HARSH PATEL DS SEM 7

1. Implement functions for encoding and decoding an image using the following methods: A. Transform Coding (using DCT for forward transform) B. Huffman Encoding C. LZW Encoding D. Run-Length Encoding E. Arithmetic Coding For each method, display the Compression Ratio and calculate the Root Mean Square Error (RMSE) between the original and reconstructed image to quantify any loss of information.

1. Transform Coding (DCT)

In [5]:
```
import numpy as np
from scipy.fftpack import dct, idct
from skimage import io

def dct_encode(image):
    # Apply DCT
    dct_image = dct(dct(image.T, norm='ortho').T, norm='ortho')
    return dct_image

def dct_decode(dct_image):
    # Apply inverse DCT
    reconstructed_image = idct(idct(dct_image.T, norm='ortho').T, norm='ortho')
    return np.clip(reconstructed_image, 0, 255)

def calculate_rmse(original, reconstructed):
    return np.sqrt(np.mean((original - reconstructed) ** 2))

def calculate_compression_ratio(original_size, compressed_size):
    return original_size / compressed_size

# Example usage
image = io.imread('giraffe.jpg', as_gray=True)
dct_image = dct_encode(image)
reconstructed_image = dct_decode(dct_image)

original_size = image.size
compressed_size = dct_image.size  # Simplified for demonstration
cr = calculate_compression_ratio(original_size, compressed_size)
rmse = calculate_rmse(image, reconstructed_image)
print(f'DCT Compression Ratio: {cr}, RMSE: {rmse}')
```

```
DCT Compression Ratio: 1.0, RMSE: 1.8099436130586364e-16
```

2. Huffman Encoding

In [8]:
```
import heapq
from collections import defaultdict

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 huffman_encoding(data):
    frequency = defaultdict(int)
    for char in data:
        frequency[char] += 1

    heap = [Node(char, freq) for char, freq in frequency.items()]
    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)

    root = heap[0]
    huffman_codes = {}

    def generate_codes(node, current_code=""):
        if node is not None:
            if node.char is not None:
                huffman_codes[node.char] = current_code
            generate_codes(node.left, current_code + "0")
            generate_codes(node.right, current_code + "1")

    generate_codes(root)

    encoded_data = ''.join(huffman_codes[char] for char in data)
    return encoded_data, huffman_codes

# Example usage with binary data from an image.
data = 'example data string'
encoded_data, codes = huffman_encoding(data)
print(f'Huffman Encoded Data: {encoded_data}, Codes: {codes}')
```

```
Huffman Encoded Data: 10000001101011111011111010001010101100011100100001001111001
111101110110, Codes: {'x': '0000', 's': '0001', 't': '001', ' ': '010', 'g': '011
0', 'n': '0111', 'e': '100', 'd': '1010', 'm': '1011', 'a': '110', 'r': '11100',
'p': '11101', 'l': '11110', 'i': '11111'}
```

3. LZW Encoding

In [11]:

```
def lzw_encode(input_string):
    max_table_size = 2 ** 12  # Limit dictionary size to 4096 entries.
    dictionary = {chr(i): i for i in range(256)}
    p = ""
    code = 256
    output_codes = []

    for c in input_string:
        pc = p + c
        if pc in dictionary:
```

```
            p = pc
        else:
            output_codes.append(dictionary[p])
            if code < max_table_size:
                dictionary[pc] = code
                code += 1
            p = c

    if p:
        output_codes.append(dictionary[p])

    return output_codes

# Example usage with binary data from an image.
data_string = 'example data string'
lzw_encoded_output = lzw_encode(data_string)
print(f'LZW Encoded Output: {lzw_encoded_output}')
```

```
LZW Encoded Output: [101, 120, 97, 109, 112, 108, 101, 32, 100, 97, 116, 97, 32,
115, 116, 114, 105, 110, 103]
```

4. Run-Length Encoding (RLE)

In [15]:
```
def rle_encode(data):
    encoding = []
    prev_char = ''
    count = 1

    for char in data:
        if char != prev_char:
            if prev_char:
                encoding.append((prev_char, count))
            count = 1
            prev_char = char
        else:
            count += 1

    encoding.append((prev_char, count))

    return encoding

# Example usage with binary data from an image.
data_string_rle = 'example data string'
rle_encoded_output = rle_encode(data_string_rle)
print(f'RLE Encoded Output: {rle_encoded_output}')
```

```
RLE Encoded Output: [('e', 1), ('x', 1), ('a', 1), ('m', 1), ('p', 1), ('l', 1),
('e', 1), (' ', 1), ('d', 1), ('a', 1), ('t', 1), ('a', 1), (' ', 1), ('s', 1),
('t', 1), ('r', 1), ('i', 1), ('n', 1), ('g', 1)]
```

5. Arithmetic Coding

In [19]:
```
def arithmetic_encoding(data):
    frequency = defaultdict(int)

    for char in data:
        frequency[char] += 1

    total_chars = sum(frequency.values())
```

```
    cumulative_frequency = {}
    cumulative_sum = 0

    for char in sorted(frequency.keys()):
        cumulative_frequency[char] = cumulative_sum / total_chars
        cumulative_sum += frequency[char]

    low, high = 0.0, 1.0

    for char in data:
        range_width = high - low
        high = low + range_width * (cumulative_frequency[char] + frequency[char]
        low += range_width * cumulative_frequency[char]

    return (low + high) / 2

# Example usage with binary data from an image.
data_string_arithmetic = 'example data string'
arithmetic_encoded_value = arithmetic_encoding(data_string_arithmetic)
print(f'Arithmetic Encoded Value: {arithmetic_encoded_value}')
```

```
Arithmetic Encoded Value: 0.4166348997284715
```