Huffman coding is a variable-length prefix coding technique. It relies on creating a variable-length code for each symbol (in this case, pixel values) based on their frequency of occurrence in the image. It does not involve any mathematical transforms.

In [60]:
```
import heapq
import collections
from PIL import Image

image_path = r"img.jpeg"
image = Image.open(image_path)
image = image.resize((1024, 1024))

pixel_values = list(image.getdata())

frq = collections.Counter(pixel_values)

heap = [[weight, [pixel, ""]] for pixel, weight in frq.items()]
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:])

huffman_dict = dict(heap[0][1:])

encoding = ''.join(huffman_dict[pixel] for pixel in pixel_values)

original_size = len(pixel_values) * 8  # Assuming 8 bits per pixel
compressed_size = len(encoding)

huffman_compression_ratio = original_size / compressed_size
print(f"Huffman Compression Ratio: {huffman_compression_ratio:}")
```

```
Huffman Compression Ratio: 2.488950814740001
```

DCT is a mathematical transform that converts image data into a frequency-domain representation. It is widely used in JPEG compression. DCT captures the frequency components of an image, allowing for efficient compression by quantizing high-frequency components.

In [65]:
```
import numpy as np
from scipy.fftpack import dct
from PIL import Image

image_path = r'img.jpeg'
image = Image.open(image_path)
# print(image.size)
image = image.resize((1024, 1024))

gray = image.convert("L")
```

```
imarr = np.array(gray)
# print(imarr.shape)
block_size = 8

quantmat = np.array([[16, 11, 10, 16, 24, 40, 51, 61],
                     [12, 12, 14, 19, 26, 58, 60, 55],
                     [14, 13, 16, 24, 40, 57, 69, 56],
                     [14, 17, 22, 29, 51, 87, 80, 62],
                     [18, 22, 37, 56, 68, 109, 103, 77],
                     [24, 35, 55, 64, 81, 104, 113, 92],
                     [49, 64, 78, 87, 103, 121, 120, 101],
                     [72, 92, 95, 98, 112, 100, 103, 99]])

def quantize(k, Q):
    return np.round(k / Q)

def dequantize(l, Q):
    return l * Q

height, width = imarr.shape
dctb = np.zeros_like(imarr)

for i in range(0, height, block_size):
    for j in range(0, width, block_size):
        block = imarr[i:i + block_size, j:j + block_size]
        dct_block = dct(dct(block, axis=0), axis=1)

        qb = quantize(dct_block, quantmat)
        dctb[i:i + block_size, j:j + block_size] = qb

original = height * width * 8
compressed = dctb.size * np.log2(quantmat.max())

compression_ratio = original / compressed

print(f"Compression Ratio: {compression_ratio:}")
```

```
Compression Ratio: 1.1562593052715515
```

KL transform (Karhunen-Loève transform) is a linear transformation that converts data into a set of uncorrelated variables (principal components). It is used for decorrelation and dimensionality reduction.

In [62]:

```
import matplotlib.pyplot

image_path = r'img.jpeg'
image = Image.open(image_path)
image = image.resize((1024, 1024))

gray = image.convert("L")
imarr = np.array(gray)
covariance_matrix = np.cov(imarr.astype(float))
e1, e = np.linalg.eigh(covariance_matrix)

ind = np.argsort(e1)[::-1]
e1 = e1[ind]
e = e[:, ind]

compression_ratio = 0.1
```

```
eign = int(compression_ratio * len(e1))
sel = e[:, :eign]
compressed_image = np.dot(sel.T, imarr.T).T
newimage = np.dot(compressed_image, sel.T)

original = imarr.size * 8
compressed = eign * (len(e1) + 1) * 64

compression_efficiency = original / compressed

print(f"Compression Efficiency: {compression_efficiency:.2f}")

# matplotlib.pyplot.imshow(image, cmap='gray')
# matplotlib.pyplot.imshow(newimage, cmap='gray')
```

```
Compression Efficiency: 1.25
```

Haar wavelet transform is a mathematical technique that decomposes an image into wavelet coefficients representing different levels of detail.

In [64]:
```
import numpy as np
from PIL import Image
import pywt
import sys

image_path = r'img.jpeg'
image = Image.open(image_path)
image = image.resize((1024, 1024))
gray = image.convert("L")

imarr = np.array(gray)

coeffs = pywt.dwt2(imarr, 'haar')
threshold = 15.0
qc = [np.where(np.abs(coef) < threshold, 0, coef) for coef in coeffs]

reconstructed_image = pywt.idwt2(qc, 'haar')

original_size_bits = imarr.size * 8
compressed_size_bits = sum([np.sum(np.abs(coef) > 0) * np.ceil(np.log2(np

compression_efficiency = original_size_bits / compressed_size_bits

print(f"Compression Efficiency: {compression_efficiency:.2f}")
```

```
Compression Efficiency: 2.79
```

Conclusion:

Huffman Coding: Huffman coding is typically used for lossless compression, so it preserves image quality but may not achieve very high compression ratios.

DCT Coding: DCT coding is often used for lossy compression. The trade-off between image quality and compression ratio can be adjusted by varying the quantization step size.

KL Transform-Based Coding: KL transform can be used for both lossless and lossy compression, offering a flexible trade-off between image quality and compression ratio.

Haar Wavelet Compression: Haar wavelet compression can also be used for both lossless and lossy compression, providing control over image quality and compression ratio.