print("SUCCESS!!! THE ORIGINAL MESSAGE AND THE DECODED MESSAGE
ARE **EQUAL**.")
    else:
        print(f"Error! The input file size ({input_size} bytes) does not match the output file size
({output_size} bytes).")
        print("THE ORIGINAL MESSAGE AND THE DECODED MESSAGE ARE NOT
**EQUAL**.")
if __name__ == "__main__":
    main()
```

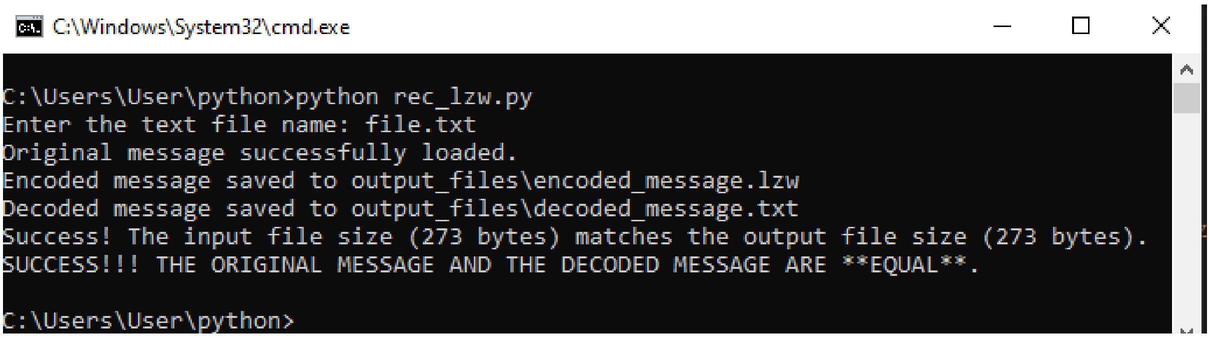
INPUT AND OUTPUT:
INPUT TEXT FILE CONTENT:

In 2024, a groundbreaking innovation in sustainable technology took center stage! This advancement was marked by the integration of biodegradable materials, machine learning, and eco-friendly designs.some of the symbols used to represent the above enhancements are @,#,%,*.

INPUT FILE: – file1.txt







OUTPUT FILES:

Encoded message:

73 110 32 50 48 50 52 44 32 97 32 103 114 111 117 110 100 98 114 101 97 107 105 110 103 32 278 110 111 118 97 116 105 111 257 278 32 115 117 115 116 97 278 97 98 108 101 32

116 101 99 104 283 108 111 103 121 303 111 111 107 32 99 101 110 304 114 292 296 103 101 33 32 84 104 105 115 264 100 285 110 318 109 319 116 32 119 97 332 109 97 114 107 101 100 32 98 312 116 104 302 278 304 267 286 288 257 111 102 351 288 100 101 359 100 299 301 32 345 321 105 97 108 115 263 345 306 278 302 301 346 110 278 103 263 97 271 32 305 111 45 102 114 105 319 100 108 312 367 115 105 103 110 115 46 115 111 338 32 363 303 355 292 121 109 98 111 378 32 294 349 313 32 274 112 274 115 339 416 302 299 284 302 319 104 391 337 339 332 346 302 64 44 35 44 37 44 42 46

Encoded file:

encoded_message.lzw - Notepad

File Edit Format View Help

```
73 110 32 50 48 50 52 44 32 97 32 103 114 111 117 110 100 98 114 101 97 107 105 110 103 32 278 110 111
118 97 116 105 111 257 278 32 115 117 115 116 97 278 97 98 108 101 32 116 101 99 104 283 108 111 103 121
303 111 111 107 32 99 101 110 304 114 292 296 103 101 33 32 84 104 105 115 264 100 285 110 318 109 319
116 32 119 97 332 109 97 114 107 101 100 32 98 312 116 104 302 278 304 267 286 288 257 111 102 351 288
100 101 359 100 299 301 32 345 321 105 97 108 115 263 345 306 278 302 301 346 110 278 103 263 97 271 32
305 111 45 102 114 105 319 100 108 312 367 115 105 103 110 115 46 115 111 338 32 363 303 355 292 121 109
98 111 378 32 294 349 313 32 274 112 274 115 339 416 302 299 284 302 319 104 391 337 339 332 346 302 64
44 35 44 37 44 42 46
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

Decoded message:

In 2024, a groundbreaking innovation in sustainable technology took center stage! This advancement was marked by the integration of biodegradable materials, machine learning, and eco-friendly designs.some of the symbols used to represent the above enhancements are @,#,%,*.

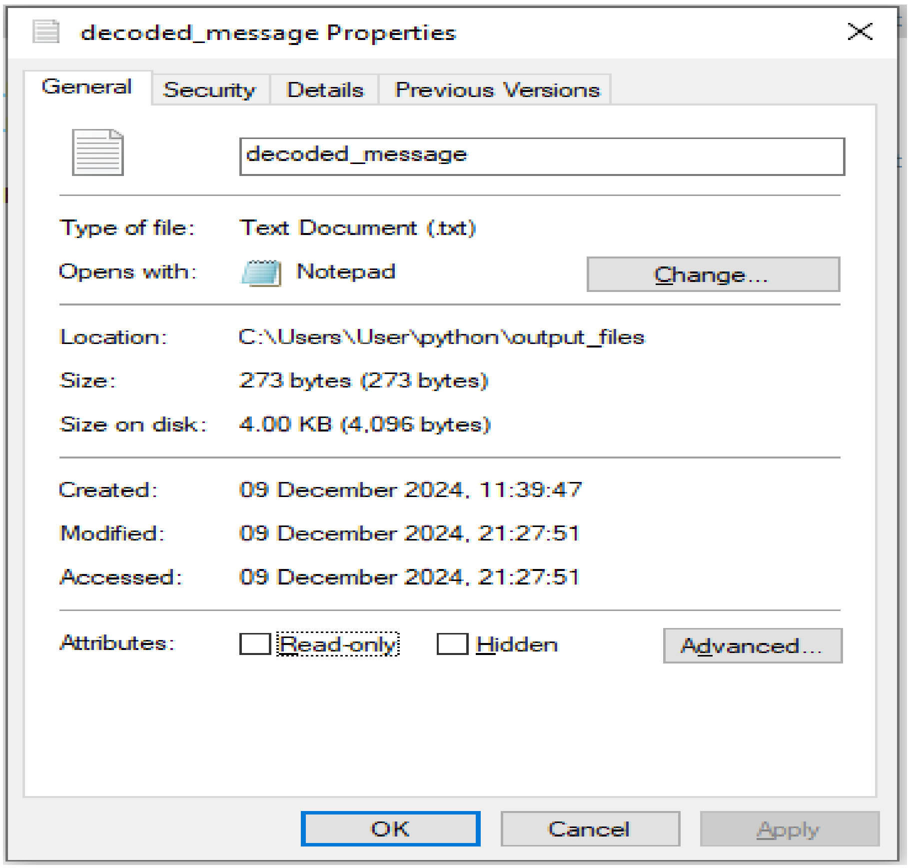
Decoded file:

decoded_message - Notepad

File Edit Format View Help

```
In 2024, a groundbreaking innovation in sustainable technology took center
stage! This advancement was marked by the integration of biodegradable
materials, machine learning, and eco-friendly designs.some of the symbols
used to represent the above enhancements are @,#,%,*.
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8



RESULT:

Thus, The implementation of dictionary-based coding, specifically using the Lempel-Ziv-Welch (LZW) algorithm, efficiently compresses input data by constructing a dictionary of unique patterns encountered in the data stream. The algorithm replaces repetitive sequences with shorter codes from the dictionary, significantly reducing the size of data with frequent redundancies.

AIM:

To implement Linear Predictive Coding (LPC) for audio signal processing, encoding an audio signal to extract LPC coefficients and decoding these coefficients to reconstruct the original audio signal.

ALGORITHM:

Step 1: Read the input WAV file specified by the user.

Step 2: Convert the audio signal to mono if it is stereo.

Step 3: Compute the autocorrelation of the audio signal.

Step 4: Use the Levinson-Durbin recursion to calculate LPC from the autocorrelation.

Step 5: Save the LPC coefficients to a file.

Step 6: Synthesize the audio signal from the LPC coefficients, using noise for excitation.

Step 7: Normalize the reconstructed signal and convert it to 16-bit PCM format.

Step 8: Save the reconstructed signal to a new WAV file.

SOURCE CODE:

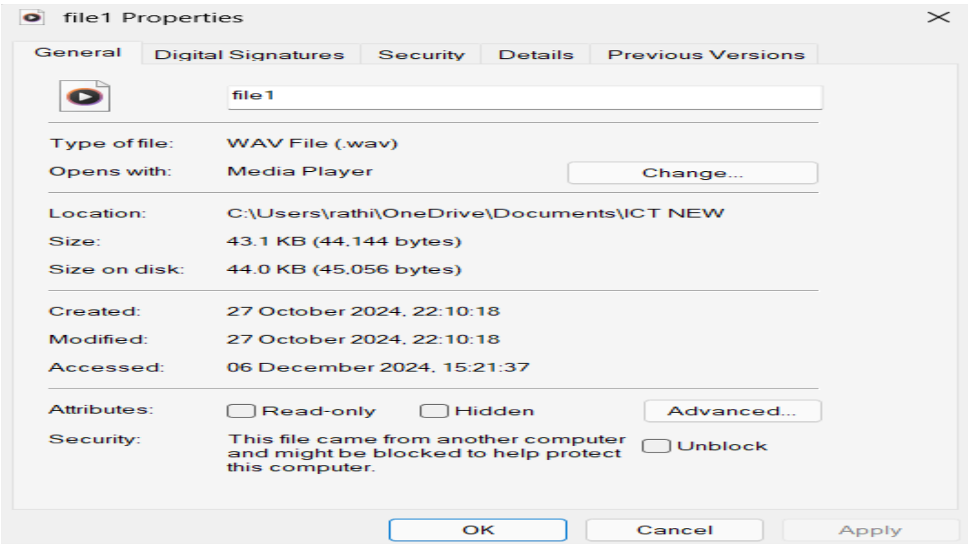
```
import numpy as np
import wave
import struct
from scipy.io import wavfile
def levinson_durbin(r, order):
    a = np.zeros(order + 1)
    e = r[0]
    a[0] = 1
    for i in range(1, order + 1):
        acc = 0
        for j in range(1, i):
            acc += a[j] * r[i - j]
                k = (r[i] - acc) / e
        new_a = a.copy()
        for j in range(1, i):
            new_a[j] = a[j] - k * a[i - j]
        new_a[i] = k
        a = new_a
        e *= (1 - k ** 2)
    return a
def lpc(signal, order):
    autocorr = np.correlate(signal, signal, mode='full')[len(signal)-1:]
    R = autocorr[:order + 1]
    a_coeffs = levinson_durbin(R, order)
    return a_coeffs
def encode_lpc(signal, order, filename):
    coefficients = lpc(signal, order)
    with open(filename, 'w') as f:
        f.write(' '.join(map(str, coefficients)))
    return coefficients
def decode_lpc(coefficients, length, noise_scale=0.1):
    signal = np.zeros(length)
    for n in range(length):
        acc = np.random.normal(0, noise_scale)
        for i in range(1, len(coefficients)):
            if n - i >= 0:
                acc -= coefficients[i] * signal[n - i]
        signal[n] = np.clip(acc, -1e4, 1e4)
    return signal
input_wav = input("Enter the path of the input WAV file: ")
sample_rate, signal = wavfile.read(input_wav)
if signal.ndim > 1:
    signal = signal[:, 0]
order = 10
```

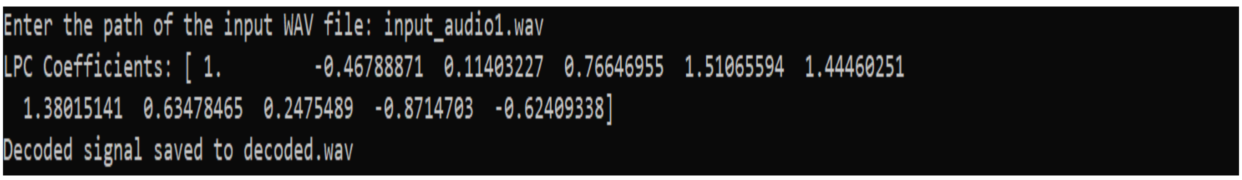
```
encoded_file = 'encoded.lpc'
coefficients = encode_lpc(signal, order, encoded_file)
decoded_signal = decode_lpc(coefficients, len(signal))
decoded_wav = 'decoded.wav'
decoded_signal = np.nan_to_num(decoded_signal / np.max(np.abs(decoded_signal)) *
32767, nan=0.0)
decoded_signal = np.int16(decoded_signal)
wavfile.write(decoded_wav, sample_rate, decoded_signal)
print("LPC Coefficients:", coefficients)
print("Decoded signal saved to", decoded_wav)
```

OUTPUT:

INPUT FILE:

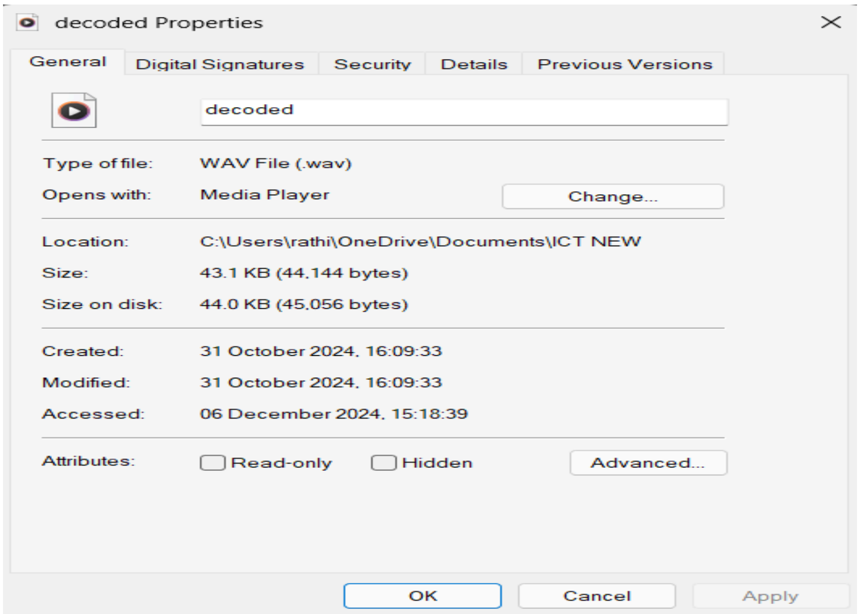
ENCODED FILE:




```
Enter the path of the input WAV file: input_audio1.wav
LPC Coefficients: [ 1.         -0.46788871  0.11403227  0.76646955  1.51065594  1.44460251
  1.38015141  0.63478465  0.2475489  -0.8714703  -0.62409338]
Decoded signal saved to decoded.wav
```

OUTPUT FILES:

DECODED FILE:



RESULT:

Thus, the implementation of Linear Predictive Coding (LPC) provides a method for analyzing and synthesizing audio signals and the process results in the calculation of LPC coefficients, which represent the linear predictive model of the audio signal.

AIM:

The aim of the code is to design and implement an image compression and decompression system that simulates the core principles of the JPEG compression standard. The system demonstrates how images can be efficiently compressed to reduce storage space while maintaining acceptable visual quality.

ALGORITHM:

STEP 1: Image Conversion: Convert the input RGB image to YCbCr color space using the rgb_to_ycbcr function.

STEP 2: Block Padding: Pad the image to ensure dimensions are multiples of 8 using pad_image function.

STEP 3: Block Processing: Divide the Y, Cb, and Cr channels into 8x8 blocks for processing.

STEP 4: DCT Transformation: Apply the Discrete Cosine Transform (DCT) to each 8x8 block using the dct function.

STEP 5: Quantization: Quantize the DCT coefficients using a predefined quantization matrix (Q_MATRIX).

STEP 6: Zigzag Ordering: Rearrange the quantized coefficients into a zigzag order using the zigzag_order function.

STEP 7: Huffman Encoding: Encode the zigzagged data using Huffman coding with the huffman_encode function.

STEP 8: Store Compressed Data: Save the encoded data and Huffman dictionaries in a structured format (e.g., JSON).

STEP 9: Decompression: To reconstruct the image:Decode Huffman-encoded data using huffman_decode.Inverse zigzag ordering using inverse_zigzag_order.Dequantize the coefficients.Apply inverse DCT to reconstruct each block.

STEP 10: Image Reconstruction: Combine processed chan nels back into YCbCr format, convert back to RGB, and save the output image.

SOURCE CODE:

```
import numpy as np
from PIL import Image
from scipy.fftpack import dct, idct
import heapq
from collections import defaultdict
import os
import json
zigzag_indices = [
    (0, 0), (0, 1), (1, 0), (2, 0), (1, 1), (0, 2), (0, 3), (1, 2),
    (2, 1), (3, 0), (4, 0), (3, 1), (2, 2), (1, 3), (0, 4), (0, 5),
    (1, 4), (2, 3), (3, 2), (4, 1), (5, 0), (6, 0), (5, 1), (4, 2),
    (3, 3), (2, 4), (1, 5), (0, 6), (0, 7), (1, 6), (2, 5), (3, 4),
    (4, 3), (5, 2), (6, 1), (7, 0), (7, 1), (6, 2), (5, 3), (4, 4),
    (3, 5), (2, 6), (1, 7), (2, 7), (3, 6), (4, 5), (5, 4), (6, 3),
    (7, 2), (7, 3), (6, 4), (5, 5), (4, 6), (3, 7), (4, 7), (5, 6),
    (6, 5), (7, 4), (7, 5), (6, 6), (5, 7), (6, 7), (7, 6), (7, 7)
]
Q_MATRIX = np.array([
    [16, 11, 10, 16, 24, 40, 51, 61],
    [12, 12, 14, 19, 26, 58, 60, 55],
    [14, 13, 16, 24, 40, 57, 69, 56],
    [14, 17, 22, 29, 51, 87, 80, 62],
    [18, 22, 37, 56, 68, 109, 103, 77],
    [24, 35, 55, 64, 81, 104, 113, 92],
    [49, 64, 78, 87, 103, 121, 120, 101],
    [72, 92, 95, 98, 112, 100, 103, 99]
])
def zigzag_order(block):
    """Apply zigzag ordering to an 8x8 block."""
    return [block[i][j] for i, j in zigzag_indices]
def inverse_zigzag_order(data, size=8):
    """Reconstruct an 8x8 block from zigzag ordering."""
    block = np.zeros((size, size))
    for idx, (i, j) in enumerate(zigzag_indices):
        block[i][j] = data[idx]
    return block
def huffman_encode(data):
    """Perform Huffman encoding."""
    if not data:
        return "", {}
```

```
    freq = defaultdict(int)
    for value in data:
        freq[value] += 1
    heap = [[weight, [symbol, ""]] for symbol, weight in freq.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
        for pair in lo[1:]:
            pair[1] = "0" + pair[1]
        for pair in hi[1:]:
            pair[1] = "1" + pair[1]
        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
    if len(heap) != 1 or not heap[0][1:]:
        raise ValueError("Invalid heap structure during Huffman encoding")
    huffman_dict = {}
    for entry in heap[0][1:]:
        if len(entry) != 2:
            raise ValueError(f"Unexpected entry in heap: {entry}")
        symbol, code = entry
        huffman_dict[symbol] = code
    encoded_data = "".join(huffman_dict[value] for value in data)
    return encoded_data, huffman_dict
def huffman_decode(encoded_data, huffman_dict):
    """Decode Huffman encoded data."""
    reverse_dict = {code: symbol for symbol, code in huffman_dict.items()}
    decoded_data = []
    buffer = ""
    for bit in encoded_data:
        buffer += bit
        if buffer in reverse_dict:
            decoded_data.append(reverse_dict[buffer])
            buffer = ""
    return decoded_data
def rgb_to_ycbcr(rgb):
    rgb = rgb.astype(np.float32)
    y = 0.299 * rgb[:,:,0] + 0.587 * rgb[:,:,1] + 0.114 * rgb[:,:,2]
    cb = -0.1687 * rgb[:,:,0] - 0.3313 * rgb[:,:,1] + 0.5 * rgb[:,:,2] + 128
    cr = 0.5 * rgb[:,:,0] - 0.4187 * rgb[:,:,1] - 0.0813 * rgb[:,:,2] + 128
    return np.stack([y, cb, cr], axis=-1)
def ycbcr_to_rgb(ycbcr):
    y = ycbcr[:,:,0]
    cb = ycbcr[:,:,1] - 128
    cr = ycbcr[:,:,2] - 128
    r = y + 1.402 * cr
    g = y - 0.34414 * cb - 0.71414 * cr
```

```
    b = y + 1.772 * cb
    return np.clip(np.stack([r, g, b], axis=-1), 0, 255).astype(np.uint8)
def pad_image(image, block_size=8):
    h, w = image.shape[:2]
    new_h = int(np.ceil(h / block_size) * block_size)
    new_w = int(np.ceil(w / block_size) * block_size)
    padded = np.zeros((new_h, new_w, image.shape[2]), dtype=image.dtype)
    padded[:h, :w, :] = image
    return padded
def process_channel(channel, quality=50):
    """Compress and decompress a single channel."""
    block_size = 8
    h, w = channel.shape
    compressed_blocks = []
    for i in range(0, h, block_size):
        for j in range(0, w, block_size):
            block = channel[i:i+block_size, j:j+block_size]
            dct_block = dct(dct(block.T, norm='ortho').T, norm='ortho')
            q_factor = 50 / quality
            quantized = np.round(dct_block / (Q_MATRIX * q_factor))
            zigzagged = zigzag_order(quantized)
            compressed_blocks.extend(zigzagged)
    return compressed_blocks
def decompress_channel(compressed_data, huffman_dict, original_shape, quality=50):
    """Decompress a single channel from compressed data."""
    block_size = 8
    h, w = original_shape
    decoded_data = huffman_decode(compressed_data, huffman_dict)
    decompressed = np.zeros((h, w), dtype=np.float32)
    block_index = 0
    for i in range(0, h, block_size):
        for j in range(0, w, block_size):
            if block_index < len(decoded_data) // (block_size * block_size):
                zigzagged_block = decoded_data[block_index * (block_size *
block_size):(block_index + 1) * (block_size * block_size)]
                quantized_block = inverse_zigzag_order(zigzagged_block)
                q_factor = 50 / quality
                dequantized_block = quantized_block * (Q_MATRIX * q_factor)
                idct_block = idct(idct(dequantized_block.T, norm='ortho').T, norm='ortho')
                decompressed[i:i+block_size, j:j+block_size] = idct_block
                block_index += 1
    return decompressed
def compress_image(input_path, quality):
    """Compress an image and return compressed data."""
    img = Image.open(input_path).convert("RGB")
    img_array = np.array(img)
```

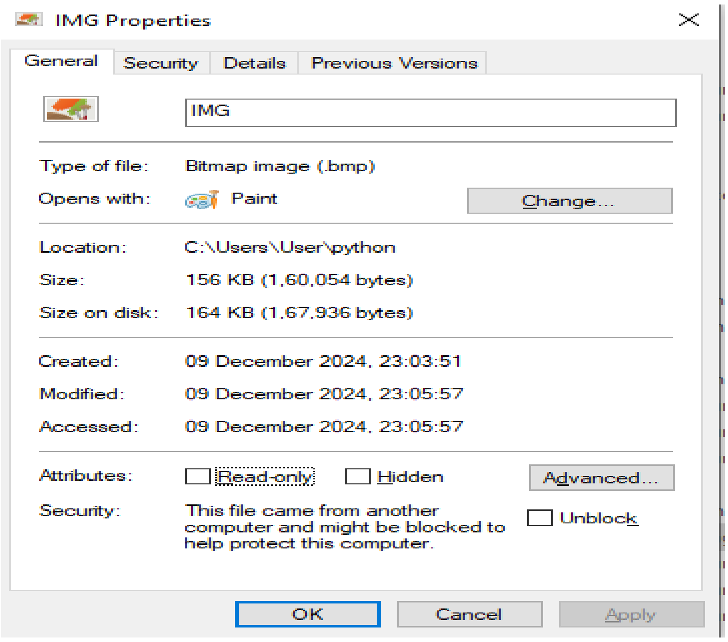
```
    ycbcr = rgb_to_ycbcr(img_array)
    y_blocks = process_channel(ycbcr[:,:,0], quality)
    cb_blocks = process_channel(ycbcr[:,:,1], quality)
    cr_blocks = process_channel(ycbcr[:,:,2], quality)
    y_encoded, y_huffman_dict = huffman_encode(y_blocks)
    cb_encoded, cb_huffman_dict = huffman_encode(cb_blocks)
    cr_encoded, cr_huffman_dict = huffman_encode(cr_blocks)
    return {
        "encoded_data": {
            "y": y_encoded, "cb": cb_encoded, "cr": cr_encoded
        },
        "huffman_dicts": {
            "y": y_huffman_dict, "cb": cb_huffman_dict, "cr": cr_huffman_dict
        },
        "original_shape": ycbcr.shape
    }
def decompress_image(compressed_data, output_path, quality):
    """Decompress an image from compressed data."""
    encoded_data = compressed_data["encoded_data"]
    huffman_dicts = compressed_data["huffman_dicts"]
    original_shape = compressed_data["original_shape"]
    y = decompress_channel(encoded_data["y"], huffman_dicts["y"], original_shape[:2],
quality)
    cb = decompress_channel(encoded_data["cb"], huffman_dicts["cb"], original_shape[:2],
quality)
    cr = decompress_channel(encoded_data["cr"], huffman_dicts["cr"], original_shape[:2],
quality)
    ycbcr = np.stack([y, cb, cr], axis=-1)
    reconstructed_rgb = ycbcr_to_rgb(ycbcr)
    Image.fromarray(reconstructed_rgb.astype(np.uint8)).save(output_path)
    print(f"Decompressed image saved to {output_path}")
if __name__ == "__main__":
    input_path = "input_image.bmp"
    compressed_data_path = "compressed_data.json"
    decompressed_output_path = "reconstructed_image.bmp"
    compressed_jpg_output_path = "compressed_image.jpg"
    quality = 50
    try:
        img = Image.open(input_path).convert("RGB")
        img_array = np.array(img)
        ycbcr = rgb_to_ycbcr(img_array)
        y_blocks = process_channel(ycbcr[:, :, 0], quality)
        cb_blocks = process_channel(ycbcr[:, :, 1], quality)
        cr_blocks = process_channel(ycbcr[:, :, 2], quality)
        print(f"Processing Y channel blocks for Huffman encoding...")
        y_encoded, y_huffman_dict = huffman_encode(y_blocks)
```

```
        print(f"Processing Cb channel blocks for Huffman encoding...")
        cb_encoded, cb_huffman_dict = huffman_encode(cb_blocks)
        print(f"Processing Cr channel blocks for Huffman encoding...")
        cr_encoded, cr_huffman_dict = huffman_encode(cr_blocks)
        compressed_data = {
            "encoded_data": {
                "y": y_encoded,
                "cb": cb_encoded,
                "cr": cr_encoded
            },
            "huffman_dicts": {
                "y": y_huffman_dict,
                "cb": cb_huffman_dict,
                "cr": cr_huffman_dict
            },
            "original_shape": ycbcr.shape
        }
        with open(compressed_data_path, "w") as f:
            json.dump(compressed_data, f)
        print(f"Compressed data saved to {compressed_data_path}")
        with open(compressed_data_path, "r") as f:
            compressed_data = json.load(f)
        y = decompress_channel(compressed_data["encoded_data"]["y"],
compressed_data["huffman_dicts"]["y"],
                               tuple(compressed_data["original_shape"][:2]), quality)
        cb = decompress_channel(compressed_data["encoded_data"]["cb"],
compressed_data["huffman_dicts"]["cb"],
                                tuple(compressed_data["original_shape"][:2]), quality)
        cr = decompress_channel(compressed_data["encoded_data"]["cr"],
compressed_data["huffman_dicts"]["cr"],
                                tuple(compressed_data["original_shape"][:2]), quality)
        ycbcr_reconstructed = np.stack([y, cb, cr], axis=-1)
        rgb_reconstructed = ycbcr_to_rgb(ycbcr_reconstructed)
        Image.fromarray(rgb_reconstructed.astype(np.uint8)).save(decompressed_output_path)
        print(f"Decompressed image saved to {decompressed_output_path}")
        compressed_jpg = Image.fromarray(rgb_reconstructed.astype(np.uint8))
        compressed_jpg.save(compressed_jpg_output_path, format="JPEG", quality=quality)
        print(f"Compressed JPEG image saved to {compressed_jpg_output_path}")
    except Exception as e:
        print(f"Error: {str(e)}")
```

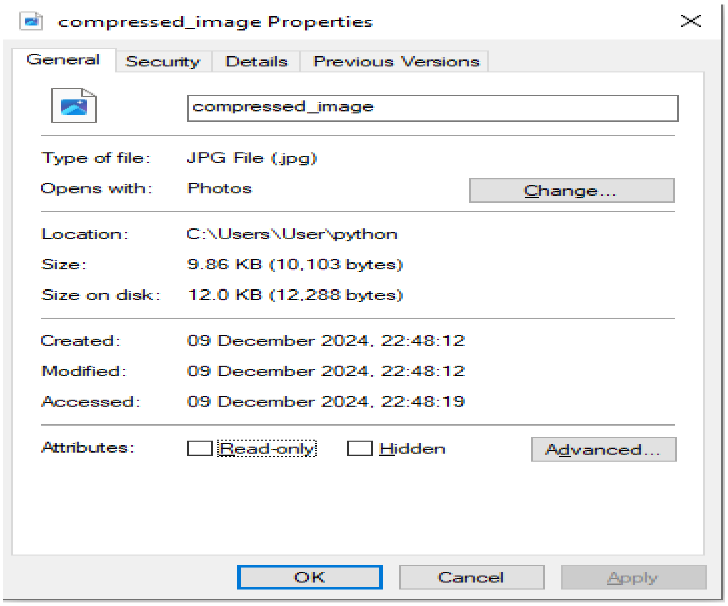
INPUT AND OUTPUT:

ENCODING:

INPUT IMAGE: img.bmp



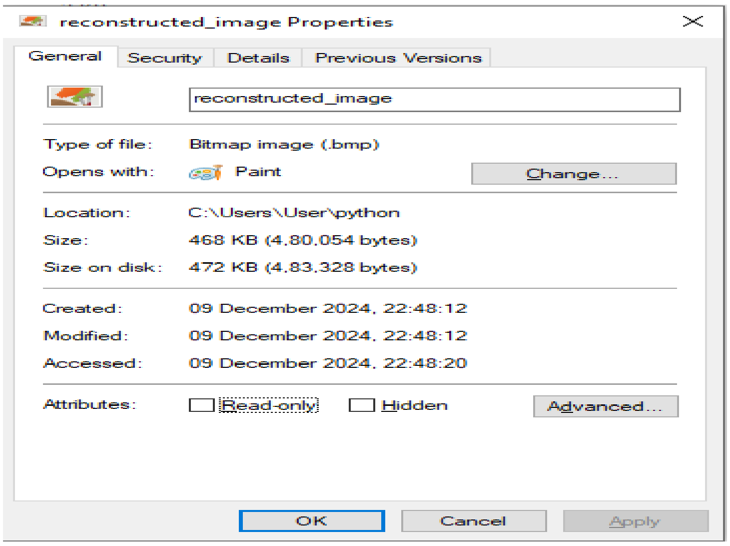
OUTPUT IMAGE: compressed_image.jpeg



DECODING:

INPUT IMAGE: compressed_image.jpeg

OUTPUT IMAGE: reconstructed_image.bmp



RESULT:

The code compresses an input image into a significantly smaller JSON file by applying JPEG techniques like DCT, quantization, and Huffman encoding. The decompression reconstructs the image, saving it as BMP and JPEG files with minor quality loss. This demonstrates effective size reduction while maintaining visual fidelity.