```python
import numpy as np
import heapq
from PIL import Image
import matplotlib.pyplot as plt
from collections import Counter
```

- **numpy**: For array and numerical operations.
- **heapq**: For priority queue operations used in building the Huffman tree.
- **PIL (Pillow)**: For image processing.
- **matplotlib**: For plotting and visualizing images.
- **collections.Counter**: For counting the frequency of pixel values.

#Defining the Node Class

```python
class Node:
   def __init__(self, symbol=None, freq=0, left=None, right=None):
      self.symbol = symbol
      self.freq = freq
      self.left = left
      self.right = right

   def __lt__(self, other):
      return self.freq < other.freq
```

- A class to represent nodes in the Huffman tree.
- Each node can store a symbol (pixel value), its frequency, and pointers to left and right child nodes.
- The __lt__ method is defined to make nodes comparable based on their frequency, which is required for the priority queue operations.

#Calculating Frequencies of Pixels in the image

```python
def calculate_frequencies(image):
   return Counter(image.flatten())
```

#Building the Huffman Tree

```python
def build_huffman_tree(frequencies):
   heap = [Node(symbol=sym, freq=freq) for sym, freq in frequencies.items()]
   heapq.heapify(heap)

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

   return heap[0]
```

- Creates a priority queue (min-heap) with nodes for each pixel value and its frequency.
- Repeatedly pops two nodes with the smallest frequencies, merges them, and pushes the merged node back into the heap.
- Continues until there is only one node left, which becomes the root of the Huffman tree.

#Generating Huffman Codes

```
def generate_huffman_codes(node, prefix='', code_table=None):
    if code_table is None:
        code_table = {}
    if node.symbol is not None:
        code_table[node.symbol] = prefix
    else:
        generate_huffman_codes(node.left, prefix + '0', code_table)
        generate_huffman_codes(node.right, prefix + '1', code_table)
    return code_table
```

#Encoding the pixels to their respective Huffman Codes

```
def encode_image_with_huffman(image, huffman_codes):
    return ''.join(huffman_codes[pixel] for pixel in image.flatten())
```

#Embedding each bit of the bitstream into the LSB of the cover image

```
def embed_bitstream_in_image(cover_image, bitstream):
    watermarked_image = np.array(cover_image).copy()
    index = 0
    for i in range(watermarked_image.shape[0]):
        for j in range(watermarked_image.shape[1]):
            if index < len(bitstream):
                watermarked_image[i, j] = (cover_image[i, j] & 0xFE) | int(bitstream[index])
                index += 1
    return Image.fromarray(watermarked_image)
```

#Reading the LSB of each pixel of the watermarked image

```
def extract_bitstream_from_image(watermarked_image, bitstream_length):
    bitstream = ''
    watermarked_image = np.array(watermarked_image)
    for i in range(watermarked_image.shape[0]):
        for j in range(watermarked_image.shape[1]):
            bitstream += str(watermarked_image[i, j] & 0x01)
            if len(bitstream) == bitstream_length:
                break
    return bitstream
```

#Decoding the bitstream to obtain the original pixel value

```
def decode_huffman_bitstream(bitstream, huffman_tree, num_pixels):
    decoded_pixels = []
    current_node = huffman_tree
    for bit in bitstream:
        current_node = current_node.left if bit == '0' else current_node.right
        if current_node.symbol is not None:
            decoded_pixels.append(current_node.symbol)
            current_node = huffman_tree
        if len(decoded_pixels) == num_pixels:
            break
    return np.array(decoded_pixels)
```

```
# Load the watermark image and cover image
watermark_image_path = 'G:/New folder (6)/misc/Image13.tiff'  # Update with the correct path
cover_image_path = 'G:/New folder (6)/misc/Image12.tiff'  # Update with the correct path


# Load and convert images
watermark_image = Image.open(watermark_image_path).convert('L')
watermark_image = np.array(watermark_image)

cover_image = Image.open(cover_image_path).convert('L')
cover_image = np.array(cover_image)
```

Converting both the Cover & the Watermark Image to Grayscale for simplicity

```
# Step 1: Huffman compression on the watermark image
frequencies = calculate_frequencies(watermark_image)
huffman_tree = build_huffman_tree(frequencies)
huffman_codes = generate_huffman_codes(huffman_tree)
encoded_bitstream = encode_image_with_huffman(watermark_image, huffman_codes)

# Step 2: Embed the Huffman encoded bitstream into the LSB of the cover image
watermarked_image = embed_bitstream_in_image(cover_image, encoded_bitstream)

# Save and display the watermarked image
watermarked_image.save('G:/New folder (6)/misc/ImageWatered670.png')

# Step 3: Extract the bitstream from the watermarked image
bitstream_length = len(encoded_bitstream)
extracted_bitstream = extract_bitstream_from_image(watermarked_image, bitstream_length)

# Step 4: Decode the bitstream to reconstruct the watermark image
num_pixels = watermark_image.size
decoded_pixels = decode_huffman_bitstream(extracted_bitstream, huffman_tree, num_pixels)
reconstructed_image = decoded_pixels.reshape(watermark_image.shape)

# Save and display the reconstructed watermark image
reconstructed_image = Image.fromarray(reconstructed_image.astype(np.uint8))
reconstructed_image.save('G:/New folder (6)/misc/ImageRecWatered670.png')

# Plot the images
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title('Original Watermark Image')
plt.imshow(watermark_image, cmap='gray')

plt.subplot(1, 2, 2)
plt.title('Reconstructed Watermark Image')
plt.imshow(reconstructed_image, cmap='gray')
plt.show()
```