```
# Part B : Coding

Name: Hetvi Vaghela


Semester-7


Roll No.: 26


Course : Msc AIML
```

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 [1]:

```
import numpy as np
import cv2
from scipy.fftpack import dct, idct
from skimage.metrics import mean_squared_error
import heapq
import collections
import itertools
import math

```

In [2]:

```
# RMSE Calculate

def calculate_rmse(original, reconstructed):
    return np.sqrt(mean_squared_error(original, reconstructed))

```

A. Transform Coding (Using DCT)

In [3]:

```python
#Forward DCT Transform and Quantization:
def dct_encode(image, block_size=8):
    h, w = image.shape
    dct_transformed = np.zeros_like(image, dtype=float)
    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_transformed[i:i+block_size, j:j+block_size] = dct(dct(b
    return dct_transformed

```

In [4]:

```python
#Inverse DCT Transform:
def dct_decode(dct_transformed, block_size=8):
    h, w = dct_transformed.shape
    reconstructed = np.zeros_like(dct_transformed)
    for i in range(0, h, block_size):
        for j in range(0, w, block_size):
            block = dct_transformed[i:i+block_size, j:j+block_size]
            reconstructed[i:i+block_size, j:j+block_size] = idct(idct(b
    return np.clip(reconstructed, 0, 255).astype(np.uint8)

```

B. Huffman Encoding

In [5]:

```python
# 1. Encoding:

def huffman_encode(image):
    frequency = collections.Counter(image.flatten())
    heap = [[weight, [symbol, ""]] for symbol, weight in frequency.item
    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:])
    huff_dict = sorted(heapq.heappop(heap)[1:], key=lambda p: (len(p[-1
    return huff_dict  # return the huffman table for decoding

```

In [6]:

```python
# 2. Decoding

def huffman_decode(encoded_image, huff_dict):
    decoded_image = []
    inverse_dict = {code: symbol for symbol, code in huff_dict}
    code = ""
    for bit in encoded_image:
        code += bit
        if code in inverse_dict:
            decoded_image.append(inverse_dict[code])
            code = ""
    return np.array(decoded_image).reshape(image.shape)

```

C. LZW Encoding

In [8]:

```python
# 1. Encoding
def lzw_encode(image):
    dictionary = {bytes([i]): i for i in range(256)}
    p = bytes()
    code = []
    for c in image.flatten():
        pc = p + bytes([c])
        if pc in dictionary:
            p = pc
        else:
            code.append(dictionary[p])
            dictionary[pc] = len(dictionary)
            p = bytes([c])
    if p:
        code.append(dictionary[p])
    return code

```

In [9]:

```python
# 2. Decoding


def lzw_decode(code):
    dictionary = {i: bytes([i]) for i in range(256)}
    p = bytes([code.pop(0)])
    decoded_image = [p]
    for k in code:
        entry = dictionary[k] if k in dictionary else p + p[:1]
        decoded_image.append(entry)
        dictionary[len(dictionary)] = p + entry[:1]
        p = entry
    return np.array(b''.join(decoded_image)).reshape(image.shape)

```

D. Run-Length Encoding

In [10]:

```
# 1. Encoding:
def run_length_encode(image):
    flattened = image.flatten()
    encoded = []
    count = 1
    for i in range(1, len(flattened)):
        if flattened[i] == flattened[i-1]:
            count += 1
        else:
            encoded.append((flattened[i-1], count))
            count = 1
    encoded.append((flattened[-1], count))  # Add last element
    return encoded

```

In [12]:

```
# 2.Decoding:
def run_length_decode(encoded, shape):
    decoded = []
    for value, count in encoded:
        decoded.extend([value] * count)
    return np.array(decoded).reshape(shape)


```

E. Arithmetic Coding

Encoding and Decoding: For arithmetic coding, you can use an external library like python-arithmetic-coding for efficient implementation since arithmetic coding can be complex to implement from scratch.

In [14]:

```
def evaluate_compression(original, compressed):
    original_size = original.size * original.itemsize
    compressed_size = len(compressed) if isinstance(compressed, list) e
    compression_ratio = original_size / compressed_size
    rmse = calculate_rmse(original, compressed)
    print(f"Compression Ratio: {compression_ratio:.2f}")
    print(f"RMSE: {rmse:.2f}")
    return compression_ratio, rmse

```

In [15]:

```
# Example
# Load a grayscale image
image = cv2.imread('R.jpeg', cv2.IMREAD_GRAYSCALE)

# DCT example
dct_transformed = dct_encode(image)
reconstructed_dct = dct_decode(dct_transformed)
evaluate_compression(image, reconstructed_dct)

```

```
Compression Ratio: 1.00
RMSE: 0.07
```

Out[15]:

```
(1.0, 0.0657586872359574)
```

In [ ]:
```
1
```