Lab 9: The Discrete Cosine Transform and JPEG

In [2]:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy as scp
```

In [3]:

```
# Define any utility functions (i.e. inner product, FFT). Feel free to reuse as
def inner_prod_2D(x, y):
  # this does the same thing but vectorized and faster.
  return np.sum(x * np.conj(y))

  # N = np.shape(x)[0]
  # inner_prod = 0
  # for i in range(N):
  #     for j in range(N):
  #         inner_prod += x[i,j] * np.conj(y[i,j])
  # return inner_prod

def complex_exp_2D(N, k, l):
  """
  k, l are frequencies. k is frequency in x direction, l is y.
  N is the duration of the signal
  returns three matrices of size NxN, the first containing the complex values of
  e_kl,NN. the second is the real part; the third is the imaginary part.
  """
  x_tiles = np.tile(np.arange(N), (N, 1))
  y_tiles = np.tile(np.arange(N), (N, 1)).T
  partial_1 = k * x_tiles / N
  partial_2 = l * y_tiles / N
  cexp = 1/N * np.exp(-2j * np.pi * (partial_1 + partial_2))

  cexp_real = np.real(cexp)
  cexp_imag = np.imag(cexp)
  return cexp, cexp_real, cexp_imag

def compute_DFT_2D(x):
  # X is a 2D array containing the coefficients of the 2D DFT of the NxN dimensi
  N = np.shape(x)[0]
  X = np.zeros([N, N], dtype=np.complex128)
  for k in range(N):
    for l in range(N):
      cexp, _, _ = complex_exp_2D(N, k, l)
      X[k, l] = inner_prod_2D(x, cexp)
  return X


def compute_iDFT_2D(X):
  # x is an NxN dimensional signal whose DFT is given by X
  # X has dimensions of... NxN
  # it goes from 0 to N-1 along each dimension.
  N = np.shape(X)[0]
  x = np.zeros((N, N), dtype=np.complex128)

  for m in range(N):
```

```python
    for n in range(N):
        sum_val = 0

        # Process the first half of frequencies
        for k in range(N//2 + 1):  # 0 to N/2
            for l in range(N):
                exponent = 2j * np.pi * (m * k / N + n * l / N)
                current_term = X[k, l] * np.exp(exponent)

                # Add contribution from the conjugate pair
                # Skip for k=0 (which are their own conjugates)
                if k > 0:
                    conj_k = (N - k) % N
                    conj_l = (N - l) % N
                    conj_exponent = 2j * np.pi * (m * conj_k / N + n * conj_l
                    current_term += np.conj(X[k, l]) * np.exp(conj_exponent)

                sum_val += current_term

        # convert coordinate system to match original
        x[N-m if m != 0 else m, N-n if n != 0 else n] = (sum_val / N)

  return x.T
```

1. Image Compression

1.1 DCT in Two Dimensions

In [4]:
```python
def discrete_cosine(k, l, N):
  x_tiles = np.tile(np.arange(N), (N, 1))
  x_tiles = 2*x_tiles +1
  y_tiles = np.tile(np.arange(N), (N, 1)).T
  y_tiles = 2*y_tiles +1
  partial_1 = k * x_tiles / (2*N)
  partial_2 = l * y_tiles / (2*N)
  cos = np.cos(np.pi * partial_1)*np.cos(np.pi * partial_2)
  return cos
```

In [5]:
```python
def compute_2D_DCT(x):
  """
  X = scp.fft.dct(scp.fft.dct(x, axis=0, norm='ortho'), axis=1, norm='ortho')
  return X
  """
  N = np.shape(x)[0]
  X = np.zeros([N, N], dtype=np.complex128)
  for k in range(N):
    for l in range(N):
      cos = discrete_cosine(k, l,N)
      if k == 0:
        c1 = 1/np.sqrt(2)
      else:
        c1 = 1
      if l == 0:
        c2 = 1/np.sqrt(2)
      else:
        c2 = 1
```

```
    X[k, l] = inner_prod_2D(x, c1*c2*cos)*(2/N)
  return X
```

In [6]:
```
def compute_2D_iDCT(X):
  '''
  x = scp.fft.idct(scp.fft.idct(X, axis=0, norm='ortho'), axis=1, norm='ortho')
  return x
  N = np.shape(X)[0]
  Xs = np.zeros([N, N], dtype=np.complex128)
  for k in range(N):
    for l in range(N):
      if k == 0:
        alpha_k = 1/np.sqrt(2)
      else:
        alpha_k = 1
      Xs[k, l] = inner_prod_2D(X, (alpha_k**2)*discrete_cosine(k, l, N).T*discre
  return Xs*(2/N)
  '''

  N = np.shape(X)[0]
  x = np.zeros([N, N], dtype=np.complex128)
  normalization_factor = 2/N
  for m in range(N):
    for n in range(N):
      for k in range(N):
        for l in range(N):
          if k == 0:
            c1 = 1/np.sqrt(2)
          else:
            c1 = 1
          if l == 0:
            c2 = 1/np.sqrt(2)
          else:
            c2 = 1

          x[m,n] += X[k, l] * c1 * c2 * np.cos(np.pi * k * (2*m+1)/(2*N)) * np.c

  return x.T * normalization_factor
```

In [ ]:
```
# If your functions run too slow, consider using scp.fft.dct
```

1.2 Image Compression

In [7]:
```
def compress_block_DCT(x: np.ndarray, K: int):
  X = compute_2D_DCT(x)
  X_magnitude = np.abs(X)
  threshold = np.sort(X_magnitude.flatten())[-K]
  X[X_magnitude<=threshold] = 0
  return compute_2D_iDCT(X)


def compress_block_DFT(x, K):
  # Implement function to compress an 8x8 block using DFT
  X = compute_DFT_2D(x)
  X_magnitude = np.abs(X)
  threshold = np.sort(X_magnitude.flatten())[-K]
```

```python
    X[X_magnitude<threshold] = 0
    return compute_iDFT_2D(X)
```

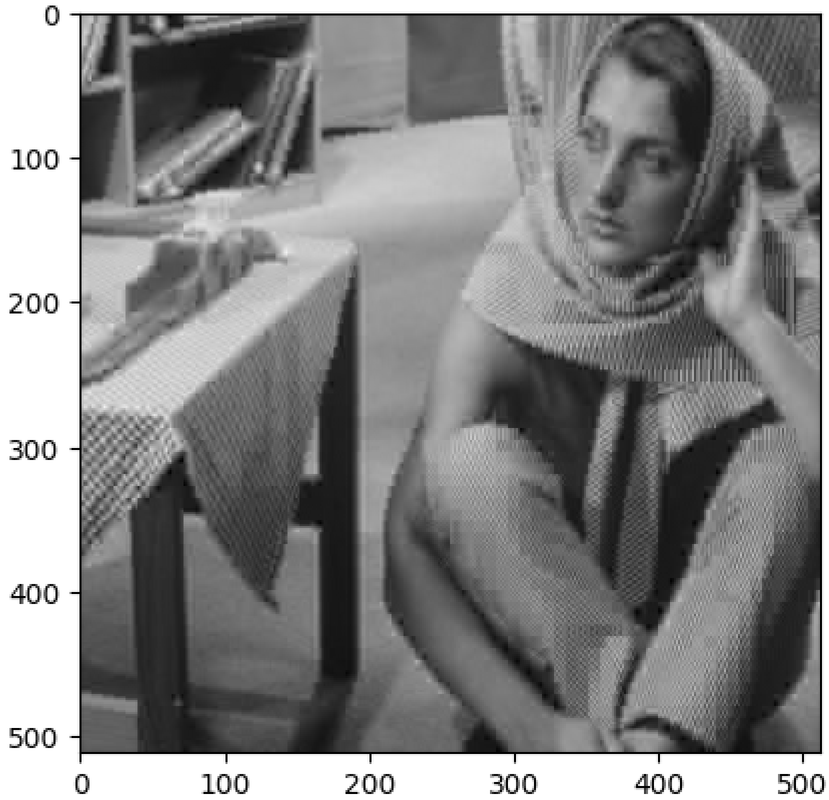
In [8]:
```python
def compress_image_DCT(x, K):
    H, W = x.shape  # Get dimensions of the input array
    blocks = []

    for i in range(0, H, 8):
        for j in range(0, W, 8):
            blocks.append(x[i:i+8, j:j+8])

    for i in range(len(blocks)):
        blocks[i] = compress_block_DCT(blocks[i], K)
    new_array = np.zeros((H, W))  # Create empty array
    index = 0
    for i in range(0, H, 8):
        for j in range(0, W, 8):
            new_array[i:i+8, j:j+8] = blocks[index]
            index += 1
    return np.abs((new_array.real))

def compress_image_DFT(x, K):
  # Implement function to partition an image into 8x8 blocks and compress the bl
    H, W = x.shape  # Get dimensions of the input array
    blocks = []

    for i in range(0, H, 8):
        for j in range(0, W, 8):
            blocks.append(x[i:i+8, j:j+8])

    for i in range(len(blocks)):
        blocks[i] = compress_block_DFT(blocks[i], K)
    new_array = np.zeros((H, W))  # Create empty array
    index = 0
    for i in range(0, H, 8):
        for j in range(0, W, 8):
            new_array[i:i+8, j:j+8] = blocks[index]
            index += 1
    return new_array.real
```

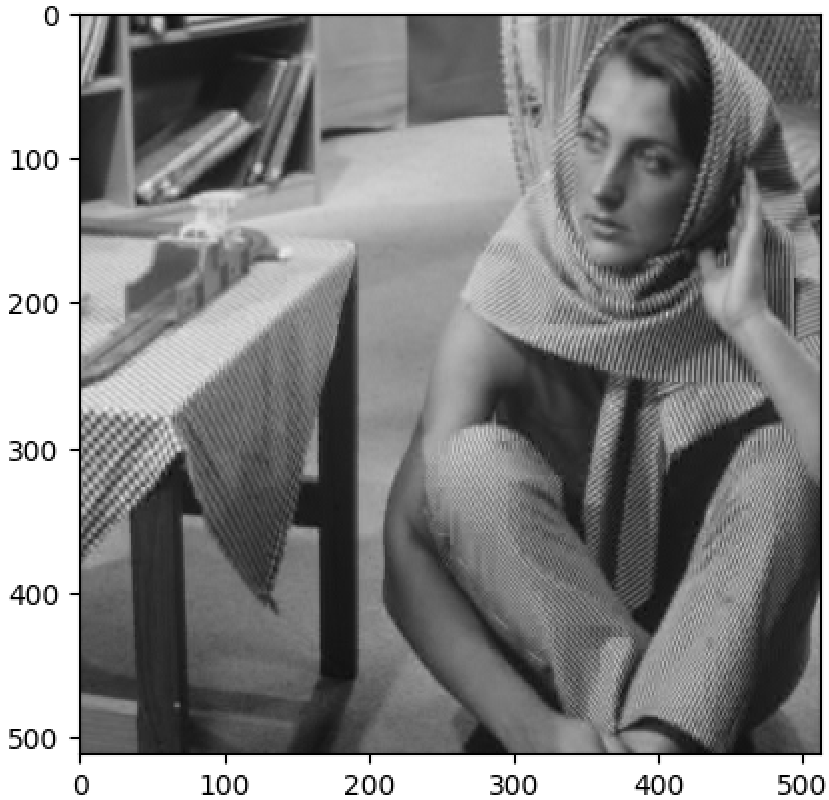
In [10]:
```python
imgB_prenoise = plt.imread("imgB_prenoise.png")
```

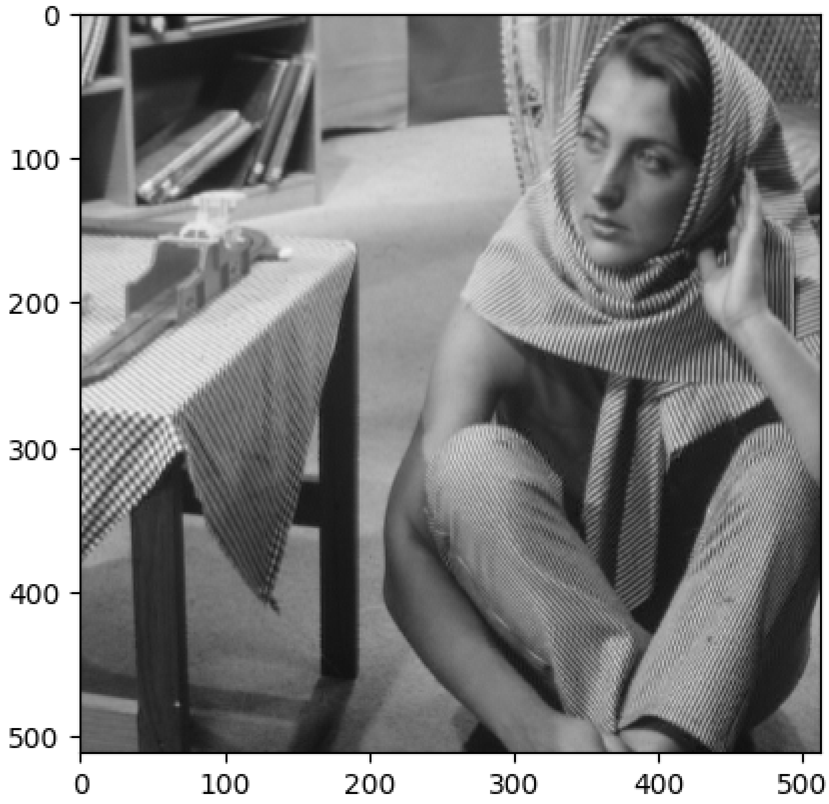
In [11]:
```python
# Try these functions out on the provided sample image A for K = 4, 8, 16, 32
plt.figure()
output1 = compress_image_DCT(imgB_prenoise, 4)
plt.imshow(output1, cmap='gray')
plt.figure()
output2 = compress_image_DCT(imgB_prenoise, 8)
plt.imshow(output2, cmap='gray')
plt.figure()
output3 = compress_image_DCT(imgB_prenoise, 16)
plt.imshow(output3, cmap='gray')
plt.figure()
output4 = compress_image_DCT(imgB_prenoise, 32)
plt.imshow(output4, cmap='gray')
```

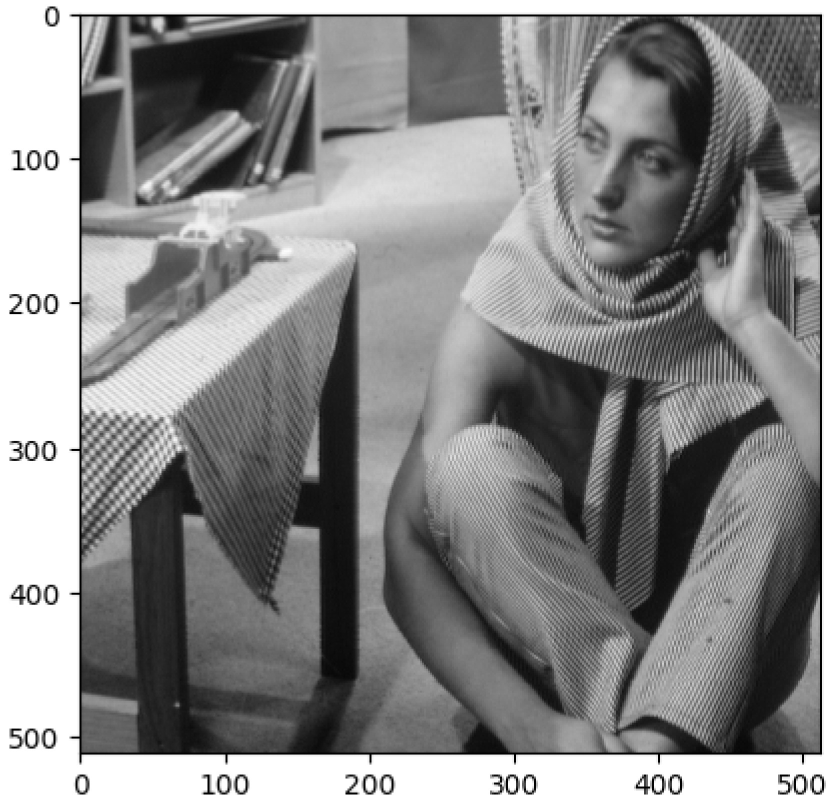
```
<ipython-input-8-cd5e73dc7f87>:15: ComplexWarning: Casting complex values to real discards the imaginary part
  new_array[i:i+8, j:j+8] = blocks[index]
```

Out[11]: `<matplotlib.image.AxesImage at 0x7ee17c646790>`







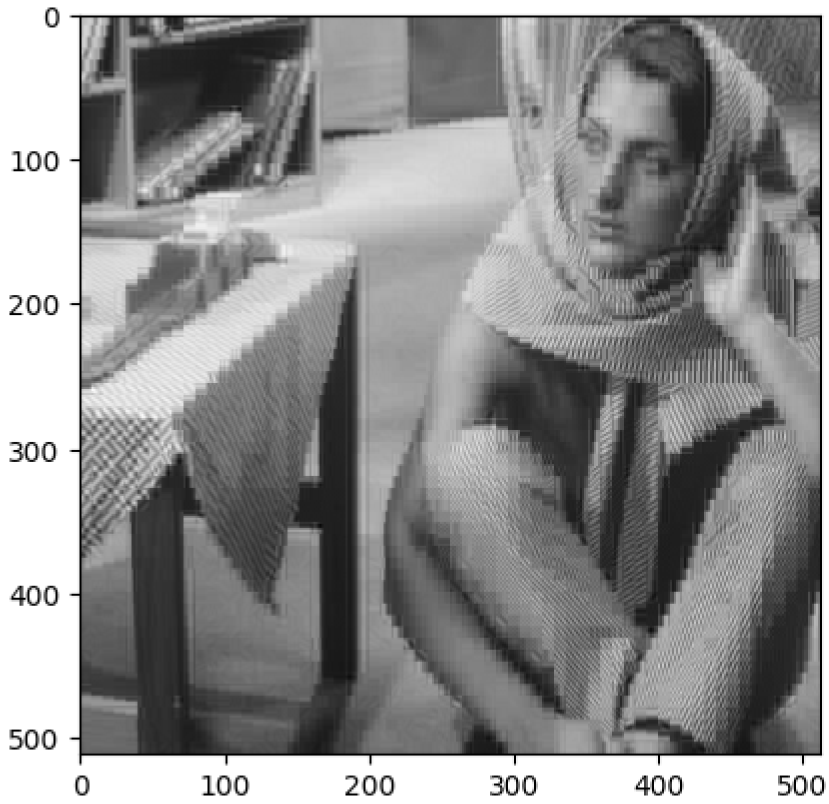


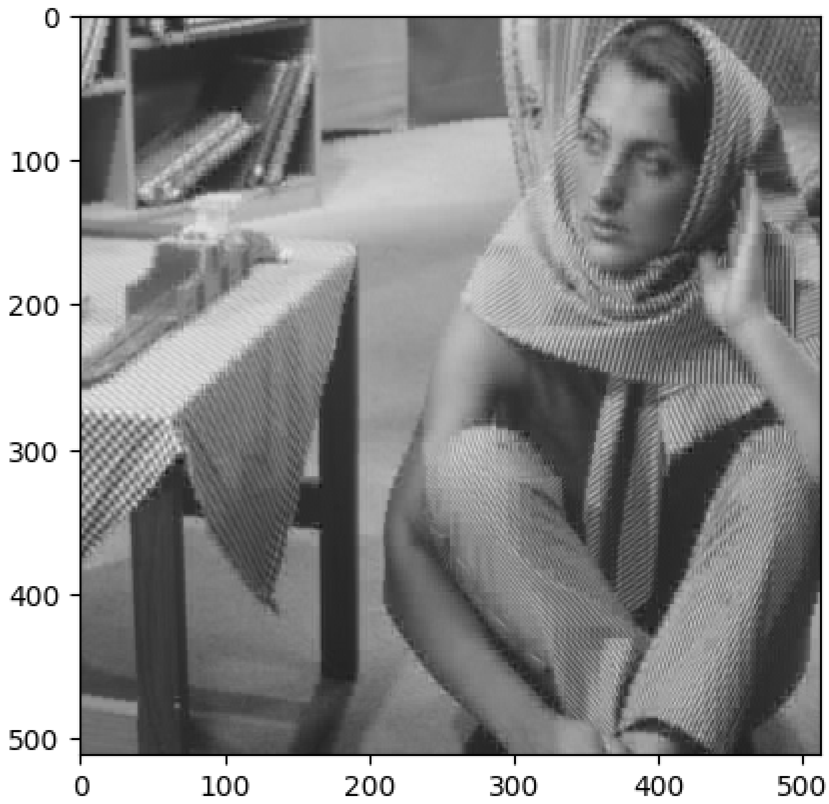
In [12]:
```
#now for the DFT
plt.figure()
output5 = compress_image_DFT(imgB_prenoise, 4)
plt.imshow(output5, cmap='gray')
plt.figure()
output6 = compress_image_DFT(imgB_prenoise, 8)
plt.imshow(output6, cmap='gray')
plt.figure()
output7 = compress_image_DFT(imgB_prenoise, 16)
```

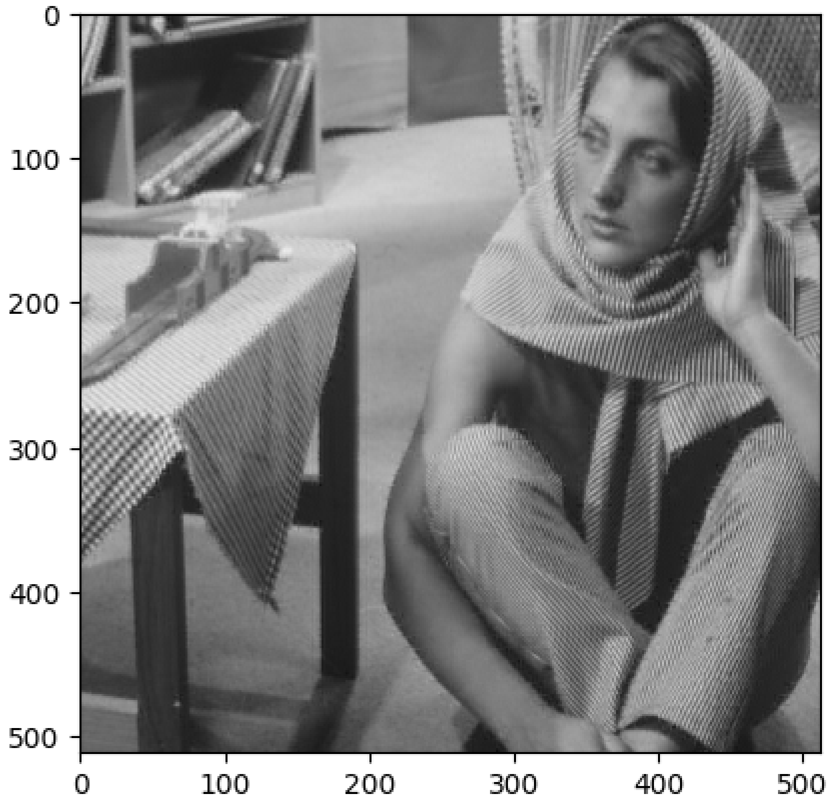
```
plt.imshow(output7, cmap='gray')
plt.figure()
output8 = compress_image_DFT(imgB_prenoise, 32)
plt.imshow(output8, cmap='gray')
```

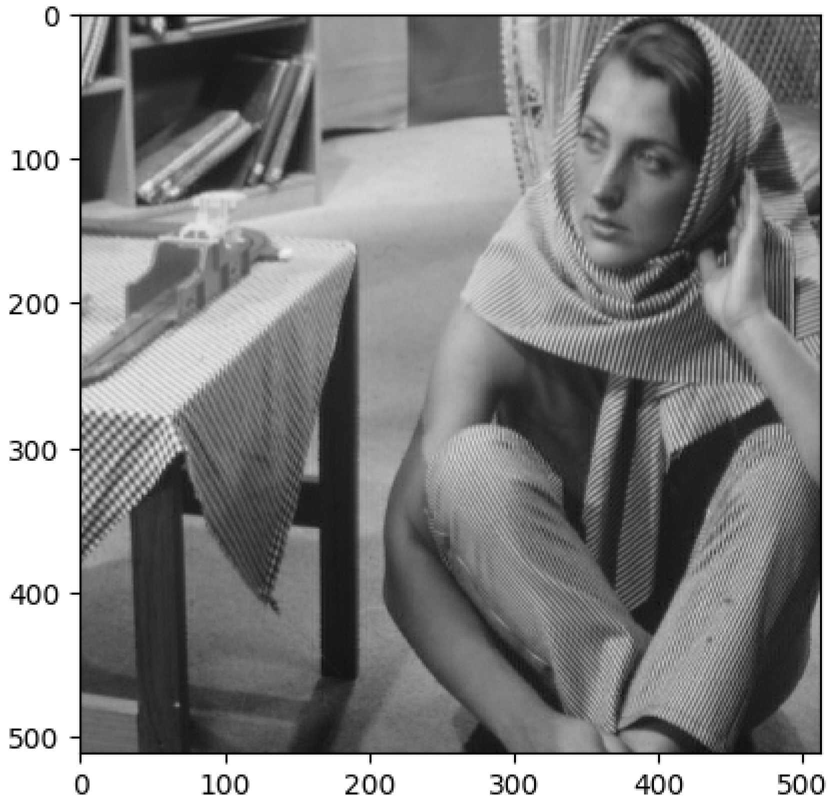
```
<ipython-input-8-cd5e73dc7f87>:34: ComplexWarning: Casting complex values to real discards the imaginary part
  new_array[i:i+8, j:j+8] = blocks[index]
```

Out[12]: <matplotlib.image.AxesImage at 0x7ee16478b710>









We can clearly see the DFT reconstructions getting clearer as K increases.

1.3 Quantization

In [13]:
```
Q = 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],
```

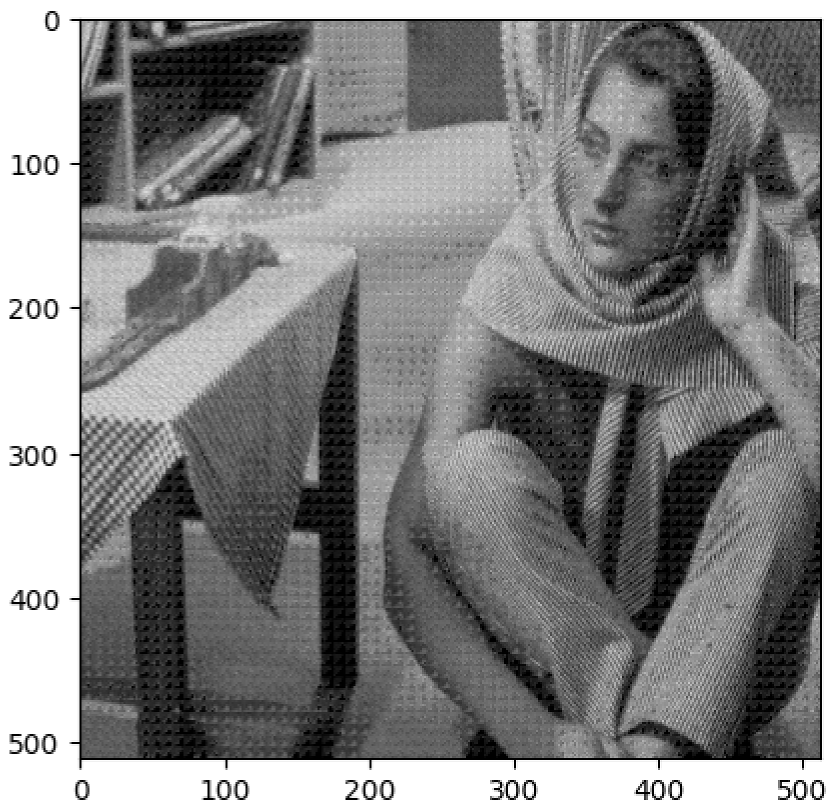
```
                    [18, 22, 37, 56, 68, 109, 103, 77],
                    [24, 36, 55, 64, 81, 104, 113, 92],
                    [49, 64, 78, 87, 103, 121, 120, 101],
                    [72, 92, 95, 98, 112, 100, 103, 99]])
```

In [14]:
```
def quantize(Q, block):
  # Implement function to quantize an 8x8 block
  #print(block)
  return np.round(block/Q)

def dequantize(Q, block):
  # Implement function to dequantize an 8x8 block
  return block*Q
```

In [15]:
```
def quantize_image(Q, x):
  # Implement function to quantize an image
    H, W = x.shape  # Get dimensions of the input array
    blocks = []
    for i in range(0, H, 8):
        for j in range(0, W, 8):
            blocks.append(x[i:i+8, j:j+8])

    for i in range(len(blocks)):
        blocks[i] = quantize(Q,blocks[i])
    new_array = np.zeros((H, W))  # Create empty array
    index = 0
    for i in range(0, H, 8):
        for j in range(0, W, 8):
            new_array[i:i+8, j:j+8] = blocks[index]
            index += 1
    return np.abs((new_array.real))

def dequantize_image(Q, x):
  # Implement function to dequantize an image
    H, W = x.shape  # Get dimensions of the input array
    blocks = []

    for i in range(0, H, 8):
        for j in range(0, W, 8):
            blocks.append(x[i:i+8, j:j+8])

    for i in range(len(blocks)):
        blocks[i] = dequantize(Q,blocks[i])
    new_array = np.zeros((H, W))  # Create empty array
    index = 0
    for i in range(0, H, 8):
        for j in range(0, W, 8):
            new_array[i:i+8, j:j+8] = blocks[index]
            index += 1
    return np.abs((new_array.real))
```

In [16]:
```
imgB_prenoise = plt.imread("imgB_prenoise.png")
plt.figure()
output10 = quantize_image(Q,imgB_prenoise*255)
output10 = dequantize_image(Q,output10)
output10 = output10/255
plt.imshow(output10, cmap='gray')
```

Out[16]:
```
<matplotlib.image.AxesImage at 0x7ee16460a550>
```



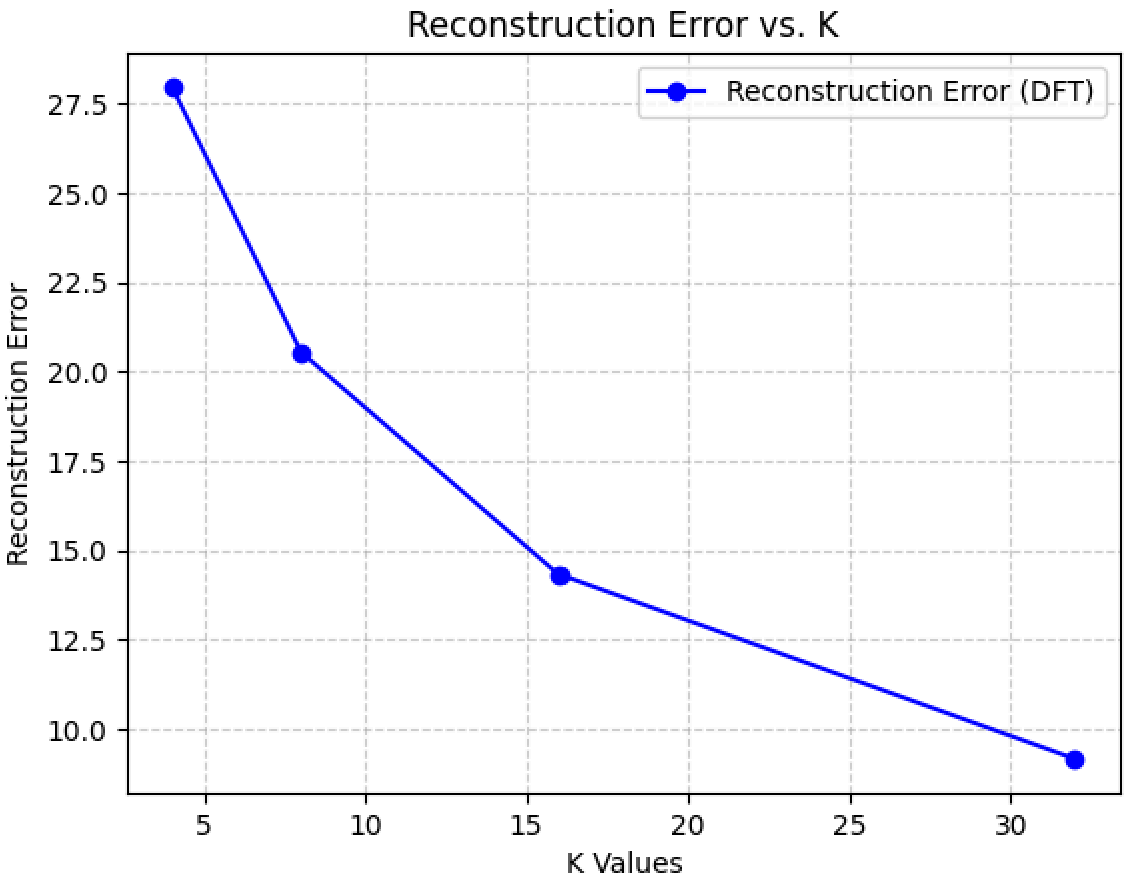
1.4 Image Reconstruction

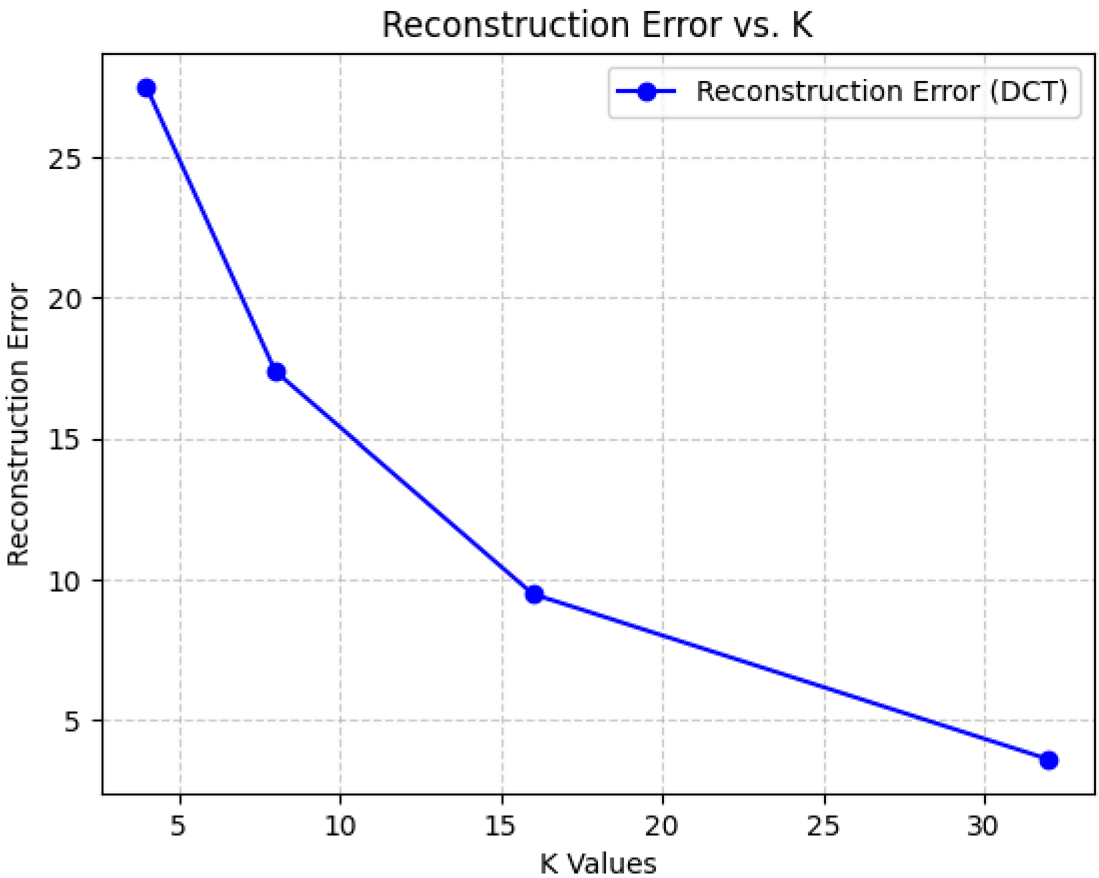
In [17]:
```
# Load image
img = plt.imread('imgB_prenoise.png')
```

In [18]:
```
def get_reconstruction_error(x_bar, x):
  # Calculate distance between x_bar and x
  return np.linalg.norm(x_bar - x)
```

In [19]:
```
# Compress and reconstruct using functions from 1.2
# Make plots of reconstructions for K = 4, 8, 16, 32
# Calculate reconstruction error for different values of K, and plot
#DFT
K_values = [4, 8, 16, 32]
DFT_outputs = [output5, output6, output7, output8]
errors = []
for k, x_bar in zip(K_values, DFT_outputs):
  error = get_reconstruction_error(x_bar, img)
  errors.append(error)
plt.figure()
plt.plot(K_values, errors, marker='o', linestyle='-', color='b', label="Reconstr

# Labels and Title
plt.xlabel("K Values")
plt.ylabel("Reconstruction Error")
plt.title("Reconstruction Error vs. K")
plt.grid(True, linestyle="--", alpha=0.6)
plt.legend()

# Show the plot
plt.show()
```

```
################################################################################
#DCT
K_values = [4, 8, 16, 32]
DCT_outputs = [output1, output2, output3, output4]
errors = []
for k, x_bar in zip(K_values, DCT_outputs):
  error = get_reconstruction_error(x_bar, img)
  errors.append(error)
plt.figure()
plt.plot(K_values, errors, marker='o', linestyle='-', color='b', label="Reconstr

# Labels and Title
plt.xlabel("K Values")
plt.ylabel("Reconstruction Error")
plt.title("Reconstruction Error vs. K")
plt.grid(True, linestyle="--", alpha=0.6)
plt.legend()

# Show the plot
plt.show()
```

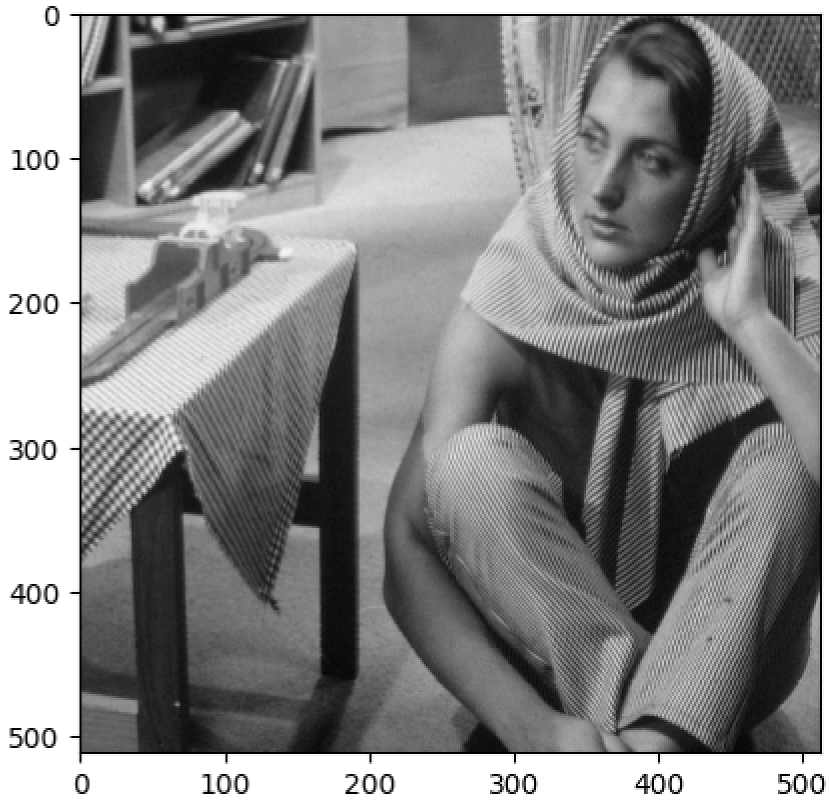




Comments: As we can see the more we compress signals the more we lose information which is a general trend in all compression schemes.

In [20]:
```
# Compress and reconstruct using functions from 1.3 (quantization)
# Make plots of reconstructions using different values of the quantization matri
Q1 = np.array([[9, 3, 4, 5, 12, 11, 8, 6],
               [11, 15, 4, 3, 1, 9, 8, 7],
               [14, 13, 6, 4, 4, 7, 9, 6],
               [4, 7, 2, 9, 5, 8, 8, 6],
               [8, 2, 7, 6, 8, 9, 13, 7],
               [4, 6, 5, 6, 8, 4, 13, 9],
               [9, 6, 8, 8, 13, 21, 12, 11],
               [7, 9, 9, 8, 12, 10, 13, 9]])
plt.figure()
output11 = quantize_image(Q1,img*255)
output11 = dequantize_image(Q1,output11)
output11 = output11/255
plt.imshow(output11, cmap='gray')
```

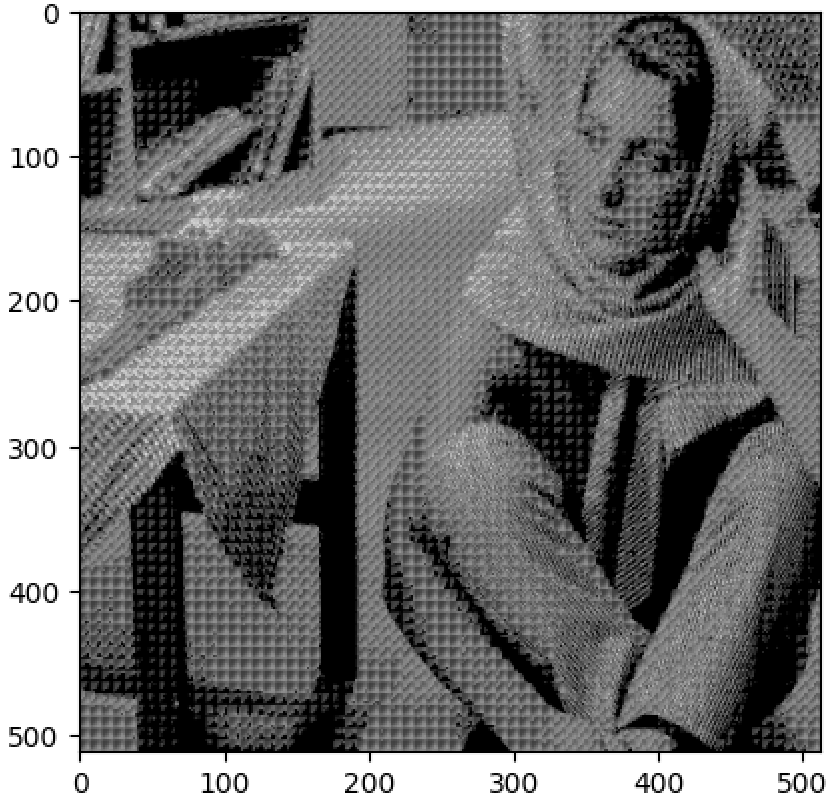
Out[20]: <matplotlib.image.AxesImage at 0x7ee17c547a10>



Comment: Here as we can see we have very low Q that does a decent compression (maybe lowers the size by a factor of two) while also providing a very detailed picture with nearly no lose of information.

In [21]:
```
Q2 = 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, 36, 55, 64, 81, 104, 113, 92],
                 [49, 64, 78, 87, 103, 121, 120, 101],
                 [72, 92, 95, 98, 112, 100, 103, 99]])
Q2 = Q2+70
plt.figure()
output12 = quantize_image(Q2,img*255)
output12 = dequantize_image(Q2,output12)
output12 = output12/255
plt.imshow(output12, cmap='gray')
```

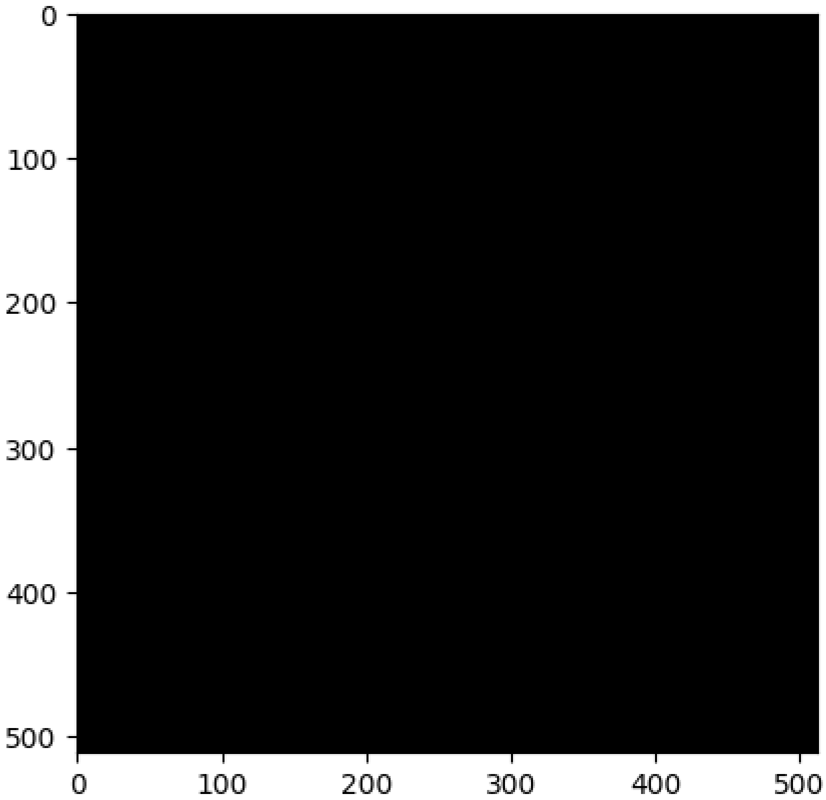
Out[21]: <matplotlib.image.AxesImage at 0x7ee17c54e490>



Comment: For moderatly high Q (that doesn't destroy information) you get decent amount of compression while also preserving the information, but as we can see the black dots got blacker and we lost a lot of shades in the process of compression.

In [22]:
```
Q3 = 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, 36, 55, 64, 81, 104, 113, 92],
                 [49, 64, 78, 87, 103, 121, 120, 101],
                 [72, 92, 95, 98, 112, 100, 103, 99]])
Q3 = Q3+700
plt.figure()
output13 = quantize_image(Q3,img*255)
output13 = dequantize_image(Q3,output13)
output13 = output13/255
plt.imshow(output13, cmap='gray')
```

Out[22]:
```
<matplotlib.image.AxesImage at 0x7ee164593910>
```



** *italicized text*Comments:** As we can see high values of Q will result in more compression but also you will lose much more information and if the Q is high enough you will lose all information.

A rule that we can deduce is High Q-> more compression -> less information and vice versa.