Name : Jenil R. Poria
Sem 7 AIML
Roll 18
Gujarat University

In [1]:
```
import numpy as np
import cv2
from scipy.fftpack import dct, idct
import heapq
import collections
from sklearn.metrics import mean_squared_error
import math
```

```
C:\Users\Admin\anaconda3\lib\site-packages\scipy\__init__.py:146: UserWarning: A NumPy version >=1.16.5 and <1.23.0
is required for this version of SciPy (detected version 1.26.4
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

Part B : Coding

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. LZWEncoding
   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.

In [3]:
```
# Helper function for RMSE calculation
def calculate_rmse(original, reconstructed):
    return np.sqrt(mean_squared_error(original.flatten(), reconstructed.flatten()))

# Helper function for compression ratio
def calculate_compression_ratio(original_size, compressed_size):
    return original_size / compressed_size

```

1. Transform Coding (DCT)

-> Transform Coding uses the Discrete Cosine Transform (DCT) to convert the image from the spatial to the frequency domain.
-> In this domain, we can discard high-frequency components that have minimal impact on perceived image quality.
-> Compression Ratio is calculated based on the number of coefficients retained.
-> RMSE is calculated between the original and the reconstructed (inverse-transformed) image.

In [2]:
```
# DCT encoding
def dct_encode(image, block_size=8):
    h, w = image.shape
    dct_blocks = np.zeros((h, w), dtype=np.float32)
    for i in range(0, h, block_size):
        for j in range(0, w, block_size):
            block = image[i:i+block_size, j:j+block_size]
            dct_blocks[i:i+block_size, j:j+block_size] = dct(dct(block, axis=0, norm='ortho'), axis=1, norm='orth
    return dct_blocks

# DCT decoding
def dct_decode(dct_blocks, block_size=8):
    h, w = dct_blocks.shape
    reconstructed = np.zeros((h, w), dtype=np.float32)
    for i in range(0, h, block_size):
        for j in range(0, w, block_size):
            block = dct_blocks[i:i+block_size, j:j+block_size]
            reconstructed[i:i+block_size, j:j+block_size] = idct(idct(block, axis=0, norm='ortho'), axis=1, norm=
    return np.clip(reconstructed, 0, 255).astype(np.uint8)

```

In [4]:

```
# Example 8x8 grayscale image (pixel values between 0-255)
original_image = np.array([
    [52, 55, 61, 66, 70, 61, 64, 73],
    [63, 59, 66, 90, 109, 85, 69, 72],
    [62, 59, 68, 113, 144, 104, 66, 73],
    [63, 58, 71, 122, 154, 106, 70, 69],
    [67, 61, 68, 104, 126, 88, 68, 70],
    [79, 65, 60, 70, 77, 68, 58, 75],
    [85, 71, 64, 59, 55, 61, 65, 83],
    [87, 79, 69, 68, 65, 76, 78, 94]
], dtype=np.uint8)

# Perform DCT encoding
dct_encoded = dct_encode(original_image)

# Perform DCT decoding to reconstruct the image
dct_decoded = dct_decode(dct_encoded)

# Calculate Compression Ratio and RMSE
original_size = original_image.size * original_image.itemsize
compressed_size = dct_encoded.size * dct_encoded.itemsize  # Can adjust based on quantization for real compressio
compression_ratio = calculate_compression_ratio(original_size, compressed_size)
rmse = calculate_rmse(original_image, dct_decoded)

print("Original Image:\n", original_image)
print("DCT Encoded (Frequency Domain):\n", dct_encoded)
print("Reconstructed Image:\n", dct_decoded)
print(f"Compression Ratio: {compression_ratio:.2f}")
print(f"RMSE: {rmse:.2f}")
```

```
Original Image:
 [[ 52  55  61  66  70  61  64  73]
 [ 63  59  66  90 109  85  69  72]
 [ 62  59  68 113 144 104  66  73]
 [ 63  58  71 122 154 106  70  69]
 [ 67  61  68 104 126  88  68  70]
 [ 79  65  60  70  77  68  58  75]
 [ 85  71  64  59  55  61  65  83]
 [ 87  79  69  68  65  76  78  94]]
DCT Encoded (Frequency Domain):
 [[ 6.1000000e+02 -2.9105387e+01 -6.1941208e+01  2.5332142e+01
   5.4750000e+01 -1.9715813e+01 -5.9112304e-01  2.0786445e+00]
 [ 6.0823526e+00 -2.0587105e+01 -6.1633060e+01  8.0110302e+00
   1.1528281e+01 -6.6413360e+00 -6.4229479e+00  6.7780781e+00]
 [-4.6090340e+01  7.9552679e+00  7.6726662e+01 -2.5594141e+01
  -2.9655832e+01  1.0138830e+01  6.3890872e+00 -4.7739291e+00]
 [-4.8914330e+01  1.1770298e+01  3.4305077e+01 -1.4233221e+01
  -9.8612452e+00  6.1913018e+00  1.3355051e+00  1.4998544e+00]
 [ 1.0750000e+01 -7.6337805e+00 -1.2451977e+01 -2.0442479e+00
  -5.0000000e-01  1.3659228e+00 -4.5837522e+00  1.5184534e+00]
 [-9.6419239e+00  1.4069998e+00  3.4119539e+00 -3.2939796e+00
  -4.7061691e-01  4.1520187e-01  1.8118628e+00 -3.9391515e-01]
 [-2.8271980e+00 -1.2284524e+00  1.3890873e+00  7.6289102e-02
   9.1873014e-01 -3.5149665e+00  1.7733406e+00 -2.7744372e+00]
 [-1.2457062e+00 -7.0720315e-01 -4.8686570e-01 -2.6944506e+00
  -8.9983523e-02 -3.9582360e-01 -9.1025054e-01  4.0512446e-01]]
Reconstructed Image:
 [[ 52  55  61  66  70  61  64  73]
 [ 63  59  66  90 109  85  69  72]
 [ 62  59  68 113 144 104  66  73]
 [ 63  58  71 122 154 106  70  69]
 [ 67  61  68 104 126  88  68  70]
 [ 79  65  60  70  77  68  58  75]
 [ 85  71  64  59  55  61  65  83]
 [ 87  79  69  68  65  76  78  94]]
Compression Ratio: 0.25
RMSE: 0.00
```

Huffman Coding

-> Huffman Encoding is a lossless compression method that uses variable-length codes for each pixel based on its frequency.
-> More frequent values get shorter codes, and less frequent values get longer codes.
-> Compression Ratio is based on the average length of codes versus the original 8-bit encoding.
-> RMSE is not applicable here as Huffman encoding is lossless.

In [6]:
```python
# Huffman encoding and decoding functions
class HuffmanNode:
    def __init__(self, symbol, frequency):
        self.symbol = symbol
        self.frequency = frequency
        self.left = None
        self.right = None

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

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

def build_huffman_codes(node, prefix="", code_dict={}):
    if node.symbol is not None:
        code_dict[node.symbol] = prefix
    else:
        build_huffman_codes(node.left, prefix + "0", code_dict)
        build_huffman_codes(node.right, prefix + "1", code_dict)
    return code_dict

def huffman_encode(image):
    frequency_dict = collections.Counter(image.flatten())
    huffman_tree = build_huffman_tree(frequency_dict)
    huffman_codes = build_huffman_codes(huffman_tree)
    encoded_image = ''.join(huffman_codes[pixel] for pixel in image.flatten())
    return encoded_image, huffman_codes

# Function to decode Huffman encoded image
def huffman_decode(encoded_image, huffman_codes, shape):
    reverse_codes = {v: k for k, v in huffman_codes.items()}
    current_code = ""
    decoded_pixels = []

    for bit in encoded_image:
        current_code += bit
        if current_code in reverse_codes:
            decoded_pixels.append(reverse_codes[current_code])
            current_code = ""

    return np.array(decoded_pixels).reshape(shape)

```

In [7]:

```python
# Example 4x4 image for simplicity
example_image = np.array([
    [45, 45, 255, 255],
    [45, 45, 255, 255],
    [45, 200, 200, 255],
    [45, 200, 200, 255]
], dtype=np.uint8)

# Huffman Encode the image
encoded_image, huffman_codes = huffman_encode(example_image)

# Decode the encoded image
decoded_image = huffman_decode(encoded_image, huffman_codes, example_image.shape)

# Calculate Compression Ratio
original_size = example_image.size * 8  # 8 bits per pixel in original image
compressed_size = len(encoded_image)
compression_ratio = original_size / compressed_size

# Display results
print("Original Image:\n", example_image)
print("Huffman Codes:", huffman_codes)
print("Encoded Image:", encoded_image)
print("Decoded Image:\n", decoded_image)
print(f"Compression Ratio: {compression_ratio:.2f}")
```

```
Original Image:
 [[ 45  45 255 255]
 [ 45  45 255 255]
 [ 45 200 200 255]
 [ 45 200 200 255]]
Huffman Codes: {45: '0', 200: '10', 255: '11'}
Encoded Image: 00111100111101010110101011
Decoded Image:
 [[ 45  45 255 255]
 [ 45  45 255 255]
 [ 45 200 200 255]
 [ 45 200 200 255]]
Compression Ratio: 4.92
```

LZW

-> LZW Encoding is another lossless compression algorithm that builds a dictionary of pixel sequences.
-> Repeated sequences are replaced by dictionary references.
-> Compression Ratio is based on the size of the dictionary and the encoded data versus the original data size.
-> RMSE is not applicable for lossless LZW encoding.

In [8]:

```python
# LZW encoding function
def lzw_encode(image):
    image = image.flatten()
    dictionary = {bytes([i]): i for i in range(256)}
    current_sequence = bytes([image[0]])
    encoded_data = []
    code = 256
    for pixel in image[1:]:
        sequence_plus_pixel = current_sequence + bytes([pixel])
        if sequence_plus_pixel in dictionary:
            current_sequence = sequence_plus_pixel
        else:
            encoded_data.append(dictionary[current_sequence])
            dictionary[sequence_plus_pixel] = code
            code += 1
            current_sequence = bytes([pixel])
    encoded_data.append(dictionary[current_sequence])
    return encoded_data

# Function to perform LZW Decoding
def lzw_decode(encoded_data):
    # Initialize the dictionary for decoding
    dictionary = {i: bytes([i]) for i in range(256)}
    code = 256  # Start codes for sequences longer than one byte

    # Decode the first value
    current_sequence = dictionary[encoded_data[0]]
    decoded_image = [current_sequence]

    for code_value in encoded_data[1:]:
        if code_value in dictionary:
            entry = dictionary[code_value]
        elif code_value == code:
            entry = current_sequence + current_sequence[:1]

        # Append decoded sequence
        decoded_image.append(entry)

        # Add new sequence to the dictionary
        dictionary[code] = current_sequence + entry[:1]
        code += 1
        current_sequence = entry

    # Convert to a 1D array of pixel values
    decoded_image = b''.join(decoded_image)
    return np.frombuffer(decoded_image, dtype=np.uint8)

```

In [9]:

```python

# Example 4x4 grayscale image for simplicity
example_image = np.array([
    [45, 45, 45, 255],
    [45, 45, 255, 255],
    [200, 200, 45, 45],
    [200, 200, 255, 255]
], dtype=np.uint8)

# Step 1: LZW Encode the image
encoded_data = lzw_encode(example_image)

# Step 2: LZW Decode the encoded image
decoded_image = lzw_decode(encoded_data).reshape(example_image.shape)

# Calculate Compression Ratio
original_size = example_image.size * 8  # 8 bits per pixel in the original image
compressed_size = len(encoded_data) * 16  # Assuming each encoded entry takes 16 bits
compression_ratio = original_size / compressed_size

# Display results
print("Original Image:\n", example_image)
print("Encoded Data:", encoded_data)
print("Decoded Image:\n", decoded_image)
print(f"Compression Ratio: {compression_ratio:.2f}")

```

```
Original Image:
 [[ 45  45  45 255]
 [ 45  45 255 255]
 [200 200  45  45]
 [200 200 255 255]]
Encoded Data: [45, 256, 255, 257, 255, 200, 200, 256, 261, 255, 255]
Decoded Image:
 [[ 45  45  45 255]
 [ 45  45 255 255]
 [200 200  45  45]
 [200 200 255 255]]
Compression Ratio: 0.73
```

4. Run-Length Encoding (RLE)

-> Run-Length Encoding encodes consecutive identical pixels as a single value and a run length
-> Works well for images with large areas of uniform color
-> Compression Ratio is calculated based on the reduction in the number of pixels stored
-> RMSE is not applicable as RLE is also lossless

In [10]:

```python
import numpy as np

# Function to perform RLE Encoding
def rle_encode(image):
    # Flatten the image to treat it as a 1D sequence
    image = image.flatten()
    encoded_data = []
    i = 0

    # Traverse through the image pixels
    while i < len(image):
        count = 1
        while i + 1 < len(image) and image[i] == image[i + 1]:
            count += 1
            i += 1
        # Store the pixel value and its count
        encoded_data.append((image[i], count))
        i += 1
    return encoded_data

# Function to perform RLE Decoding
def rle_decode(encoded_data, shape):
    decoded_image = []

    # Expand each (value, count) pair
    for value, count in encoded_data:
        decoded_image.extend([value] * count)

    # Convert list to a numpy array and reshape to original image shape
    return np.array(decoded_image, dtype=np.uint8).reshape(shape)

# Example 4x4 grayscale image for simplicity
example_image = np.array([
    [45, 45, 45, 255],
    [45, 45, 255, 255],
    [200, 200, 45, 45],
    [200, 200, 255, 255]
], dtype=np.uint8)

# Step 1: RLE Encode the image
encoded_data = rle_encode(example_image)

# Step 2: RLE Decode the encoded image
decoded_image = rle_decode(encoded_data, example_image.shape)

# Calculate Compression Ratio
original_size = example_image.size * 8  # 8 bits per pixel in the original image
compressed_size = sum(len(bin(value)[2:]) + 8 for value, count in encoded_data)  # compressed size
compression_ratio = original_size / compressed_size

# Display results
print("Original Image:\n", example_image)
print("RLE Encoded Data:", encoded_data)
print("Decoded Image:\n", decoded_image)
print(f"Compression Ratio: {compression_ratio:.2f}")

```

```
Original Image:
 [[ 45  45  45 255]
 [ 45  45 255 255]
 [200 200  45  45]
 [200 200 255 255]]
RLE Encoded Data: [(45, 3), (255, 1), (45, 2), (255, 2), (200, 2), (45, 2), (200, 2), (255, 2)]
Decoded Image:
 [[ 45  45  45 255]
 [ 45  45 255 255]
 [200 200  45  45]
 [200 200 255 255]]
Compression Ratio: 1.05
```

5. Arithmetic Coding

-> Arithmetic Coding assigns probabilities to pixel sequences and encodes them as a single fraction.
-> This method is lossless but computationally more intensive than Huffman coding.
-> Compression Ratio is based on the final encoded length compared to the original.
-> RMSE is not applicable as it is lossless.

In [13]:

```python
from collections import Counter
import numpy as np

# Function to calculate probability ranges for each pixel value
def calculate_prob_ranges(sequence):
    total_pixels = len(sequence)
    freq = Counter(sequence)
    prob_ranges = {}
    current_low = 0.0

    # Calculate cumulative probability ranges for each pixel value
    for pixel_value, count in sorted(freq.items()):
        probability = count / total_pixels
        current_high = current_low + probability
        prob_ranges[pixel_value] = (current_low, current_high)
        current_low = current_high

    return prob_ranges

# Arithmetic encoding function
def arithmetic_encode(sequence, prob_ranges):
    low, high = 0.0, 1.0

    for pixel in sequence:
        pixel_low, pixel_high = prob_ranges[pixel]
        range_ = high - low
        high = low + range_ * pixel_high
        low = low + range_ * pixel_low

    return (low + high) / 2  # Encoded as a single value within the final range

# Arithmetic decoding function
def arithmetic_decode(encoded_value, prob_ranges, sequence_length):
    low, high = 0.0, 1.0
    decoded_sequence = []

    for _ in range(sequence_length):
        range_ = high - low
        for pixel, (pixel_low, pixel_high) in prob_ranges.items():
            pixel_range_low = low + range_ * pixel_low
            pixel_range_high = low + range_ * pixel_high
            if pixel_range_low <= encoded_value < pixel_range_high:
                decoded_sequence.append(pixel)
                low, high = pixel_range_low, pixel_range_high
                break

    return decoded_sequence

# Example image represented as a 4x4 grayscale image
example_image = np.array([
    [45, 45, 45, 255],
    [45, 45, 255, 255],
    [200, 200, 45, 45],
    [200, 200, 255, 255]
], dtype=np.uint8)

# Flatten the image to create a sequence
sequence = example_image.flatten()

# Step 1: Calculate probability ranges for each pixel value
prob_ranges = calculate_prob_ranges(sequence)

# Step 2: Encode the sequence using Arithmetic Encoding
encoded_value = arithmetic_encode(sequence, prob_ranges)

# Step 3: Decode the sequence to retrieve the original image
decoded_sequence = arithmetic_decode(encoded_value, prob_ranges, len(sequence))
decoded_image = np.array(decoded_sequence, dtype=np.uint8).reshape(example_image.shape)

# Calculate Compression Ratio
original_size = example_image.size * 8  # 8 bits per pixel in the original image
compressed_size = len(bin(int(encoded_value * (2 ** 32)))) - 2  # Approx. bits for encoding
compression_ratio = original_size / compressed_size

# Display results
print("Original Image:\n", example_image)
print("Encoded Value:", encoded_value)
print("Decoded Image:\n", decoded_image)
print(f"Compression Ratio: {compression_ratio:.2f}")

```

```
Original Image:
 [[ 45  45  45 255]
 [ 45  45 255 255]
 [200 200  45  45]
 [200 200 255 255]]
Encoded Value: 0.06236219020966758
Decoded Image:
 [[ 45  45  45 255]
 [ 45  45 255 255]
 [200 200  45  45]
 [200 200 255 255]]
Compression Ratio: 4.57
```

In [ ]:

```
1
```