NAME : SAHIL YADAV

Roll No : 21ECE1039

ITC Assignment

Q1:Write a program that performs Huffman coding, given the source probabilities. It should generate the code and give the coding efficiency.

```
import heapq

# Node class to represent a node in the Huffman Tree
class Node:
    def __init__(self, freq, symbol, left=None, right=None):
        self.freq = freq  # frequency of the symbol
        self.symbol = symbol  # symbol (character)
        self.left = left  # left child
        self.right = right  # right child
        self.huff = ''  # Huffman code for the symbol

    # To make Node class comparable based on frequency (for heapq)
    def __lt__(self, nxt):
        return self.freq < nxt.freq

# Utility function to calculate and print the Huffman Codes
def generate_huffman_codes(node, val=''):
    codes = {}
    new_val = val + str(node.huff)

    if node.left:  # Traverse left
        codes.update(generate_huffman_codes(node.left, new_val))
    if node.right:  # Traverse right
        codes.update(generate_huffman_codes(node.right, new_val))

    if not node.left and not node.right:  # If it's a leaf node, store its code
        codes[node.symbol] = new_val

    return codes

# Function to calculate coding efficiency
def calculate_efficiency(codes, freq):
    total_bits = 0  # Sum of the bits used in the Huffman code
    original_bits = 0  # Sum of bits in the original uncompressed message

    # Calculate total bits used in the Huffman encoding
    for symbol, code in codes.items():
        total_bits += len(code) * freq[symbol]

    # Calculate original number of bits (if fixed-length encoding was used)
    original_bits = sum(freq.values()) * len(bin(len(freq)-1)[2:])  # Using fixed-length encoding

    # Calculate coding efficiency
    efficiency = original_bits / total_bits
    return efficiency

# Driver code to build the Huffman tree and print the results
if __name__ == "__main__":
```

```
if __name__ == "__main__":
    chars = ['a', 'b', 'c', 'd', 'e', 'f']  # Characters to be encoded
    freq = [5, 9, 12, 13, 16, 45]  # Frequencies of the characters or probability

    # Create a priority queue (min-heap)
    nodes = []

    # Build the priority queue from the characters and their frequencies
    for x in range(len(chars)):
        heapq.heappush(nodes, Node(freq[x], chars[x]))

    # Build the Huffman Tree
    while len(nodes) > 1:
        left = heapq.heappop(nodes)
        right = heapq.heappop(nodes)

        left.huff = 0  # Assign 0 to the left child
        right.huff = 1  # Assign 1 to the right child

        # Create a new internal node with combined frequency
        new_node = Node(left.freq + right.freq, left.symbol + right.symbol, left, right)
        heapq.heappush(nodes, new_node)

    # The root of the Huffman Tree is nodes[0]
    root = nodes[0]

    # Generate Huffman codes from the tree
    huffman_codes = generate_huffman_codes(root)

    # Print the generated Huffman codes
    print("Huffman Codes:")
    for symbol, code in huffman_codes.items():
        print(f"{symbol} -> {code}")

    # Calculate the coding efficiency
    symbol_freq_dict = {chars[i]: freq[i] for i in range(len(chars))}
    efficiency = calculate_efficiency(huffman_codes, symbol_freq_dict)
    print(f"\nCoding Efficiency: {efficiency:.4f}")
```

```
Huffman Codes:
f -> 0
c -> 100
d -> 101
a -> 1100
b -> 1101
e -> 111

Coding Efficiency: 1.3393
```

Q2 :Write a program that performs Run Length encoding on a sequence of bits and gives the coded output along with the compression ratio

```
# Run Length Encoding for bit sequence
def printRLE(st):
    n = len(st)
    i = 0
    encoded_output = ""
    original_size = n  # Original size of the bit sequence
```

```
    while i < n:
        # Count occurrences of current bit
        count = 1
        while i < n - 1 and st[i] == st[i + 1]:
            count += 1
            i += 1
        i += 1

        # Append bit and its count to encoded output
        encoded_output += st[i - 1] + str(count)

    compressed_size = len(encoded_output)  # Compressed size after encoding
    compression_ratio = original_size / compressed_size  # Calculate compression ratio

    # Print encoded bit sequence and the compression ratio
    print("Encoded Output:", encoded_output)
    print("Compression Ratio:", compression_ratio)

# Driver code
if __name__ == "__main__":
    st = "111000111100000011"
    printRLE(st)
```

```
Encoded Output: 1303140612
Compression Ratio: 1.8
```

Q3:Write a program that takes in a 2^n level gray scale image (n bits per pixel) and perform the following operation: a. Break it up into 8 by 8 blocks b. Perform DCT on each of the 8 by 8 blocks c. Quantizes the DCT coefficients by retaining only the m MSB where m<n d. Perform the zig-zag coding followed by run length encoding e. Perform Huffman coding on the bit stream obtained above (think a reasonable way of calculating the symbol probabilities) f. Calculate the compression ratio. g. Perform the decompression i.e the inverse operation.

Algorithm

1. Input image: Image of size Height x Width with n bits per pixel
2. Break image into 8x8 blocks:
    - For each block:
        - Apply DCT to the block
        - Quantize DCT coefficients by retaining only m MSB
        - Perform Zig-Zag scan to rearrange coefficients in a 1D array
3. Apply Run-Length Encoding (RLE) to the zig-zagged coefficients:
    - For each block's zig-zagged coefficients:
        - Encode runs of identical values
4. Perform Huffman Coding on the RLE bitstream:
    - Count frequency of each RLE symbol
    - Build Huffman Tree

    - Generate Huffman codes for each symbol

5. Calculate Compression Ratio:
    - Original size = Height * Width * n bits
    - Compressed size = Sum of the sizes of Huffman codes for all symbols
    - Compression ratio = Original size / Compressed size
6. Decompression:
    - Input: Compressed bitstream (Huffman codes, RLE symbols, etc.)
    - Decode the Huffman bitstream to obtain RLE symbols
    - Decode RLE to reconstruct the zig-zagged coefficients
    - Reverse Zig-Zag scan to convert 1D array back to 8x8 blocks
    - Reverse quantization to restore original DCT coefficients
    - Apply Inverse DCT to each block
    - Reconstruct the image
7. Output: Decompressed image

Import libararies

```
import numpy as np
import heapq
import itertools
from collections import Counter
from scipy.fftpack import dct, idct
from PIL import Image

# Step 1: Apply DCT to 8x8 blocks in the image
def apply_dct(image, block_size=8):
    dct_blocks = []
    for i in range(0, image.shape[0], block_size):
        for j in range(0, image.shape[1], block_size):
            block = image[i:i+block_size, j:j+block_size]
            if block.shape == (block_size, block_size):  # Ensure block is 8x8
                dct_block = dct(dct(block.T, norm='ortho').T, norm='ortho')
                dct_blocks.append(dct_block)
    return dct_blocks


# Step 2: Quantize the DCT coefficients by retaining only the m MSB
def quantize_dct_coefficients(dct_blocks, m, n):
    quantized_blocks = []
    for block in dct_blocks:
        quantized_block = np.round(block)
        quantized_block = np.floor(quantized_block / (2 ** (n - m)))  # Quantization
        quantized_blocks.append(quantized_block)
    return quantized_blocks


# Step 3: Zig-Zag scan for each 8x8 block
def zigzag_scan(block):
    zigzag_order = [(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), (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), (2, 7)]

    zigzag_values = [block[i, j] for i, j in zigzag_order]
    return zigzag_values


# Step 4: Run-Length Encoding (RLE)
def run_length_encoding(zigzag_values):
    rle = []
    for key, group in itertools.groupby(zigzag_values):
        rle.append((key, len(list(group))))  # Count the occurrences of each symbol
    return rle


# Step 5: Huffman Coding
class Node:
    def __init__(self, freq, symbol, left=None, right=None):
        self.freq = freq
        self.symbol = symbol
        self.left = left
        self.right = right
        self.huff = ''

    def __lt__(self, nxt):
        return self.freq < nxt.freq

def generate_huffman_tree(symbols_freq):
    heap = [Node(freq, symbol) for symbol, freq in symbols_freq]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        new_node = Node(left.freq + right.freq, left.symbol + right.symbol, left, right)
        heapq.heappush(heap, new_node)

    return heap[0]

def generate_huffman_codes(node, val=''):
    codes = {}
    if node.left:
        codes.update(generate_huffman_codes(node.left, val + '0'))
    if node.right:
        codes.update(generate_huffman_codes(node.right, val + '1'))
    if not node.left and not node.right:
        codes[node.symbol] = val
    return codes

# Step 6: Calculate Compression Ratio
def calculate_compression_ratio(original_size, compressed_size):
    return original_size / compressed_size


# Step 7: Inverse DCT (IDCT) to reconstruct the image
def inverse_dct(blocks, block_size=8):
    img_reconstructed = []
    for block in blocks:
        idct_block = idct(idct(block.T, norm='ortho').T, norm='ortho')
```

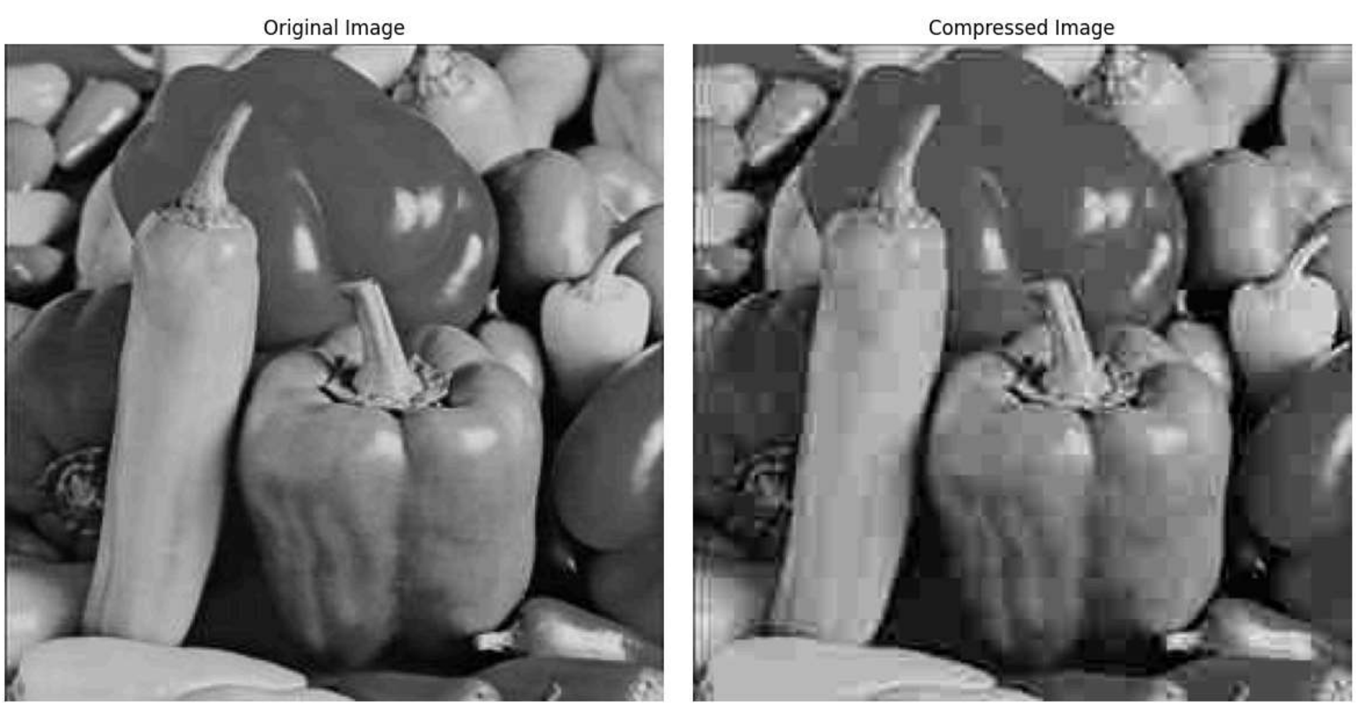
```
        idct_block = idct(idct(block.T, norm='ortho').T, norm='ortho')
        img_reconstructed.append(idct_block)

    img_reconstructed = np.array(img_reconstructed)
    img_reconstructed = np.clip(img_reconstructed, 0, 255).astype(np.uint8)
    return img_reconstructed


# Step 8: Decompress (Rebuild the image from RLE and Huffman codes)
def decode_huffman(huffman_codes, encoded_data):
    reverse_codes = {v: k for k, v in huffman_codes.items()}
    current_code = ''
    decoded_data = []
    for bit in encoded_data:
        current_code += bit
        if current_code in reverse_codes:
            decoded_data.append(reverse_codes[current_code])
            current_code = ''
    return decoded_data

def decompress_image(huffman_codes, rle_values, quantized_blocks, image_shape, block_size=8):
    # Reverse Huffman coding (decode the RLE data back into the original symbols)
    decoded_rle = []
    for rle_block in rle_values:
        for key, count in rle_block:
            decoded_rle.extend([key] * count)

    # Reverse Zig-Zag scan
    quantized_blocks_reconstructed = []
    zigzag_order = [(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), (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), (2, 7)]

    for block_data in quantized_blocks:
        block = block_data.flatten()
        zigzag_reversed = np.zeros((block_size, block_size))
        for idx, (i, j) in enumerate(zigzag_order):
            zigzag_reversed[i, j] = block[idx]
        quantized_blocks_reconstructed.append(zigzag_reversed)

    # Perform inverse DCT on each block
    reconstructed_blocks = inverse_dct(quantized_blocks_reconstructed)

    # Reconstruct the full image from blocks
    num_blocks_x = image_shape[1] // block_size
    num_blocks_y = image_shape[0] // block_size
    reconstructed_image = np.block(
        [[reconstructed_blocks[i + j * num_blocks_x] for j in range(num_blocks_y)]
         for i in range(num_blocks_x)]
    )

    expected_shape = (num_blocks_y * block_size, num_blocks_x * block_size)

    # Resize the image to the original size
    if expected_shape != image_shape:
        reconstructed_image = reconstructed_image[:image_shape[0], :image_shape[1]]
    print("Reconstructed image shape:", reconstructed_image.shape)
```

```
        return reconstructed_image


    # Main program
    def compress_image(image_path, m=4, n=8):
        img = Image.open(image_path).convert('L')
        image = np.array(img)

        # Ensure image dimensions are divisible by 8 (pad if necessary)
        if image.shape[0] % 8 != 0 or image.shape[1] % 8 != 0:
            image = np.pad(image, ((0, 8 - (image.shape[0] % 8)),
                                   (0, 8 - (image.shape[1] % 8))),
                           mode='constant', constant_values=255)

        # Perform DCT on 8x8 blocks
        dct_blocks = apply_dct(image)

        # Quantize the DCT coefficients
        quantized_blocks = quantize_dct_coefficients(dct_blocks, m, n)

        # Perform Zig-Zag scan
        zigzag_values = [zigzag_scan(block) for block in quantized_blocks]

        # Perform Run Length Encoding (RLE)
        rle_values = [run_length_encoding(zigzag) for zigzag in zigzag_values]

        # Flatten RLE output for Huffman coding
        all_rle_symbols = [item[0] for sublist in rle_values for item in sublist]
        symbol_counts = Counter(all_rle_symbols)

        # Perform Huffman Coding
        huffman_tree = generate_huffman_tree(symbol_counts.items())
        huffman_codes = generate_huffman_codes(huffman_tree)

        # Calculate compression ratio
        original_size = image.size * 8  # in bits
        compressed_size = sum(len(huffman_codes[symbol]) * count for symbol, count in symbol_counts.items())
        compression_ratio = calculate_compression_ratio(original_size, compressed_size)

        print(f"Compression Ratio: {compression_ratio}")
        return huffman_codes, rle_values, quantized_blocks


    # Example Usage
    if __name__ == "__main__":
        input_image_path = "peppers.png"  # Replace with your image path

        # Compress Image
        huffman_codes, rle_values, quantized_blocks = compress_image(input_image_path)

        # Decompress Image (pass the original image shape)
        original_image = np.array(Image.open(input_image_path).convert('L'))
        decompressed_image = decompress_image(huffman_codes, rle_values, quantized_blocks, original_image.shape)

        # Save and display the decompressed image
        decompressed_image = np.clip(decompressed_image, 0, 255).astype(np.uint8)
        Image.fromarray(decompressed_image).save("decompressed_image.png")
        Image.fromarray(decompressed_image).show()
```

```
Compression Ratio: 9.29984390520789
Reconstructed image shape: (256, 256)
```

Ploting the Original and compressed Images

```
import matplotlib.pyplot as plt
from PIL import Image

# Open the original image and the compressed/reconstructed image
original_image = Image.open('/content/peppers.png')  # Replace with the path to the original image
compressed_image = Image.open('/content/decompressed_image.jpg')  # Path to the compressed image

# Create a subplot to display both images side by side
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

# Display the original image
axes[0].imshow(original_image)
axes[0].set_title('Original Image')
axes[0].axis('off')  # Hide axes for the original image

# Display the compressed/reconstructed image
axes[1].imshow(compressed_image)
axes[1].set_title('Compressed Image')
axes[1].axis('off')  # Hide axes for the compressed image

# Show the figure with both images
plt.tight_layout()
plt.show()
```



Double-click (or enter) to edit