This assignment is to experiment various components of JPEG codec, the main components of codec are **Mapper, Quantizer and Symbol Coder**.

Introduction:

For this assignment, the following technique are used:

Discrete Cosine Transformer (DCT) is used to convert spatial domain to frequency domain, it means to represent pixels by another format. As we learnt, the main information of image is concentrated on low frequency component, that is, to convert pixels to frequency domain and only keep low frequency component will help to compress image since without high frequency component pixels are difficult to tells different by human.

We can see below matrix example (Figure 1), it shows DCT convert image from spatial domain to frequency domain and keep the most important value (the brightest one) in the top-left pixel from both DCT log scale and keep the less important pixel (the darkest one) in the bottom-right pixel. The closer to top left the lower frequency component it represent and the more close to bottom right, the higher frequency component it is.

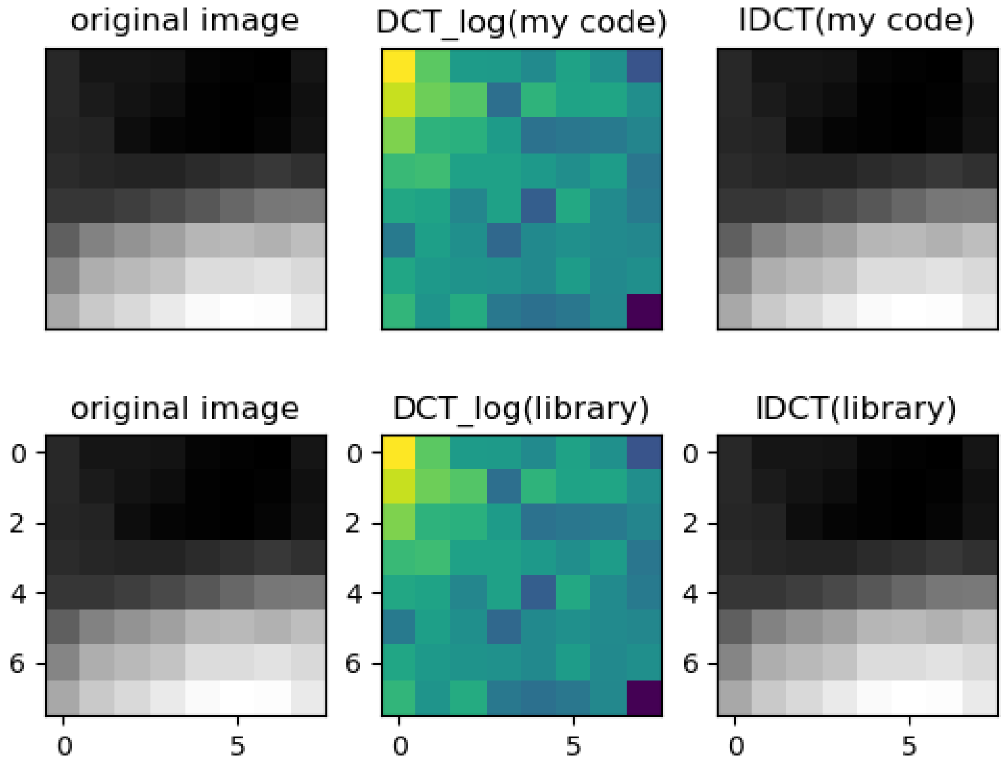

Figure 1. Sample matrix for showing my code vs library DCT and inverse DCT

DCT formula:

$$F(u,v) = c(u)c(v)\sum_{i=0}^{N-1}\sum_{i=0}^{N-1} f(i,j)\cos\left[\frac{(i+0.5)\pi}{N}u\right]\cos\left[\frac{(i+0.5)\pi}{N}v\right]$$

$$c(u) = \begin{cases} \sqrt{\frac{1}{N}}, & u = 0 \\ \sqrt{\frac{2}{N}}, & u \neq 0 \end{cases}$$

Inverse DCT(IDCT) Formula:

$$f(i,j) = \sum_{u=0}^{N-1}\sum_{v=0}^{N-1} c(u)c(v)F(u,v)\cos[\frac{(i+0.5)\pi}{N}u]\cos[\frac{(j+0.5)\pi}{N}v]$$

$$c(u) = \begin{cases} \sqrt{\frac{1}{N}}, & u = 0 \\ \sqrt{\frac{2}{N}}, & u \neq 0 \end{cases}$$

Entropy coding is an encoding method we usually use with DCT by ZigZag. ZigZag is fashion which we serialize pixels.
Below is the sequence of ZigZag to convert pixels to a list, thus we can re-arrange the order of pixel begin with the most import data (Low frequency value) and end with high frequency value.

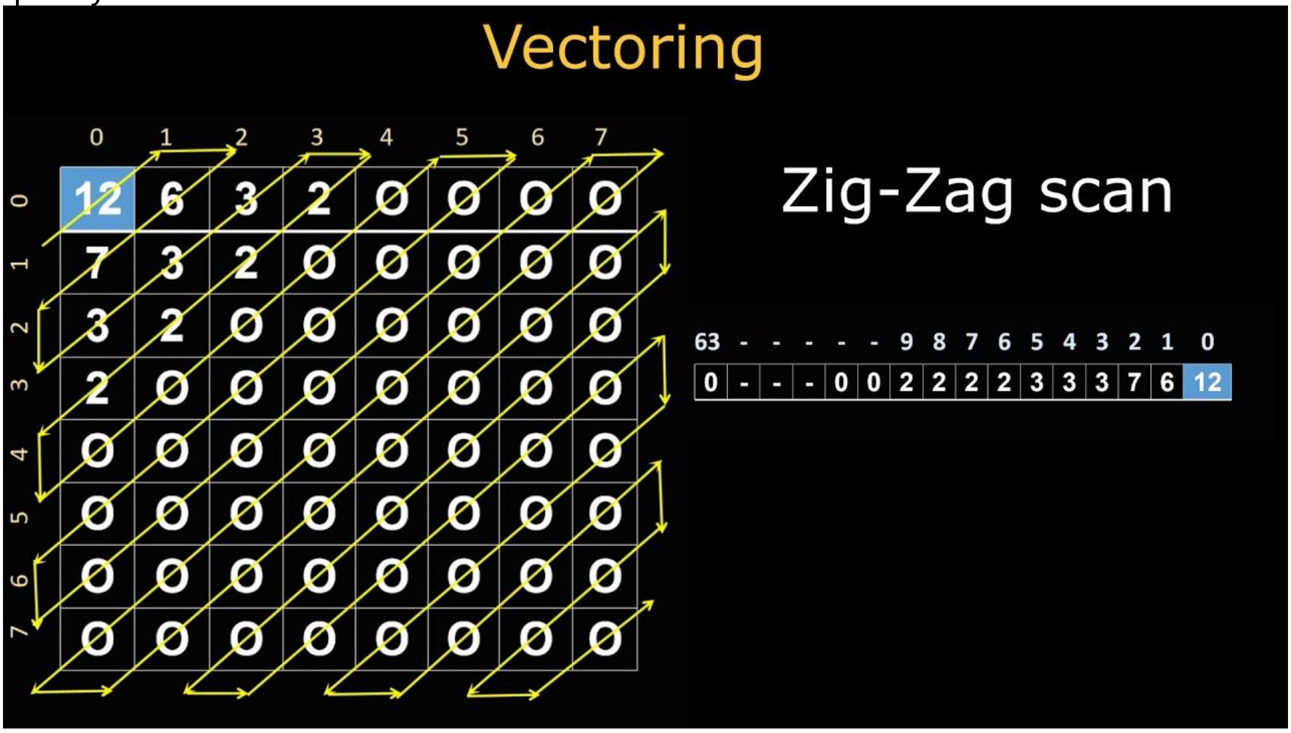

Figure 2. The Entropy coding sequence – ZigZag example

Predictive coding is an encoding method which can improve data compress rate. The goal of predictive coding is to exploit the correlation between nearby samples in the source. Usually, image will not change frequently pixel by pixel, adjacent pixels will often contain similar values for smooth area, but edges will contain sudden change value, thus, will not predict well if we are using smooth area pixel to predict edge pixel.
For example, if we have an input data like below:

f(x) = 2, 4, 6, 8, 10, 12, 12, 12, 12, 10, 8, 6, 4, 2

Let say we have predictive encoder *f(x)' = f(x) - f(x-1),* then we can represent our data like below:

f(x)' = 2, 2, 2, 2, 2, 2, 0, 0, -2, -2, -2, -2, -2, -2

Then, we can have data which contain less bit.

Run-length coding is an encoding method which use to find the same value of adjacent pixels in order to compress repeat pixels to a single data value with its's counts.
Continuous to using the example above:

$$f(x)' = 2, 2, 2, 2, 2, 2, 0, 0, -2, -2, -2, -2, -2, -2$$

we can represent our new data *f(x)'* to (6, 2), (2, 0), (6, -2), we obviously can have shorter data which can decrease our storage and it means to compress our data. This coding method have impressed compression improvement combining with DCT, Entropy coding and also Predictive coding.

Chromatic Subsampling is an encoding method, which utilize the character of human eyes are more susceptible the chances of lighting intensity than color variations. The step of Chromatic Subsampling is shown below:

1. Transfer RGB image to YCbCr where RGB mean Red, Green and Blue, Y is Luminance and CbCr are Chrominance.
2. As Luminance will contains intensity information of image which means it is the most important element which influence the view of human eyes, which we will keep the original pixel value for compression. And Chrominance is the minor information that human eyes are looking at, therefore, Chrominance is the main data which to be compressed.

Below sample (Figure 3) matrix is the original 4-4-4, subsampling4-2-0 and 4-2-2 matrix are shown below:
We can easily to tell the 4-2-0 is subsampling to 1 top-left pixel of 4 pixels
4-2-2 is subsampling the left pixel of 2 pixels.

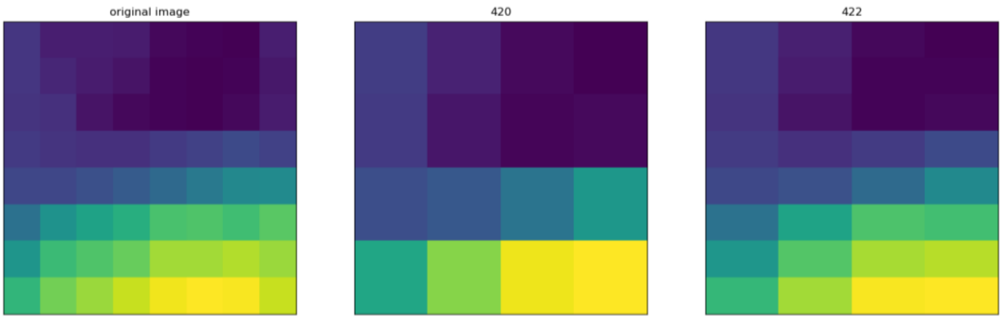

Figure 3. Sample matrix for showing Different subsampling – Original 4-4-4 and 4-2-0 and 4-2-2 from left to right with my code

Huffman coding is an encoding method that use shortest code to represent the most probability appearance pixel value and longer code to representing less probability appearance pixels values in order to save storage and speed up access rate.
See below Example(Figure 4):

(a)

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

(b)

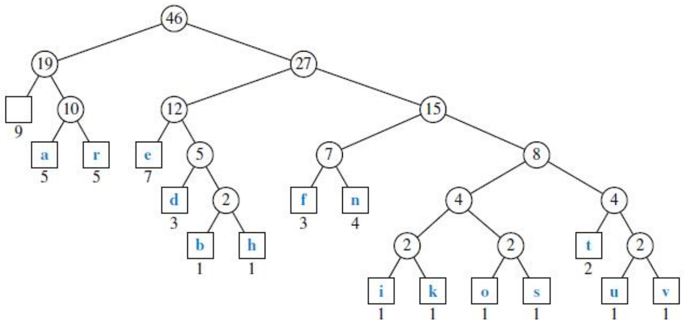

Figure 4 Huffman coding Example

The **flow of compress codec** on my assignment is follow:

Image (Transfer to YCbCr if using Chromatic Subsampling) → Mapper (DCT) → Quantizer → Symbol Encoder (Run-length coding, chromatic subsampling, Huffman coding) → Compress image

Uncompressed codec:
Compressed image (Transfer to RGB if using Chromatic Subsampling) → Symbol decoder (Run-length coding, chromatic subsampling, Huffman coding) → Quantizer → Mapper (inverse DCT)

Implementation:

First, I use gray image (House) to do DCT, then do quantizer for compress high frequency and similar data, then do encoding for compress image. Finally, I uncompressed image by decoding and show the image for comparing.
Below is the effect of my encoder and decoder:

Figure 5. Original image vs uncompressed image after compressed

We can see that the shape and the quality of image is still good to see. However, the image become darker than original image, below Figure 6 will zoom in the red box for comparison.

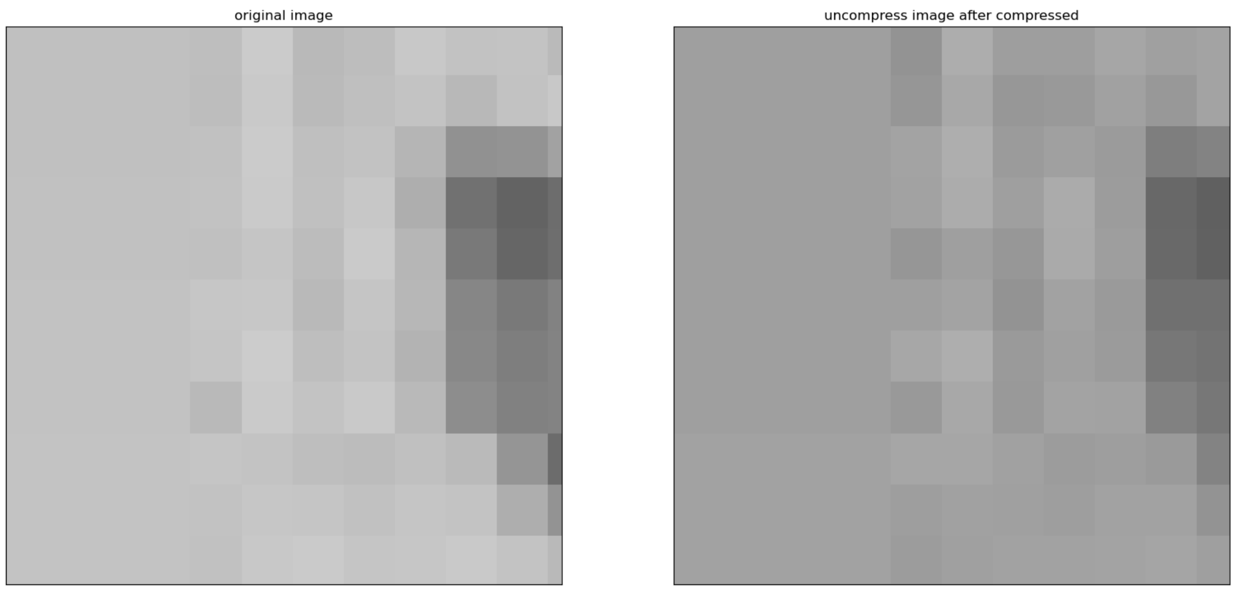

Figure 6. the Zoom in pixels comparison (Original vs Uncompressed Image)

We can see that the overall pixels become darker. However, the pattern of pixels is same as original.

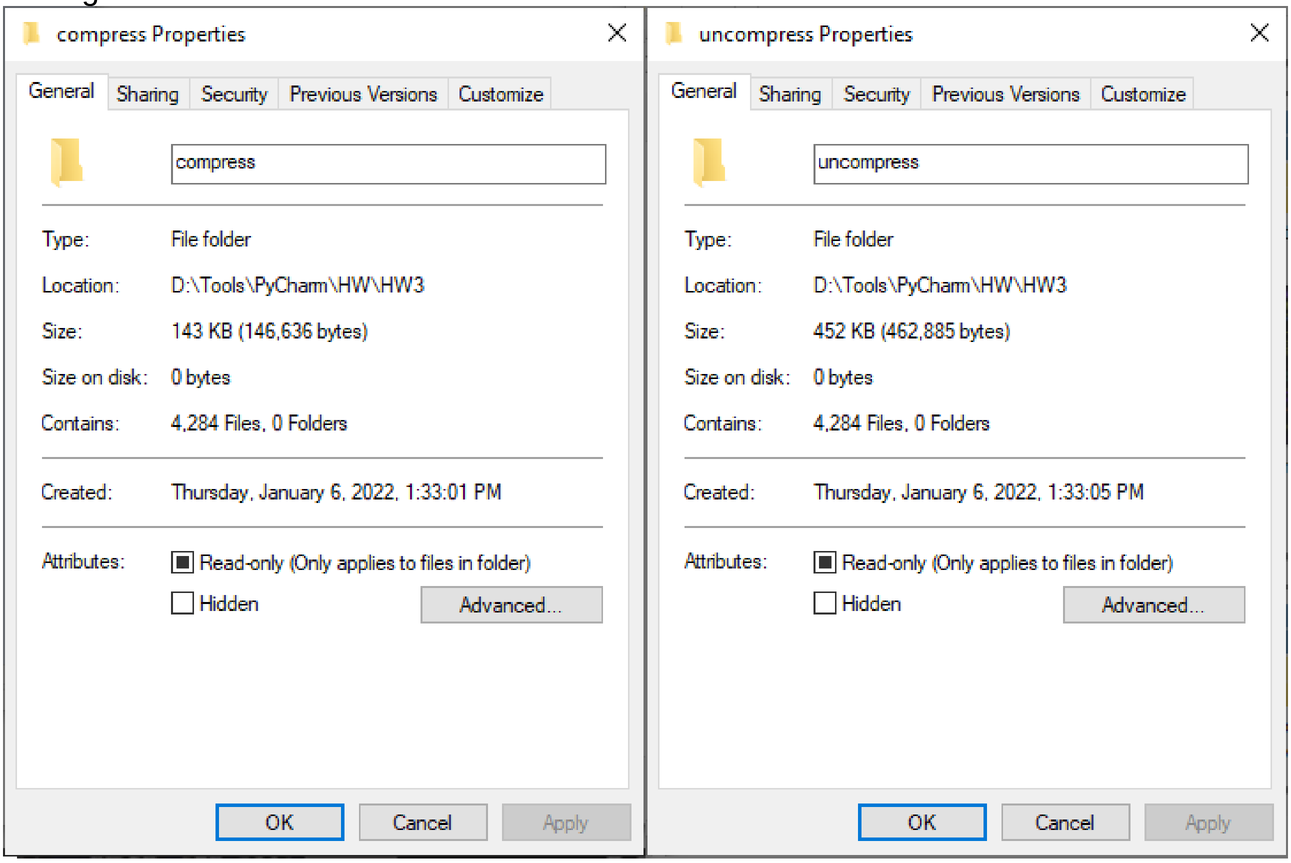

Figure 7. The size of image after compress and after uncompressed

After the compressed and uncompressed as Figure7, the image is obviously compress, that is, uncompressed file is around 3 times of compressed one.

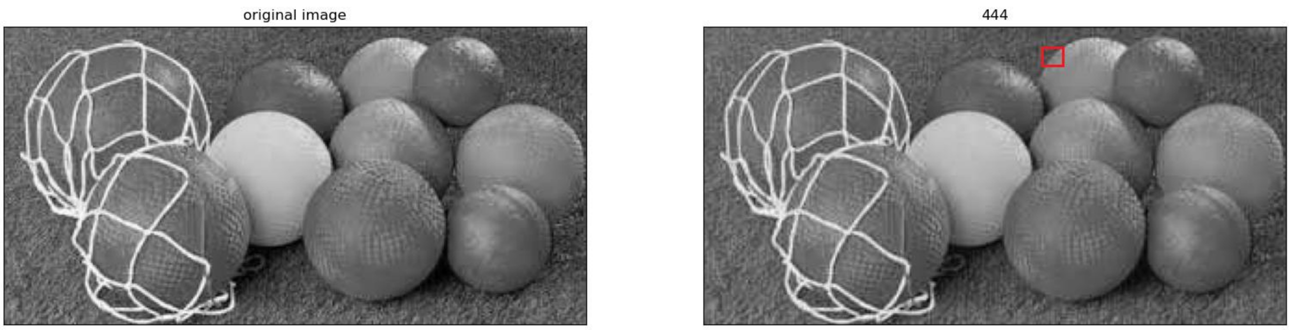

Figure 8. Original image vs uncompressed image after compressed

We can observe that the balls image cannot obviously tell the difference for overall view, therefore, I zoom in to the red box for pixel level to observe the different between two of them, we can refer to Figure 9.

In pixel level, we can see the different of my compressed and uncompressed picture, which is smoother than original, it implies the compression involve low pass filter technique as we mentioned in previous, the high frequency component has been filter out, but human eyes cannot easily tell the difference for the overall images.

Figure 10, shows the compressed picture size and uncompressed picture size using my codec, we can see that the uncompressed file is also around 3 times of compressed one.

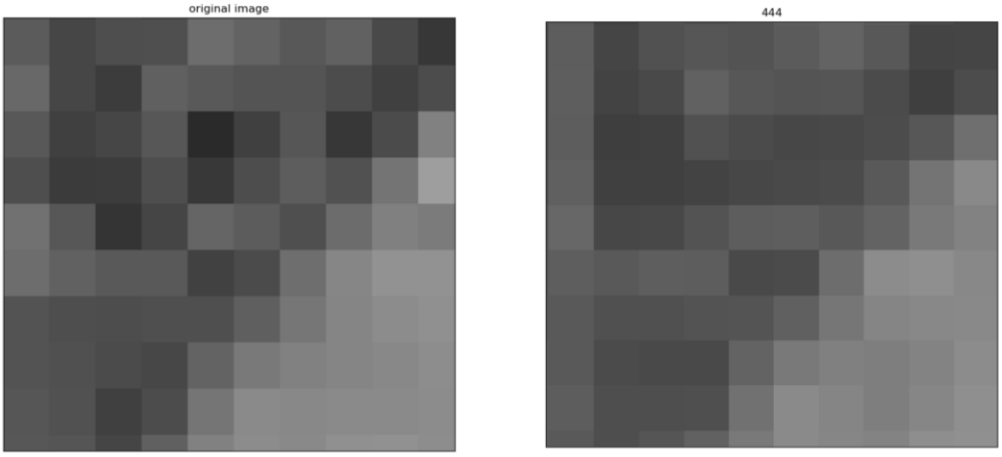

Figure 9. the Zoom in pixels comparison (Original vs Uncompressed Image)

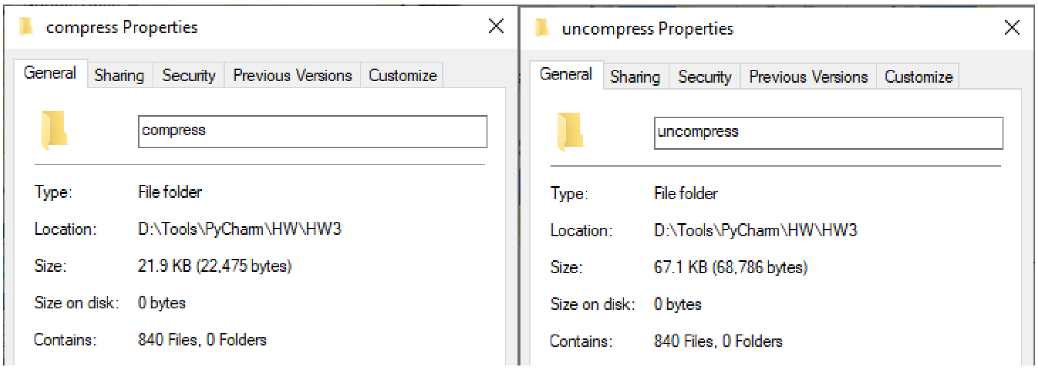

Figure 10. The size of image after compress and after uncompressed

And I try scene image for experiment, we can obtain the slightly difference, the compressed image is a little bit darker on the top and a little bit lighter on the bottom as shown on Figure 11.

And the compress size of image is shown on Figure 12.

Figure 11. Original image vs uncompressed image after compressed

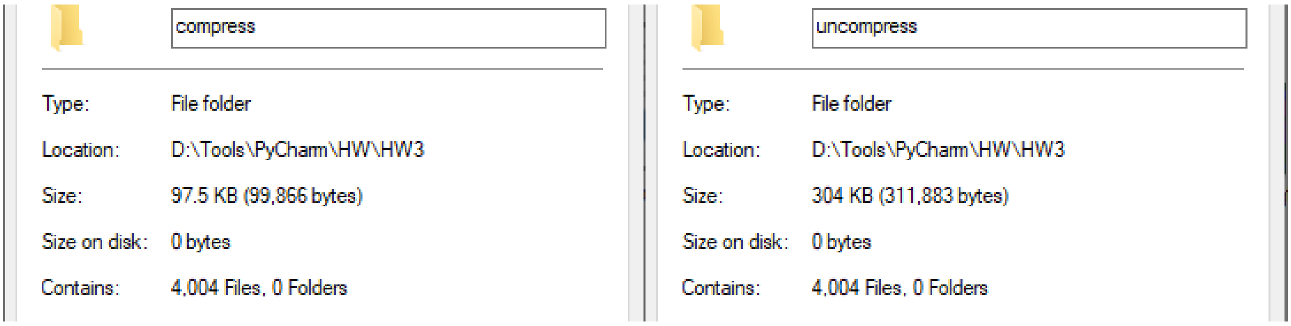

Figure 12. The size of image after compress and after uncompressed

In the Figure 13, we can obviously observe the difference between images, the compressed one have lighter value than original.

Figure 13. Original image vs uncompressed image after compressed

In Figure 14, we can see that the compress value seems contain more noise and blur than original, I believe it is also kind of low pass filter characteristic.

Figure 15 shows the compression size of image.

Figure 14. the Zoom in comparison (Original vs Uncompressed Image)

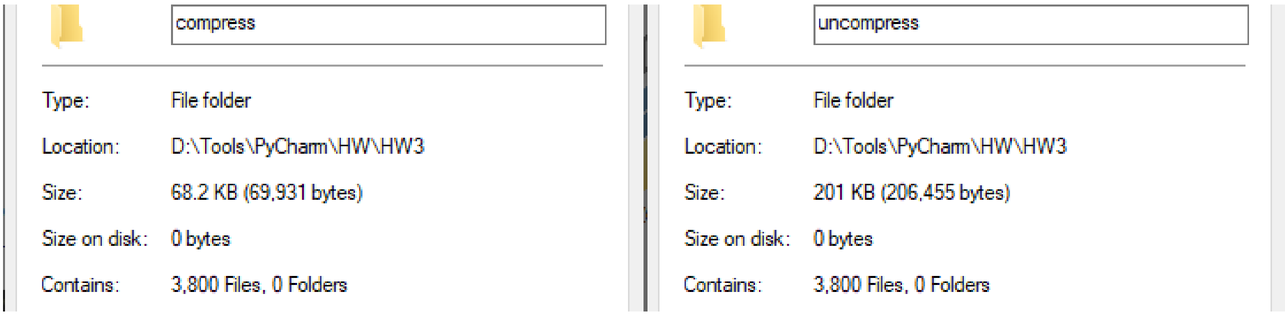

Figure 15. The size of image after compress and after uncompressed

In addition, I also find line drawing picture for experiment as shown on Figure 16. It is become much harder to tell the different between two images, so that I zoom in to the red block as shown in Figure 16 and the zoom in Figure 17 is shown below.

It is still hard to distinguish the difference of this kind of picture, because the pattern of image is very similar and benefit to compress, I believe text images are also the same feature, it will have similar pixels for adjacent neighbors. Kind of High quality and Low storage size image.

Figure 16. Original image vs uncompressed image after compressed

Furthermore, I try to experiment on color image with different kind of subsampling as shown on below Figure 18.

The below image can show that with subsampling, we still cannot easily notice the difference of image, so I zoom in again for observation as Figure 19.

It is obvious can observe that the adjacent pixel have different kind of value as we mentioned in introduction, 4-2-0 have the top-left pixel value of 4 pixels and 4-2-2 have the left pixel value of 2 pixels.

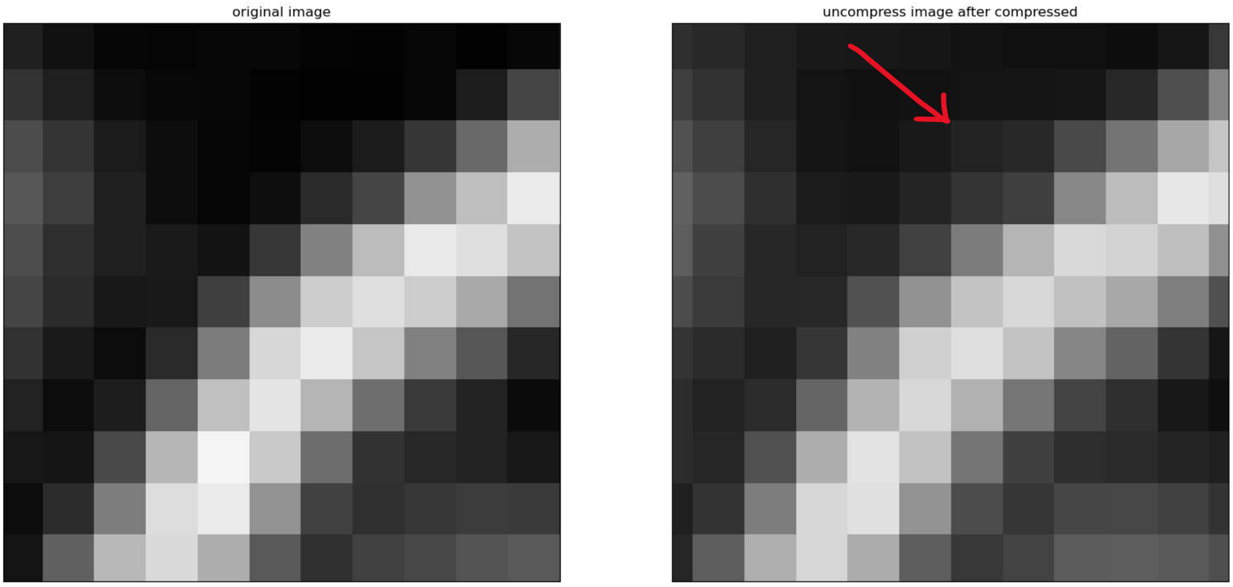

Figure 17. the Zoom in comparison (Original vs Uncompressed Image)

Figure 18. Original image vs uncompressed image after compressed

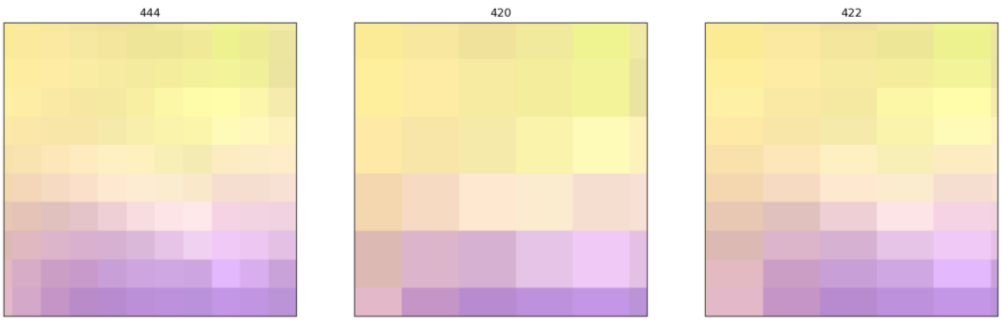

Figure 19. the Zoom in comparison (Original vs Uncompressed Image)

Size:	54.7 KB (56,098 bytes)	Size:	154 KB (157,698 bytes)
Size:	53.8 KB (55,150 bytes)	Size:	152 KB (155,704 bytes)
Size:	54.3 KB (55,650 bytes)	Size:	153 KB (156,941 bytes)

Figure 20. The size of image after compress and after uncompressed 4-4-4, 4-2-0 and 4-2-2 from top to bottom

We can obviously see that the size of compress also reduces if we have more subsampling values, however, the size of picture should be triple of Figure 20, since my coding write file size on 1 channel only and I believe it overwrite data channel by channel.

In below Figure 21 and 22, I try to apply different block size on the same image and same zoom in location with gray, we can see that the smaller block size (4) will have smoother pixel than larger block size (8).

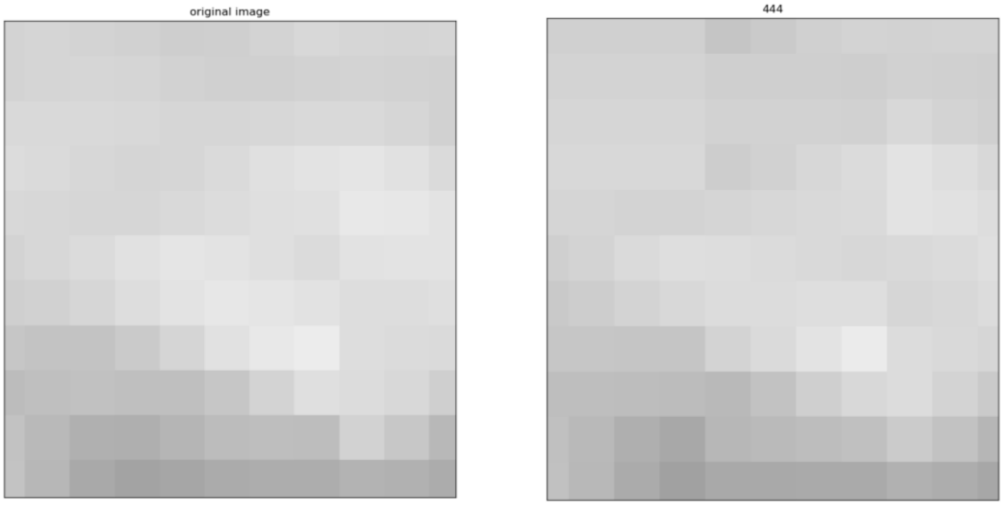

Figure 21. the Zoom in comparison (Original vs Uncompressed Image) with block size 4

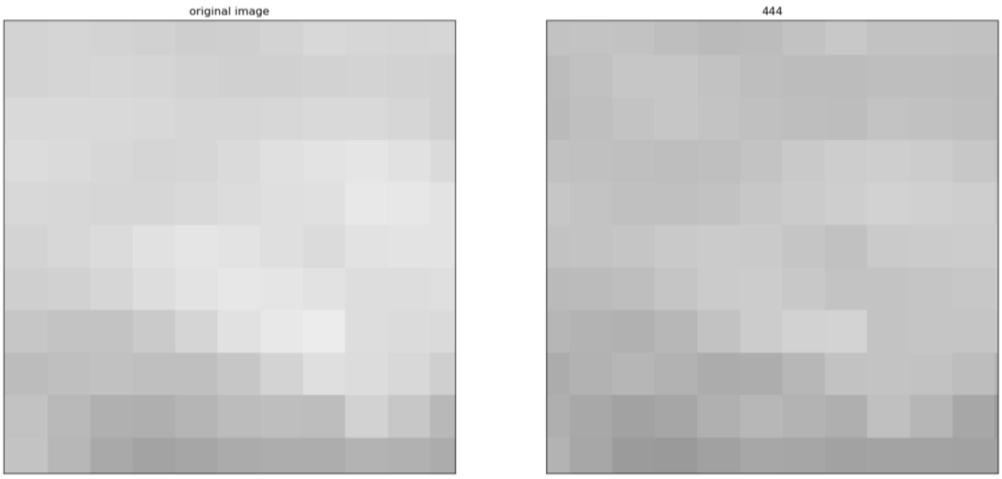

Figure 22. the Zoom in comparison (Original vs Uncompressed Image) with block size 8

Conclusion:

This assignment let me more understand JPEG compression and do experiment with different kind of encoding. Obviously, DCT is a good transformation for benefit the compression, quantization makes the similar pixels compress together and filter out the high frequency component. Then, predictive encoding and run-length encoding compress the remaining data for with Entropy ZigZag sequence. Finally, Huffman encoding further compress the sequence data into an efficiency and small size of data. In addition, color image even can-do subsampling on Chrominance for extra compression. And if the block size is too small, the image will become smoothly, so that the influence of compression or the difference of compression will be easier to obtain by human eyes.

At last, I believe the SNR is not a small value by using original image compared to compress image. However, the image is not easy to distinguish difference by human eyes, that is what we want to have 'High' quality of human used image and lower storage size of it.

Code:

I try to name the function easier to read and add some explanation, please let me know if you curious or you have any questions about this assignment. The code is shown on below:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import itertools
from heapq import heappush, heappop, heapify
from collections import defaultdict
from bitarray import bitarray
import ast
import copy

img = cv2.imread('sipder_man.jpg')  # ,0: gray pic4PR2
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
# img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
print('read image')
# img =[[62,55,55,54,49,48,47,55],  #test matrix
#      [62,57,54,52,48,47,48,53],
#      [61,60,52,49,48,47,49,54],
#      [63,61,60,60,63,65,68,65],
#      [67,67,70,74,79,85,91,92],
#      [82,95,101,106,114,115,112,117],
#      [96,111,115,119,128,128,130,127],
#      [109,121,127,133,139,141,140,133]]

img = np.array(img)
img1 = img.astype(np.float64)
# img1 = img


def color_cvt_matrix():
    transform_matrix = np.array([[0.257, 0.504, 0.098],
                                 [-0.148, -0.291, 0.439],
                                 [0.439, -0.368, -0.071]])
    shift_matrix = np.array([16, 128, 128])
    # transform_matrix = np.array([[0.299, 0.587, 0.114],
    #                              [-0.169, -0.331, 0.500],
    #                              [0.500, -0.419, -0.081]])
    # shift_matrix = np.array([0, 128, 128])
    # transform_matrix = np.array([[0.183, 0.614, 0.062],
    #                              [-0.101, -0.339, 0.439],
    #                              [0.439, -0.399, -0.040]])
    # shift_matrix = np.array([0, 128, 128])
    return transform_matrix, shift_matrix


def rgb2ycbcr(rgb_image):
    """convert rgb into ycbcr"""
    if len(rgb_image.shape) != 3 or rgb_image.shape[2] != 3:
        print("input image is not a rgb image")
    rgb_image = rgb_image.astype(np.float64)
    rgb_image_size = rgb_image.shape
    # RGB to YCbCr matrix
    ycbcr2rgb_matrix, shift_matrix = color_cvt_matrix()
    ycbcr_image = np.zeros(rgb_image.shape)

    # run all pixel with RGB and convert to YCbCr
    for i, j in itertools.product(range(rgb_image_size[0]), range(rgb_im-
age_size[1])):
        ycbcr_image[i, j, :] = np.dot(ycbcr2rgb_matrix, rgb_image[i, j, :]) +
```

```
shift_matrix
        # ycbcr_image[i, j, :] = shift_matrix - np.dot(ycbcr2rgb_matrix, rgb_im-
age[i, j, :])
    # ycbcr_image = ycbcr_image / 255
    return ycbcr_image


def ycbcr2rgb(ycbcr_image):
    """convert ycbcr into rgb"""
    # ycbcr_image = ycbcr_image * 255
    if len(ycbcr_image.shape)!=3 or ycbcr_image.shape[2]!=3:
        print("input image is not a rgb image")
    ycbcr_image = ycbcr_image.astype(np.float64)
    ycbcr_image_size = ycbcr_image.shape
    rgb2ycbcr_matrix, shift_matrix = color_cvt_matrix()

    rgb2ycbcr_matrix_inv = np.linalg.inv(rgb2ycbcr_matrix)
    rgb_image = np.zeros(ycbcr_image_size)

    for i,j in itertools.product(range(ycbcr_image_size[0]),range(ycbcr_im-
age_size[1])):
        rgb_image[i, j, :] = np.dot(rgb2ycbcr_matrix_inv, ycbcr_image[i, j, :])
- np.dot(rgb2ycbcr_matrix_inv, shift_matrix)
        # rgb_image[i, j, :] = np.dot(rgb2ycbcr_matrix_inv, ycbcr_image[i,
j, :]) + np.dot(rgb2ycbcr_matrix_inv,
        #
shift_matrix)
    return rgb_image.astype(np.uint8)


def subsampling_420(image):
    image_after_sampling = image.copy()  # 4:2:0
    image_after_sampling[1::2, :] = image_after_sampling[::2, :]  # 0
    # Vertically, every 2nd element equals to element above itself.
    image_after_sampling[:, 1::2] = image_after_sampling[:, ::2]  # 2
    # Horizontally, every 2nd element equals to the element on its left side.
    # print(image_after_sampling)
    return image_after_sampling


def subsampling_422(image):
    image_after_sampling = image.copy()  # 4:2:2
    # Horizontally, every 2nd element equals to the element on its left side.
    image_after_sampling[:, 1::2] = image_after_sampling[:, ::2]  # 4:2:2
    # print(image_after_sampling)
    return image_after_sampling


def dct_tsf(image_block):
    dct_block = np.zeros(image_block.shape)
    image_block_new = np.zeros(image_block.shape)
    m, n = image_block.shape
    dct_block[0, :] = 1 * np.sqrt(1 / n)
    for u in range(1, m):
        for x in range(n):
            dct_block[u, x] = np.cos(np.pi * u * (x + 0.5) / n) * np.sqrt(2 / n)

    image_block_new = np.dot(dct_block, image_block)
    image_block_new = np.dot(image_block_new, np.transpose(dct_block))
    return image_block_new, dct_block


def idct_tsf(image_block, dct_block):
    image_block_new = np.dot(np.transpose(dct_block), image_block)
```

```
    image_block_new = np.dot(image_block_new, dct_block)
    return image_block_new


def padding(image):
    image_temp = np.array(image) - 128
    image_size = image_temp.shape
    # print('gray size: ', image_size)
    # print('col: ', image_size[0], 'row: ', image_size[1])
    col_mod = image_size[0] % 8
    row_mod = image_size[1] % 8
    pad_col = 8 - col_mod
    pad_row = 8 - row_mod
    image_new = np.array(image)
    # print('col_mod: ', col_mod, 'row_mod: ', row_mod, 'pad_col: ', pad_col)
    if pad_col != 8 or pad_row != 8:
        image_new = cv2.copyMakeBorder(image_temp, 0, pad_col, 0, pad_row,
cv2.BORDER_CONSTANT, 0)
    image_new_size = image_new.shape
    # print('col: ', image_new_size[0], 'row: ', image_new_size[1])
    return image_new


def remove_padding(image_size, image_new):
    print('gray size: ', image_size)
    crop_image = image_new[:image_size[0], :image_size[1]]
    crop_image_size = crop_image.shape
    print('col: ', crop_image_size[0], 'row: ', crop_image_size[1])
    crop_image = np.array(crop_image)  # + 128
    return crop_image


def luminance_matrix(size):
    lum_matrix = [[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]]
    lum_matrix = np.array(lum_matrix)
    lum_matrix_size = lum_matrix.shape
    lum_matrix_update = lum_matrix[:size,:size]
    lum_matrix_update_size = lum_matrix_update.shape
    # print('lum_matrix_update_size: ', lum_matrix_update_size, 'type lum: ',
type(lum_matrix_update))
    qlt = 1
    lum_matrix_update = lum_matrix_update * qlt
    return lum_matrix_update


def quantization(image_block):
    image_block_size = image_block.shape
    lum_matrix = luminance_matrix(image_block_size[0])
    image_block_new = np.divide(image_block, lum_matrix)
    image_block_new_size = image_block_new.shape
    # print('image_block_new_size: ', image_block_new_size, 'type im-
age_block_new: ', type(image_block_new))
    return image_block_new


def inv_quantization(image_block):  # invert quantization
    image_block_size = image_block.shape
```

```python
    lum_matrix = luminance_matrix(image_block_size[0])
    image_block_new = np.multiply(image_block, lum_matrix)
    return image_block_new


def zigzag(block, size):
    n = size
    i, j = 0, 0
    zz_block = []
    zz_loc = []
    while j != (n - 1):
        # print(i, j)
        zz_block.append(block[j][i])
        zz_loc.append([j, i])
        if i == 0 and (j & 1):  # i=0, j=1 -> 1
            j += 1
            continue
        if j == 0 and (i & 1) == 0:  # j=0, i!=1 -> 1
            i += 1
            continue
        if (i ^ j) & 1:  # i!=j -> 1
            i -= 1
            j += 1
            continue
        if (i ^ j) & 1 == 0:  # [(i!=j -> 1) = 0] -> 1
            i += 1
            j -= 1
            continue
    while i != (n - 1) or j != (n - 1):
        # print(i, j)
        zz_block.append(block[j][i])
        zz_loc.append([j, i])
        if i == (n - 1) and (j & 1):
            j += 1
            continue
        if j == (n - 1) and (i & 1) == 0:
            i += 1
            continue
        if (i ^ j) & 1 == 0:
            i += 1
            j -= 1
            continue
        if (i ^ j) & 1:
            i -= 1
            j += 1
            continue
    # print(i, j)
    zz_block.append(block[j][i])
    zz_loc.append([j, i])
    return zz_block, zz_loc


def zagzig(block, size, zz_loc):
    loc_count = 0
    zz_block = np.zeros((size * size))
    zz_block = zz_block.reshape(size, size)
    block = np.array(block)
    block = block.reshape(size, size)
    for i, j in itertools.product(range(size), range(size)):
        zz_block[zz_loc[loc_count][0]][zz_loc[loc_count][1]] = block[i][j]
        loc_count += 1
    # print(i, j)
    return zz_block
```

```python
def predictive_code(arr):
    arr = np.around(arr, decimals=0)
    arr_list = arr.tolist()
    text = arr_list
    # text = ','.join('%s' %id for id in arr)
    print('zero_type: ', type(text), 'len: ', len(text), text)
    count = 0
    # if len(text) > 128:
    for i in range(1,len(arr)):
        # if len(text) > 120:
        if arr[len(arr)-i] == 0:
            count += 1
        else:
            break
    # print(count)
    arr_zero = [count, 0]

    arr_new = arr_list[:(len(arr_list)-count)] + arr_zero
    # print('zero_type: ', type(arr_new), 'len: ', len(arr_new), 'arr: ',
arr_new)
    for j in range(len(arr_new)):
        if arr_new[j] == 0:
            arr_new[j] = 0
        else:
            continue
    # print('zero_after_replace: ', type(arr_new), 'len: ', len(arr_new), 'arr:
', arr_new)
    text = ','.join('%s' %id for id in arr_new)
    # print('after_zero_type: ', type(text), 'len: ', len(text), 'arr: ', text)
    return text


def zero_remove(arr):
    arr = np.around(arr, decimals=0)
    arr_list = arr.tolist()
    text = ','.join('%s' %id for id in arr)
    # print('zero_type: ', type(text), 'len: ', len(text), text)
    count = 0
    if len(text) > 128:
        for i in range(1,len(arr)):
            if len(text) > 120:
                if arr[len(arr)-i] == 0:
                    count += 1
                else:
                    break
    # print(count)
    arr_zero = [count, 0]

    arr_new = arr_list[:(len(arr_list)-count)] + arr_zero
    # print('zero_type: ', type(arr_new), 'len: ', len(arr_new), 'arr: ',
arr_new)
    for j in range(len(arr_new)):
        if arr_new[j] == 0:
            arr_new[j] = 0
        else:
            continue
    # print('zero_after_replace: ', type(arr_new), 'len: ', len(arr_new), 'arr:
', arr_new)
    text = ','.join('%s' %id for id in arr_new)
    # print('after_zero_type: ', type(text), 'len: ', len(text), 'arr: ', text)
    return text
```

```
def inv_zero_add(arr):
    list2 = arr
    zero_add = [0] * int(list2[-2])
    del list2[-2:]
    list2.extend(zero_add)
    # print('list2_after_add_zero: ', type(list2), 'len: ', len(list2), 'recov-
ered: ', list2)
    return list2


def huffman_coding(arr,file_num):
    # print('hum_type: ', type(arr), arr)
    text = zero_remove(arr)
    # text = arr
    # text = ','.join('%s' %id for id in arr)
    # print('hum_text_type: ', type(text), text)
    file_num = str(file_num)
    freq_lib = defaultdict(int)  # generate a default library
    for ch in text:  # count each letter and record into the frequency library
        freq_lib[ch] += 1
    heap = [[fq, [sym, ""]] for sym, fq in freq_lib.items()]  # '' is for enter-
ing the huffman code later
    # print(heap)
    heapify(heap)  # transform the list into a heap tree structure
    # print(heap)
    # print(freq_lib)

    while len(heap) > 1:
        right = heappop(heap)  # heappop - Pop and return the smallest item from
the heap
        # print('right = ', right)
        left = heappop(heap)
        # print('left = ', left)

        for pair in right[1:]:
            pair[1] = '0' + pair[1]  # add zero to all the right note
        for pair in left[1:]:
            pair[1] = '1' + pair[1]  # add one to all the left note
        heappush(heap, [right[0] + left[0]] + right[1:] + left[1:])
        # add values onto the heap. Eg. h = []; heappush(h, (5, 'write code')) -
-> h = [(5, 'write code')]

    huffman_list = right[1:] + left[1:]
    # print(huffman_list)
    huffman_dict = {a[0]: bitarray(str(a[1])) for a in huffman_list}
    # print(huffman_dict)
    encoded_text = bitarray()
    encoded_text.encode(huffman_dict, text)
    # print('encoded_text: ',encoded_text)
    pad = 8 - (len(encoded_text) % 8)
    with open('./compress/compressed_file'+file_num+'.bin', 'wb') as w:
        encoded_text.tofile(w)

    # decode
    decoded_text = bitarray()
    with open('./compress/compressed_file'+file_num+'.bin', 'rb') as r:
        decoded_text.fromfile(r)
    # print('decoded_text: ', decoded_text, 'padding: ', pad)
    if len(decoded_text) != len(encoded_text):
        decoded_text = decoded_text[:-pad]  # remove padding
    decoded_text = decoded_text.decode(huffman_dict)
    decoded_text = ''.join(decoded_text)
    # print('decoded_text_rm_pad: ', decoded_text)
```

```
    with open('./uncompress/uncompress'+file_num+'.bin', 'w') as w:
        w.write(text)
        list1 = ast.literal_eval(decoded_text)
        # print(list1)
        # print('type: ', type(list1))
        list2 = list(list1)
        # print(list2)
        list3 = inv_zero_add(list2)
    #     print('type: ', type(list2))
    # print('before_len: ', len(text), 'type: ', type(text), 'before_text: ',
text)
    # print('after_len: ', len(decoded_text), 'type: ', type(decoded_text) ,
'after_text: ', decoded_text)
    return list3


def image_process(image, stride):
    image_size = image.shape  # padding
    image_after_pad = padding(image)  #
    image_after_pad_size = image_after_pad.shape
    image_new = np.zeros((image_after_pad_size[0],image_after_pad_size[1]))
    image_block = np.zeros((stride,stride))
    image_block = image_block.reshape(stride,stride)
    for i, j in itertools.product(range(0,image_after_pad_size[0],stride),
range(0,image_after_pad_size[1],stride)):  # split by block
        for x, y in itertools.product(range(stride), range(stride)):  # inside
block computation
            image_block[x][y] = image_after_pad[i + x][j + y]
        image_block_after_dct, dct_mtx = dct_tsf(image_block)  # DCT transform
        # image_block_after_dct = cv2.dct(image_block)  # Library DCT
        image_block_after_qtz = quantization(image_block_after_dct)  # Quantiza-
tion
        # print('image_block_after_qtz: ', image_block_after_qtz)
        # print('image_block_after_qtz_type: ', type(image_block_after_qtz))
        image_block_after_zza, pix_loc = zigzag(image_block_after_qtz, stride)
# Zigzag
        file_name = str(i) + '_' + str(j)
        image_block_after_hfc = huffman_coding(image_block_after_zza, file_name)
# Huffman coding
        image_block_after_zzi = zagzig(image_block_after_hfc, stride, pix_loc)
# Zagzig
        image_block_after_iqtz = inv_quantization(image_block_after_zzi)  # in-
verse Quantization
        image_block_after_idct = idct_tsf(image_block_after_iqtz, dct_mtx)  #
iDCT transform
        # image_block_after_idct = cv2.dct(image_block_after_dct)  # Library
iDCT

        for m, n in itertools.product(range(stride), range(stride)):
            image_new[i + m][j + n] = image_block_after_idct[m][n]
    image_new = remove_padding(image_size, image_new)
    print('image_new_max: ', np.max(image_new), 'image_new_min: ', np.min(im-
age_new))
    return image_new


n = 8
# image_after_idct = img
# image_after_idct = np.array(image_after_idct)
# print('type: ', image_after_idct.shape)

# block_after_dct, image_after_idct = image_process(img1, n)

img1_size = img1.shape
```

```
print('img1_size: ', img1_size)
image_after_idct = np.empty(img1_size)
image_cvt2ycc_420 = np.empty(img1_size)
image_cvt2ycc_422 = np.empty(img1_size)

image_cvt2ycc = rgb2ycbcr(img1)  # cvt to ycc
# image_after_idct = image_process(img1, n)

image_cvt2ycc_420[:,:,0] = subsampling_420(image_cvt2ycc[:,:,0])
image_cvt2ycc_420[:,:,1] = subsampling_420(image_cvt2ycc[:,:,1])
image_cvt2ycc_420[:,:,2] = subsampling_420(image_cvt2ycc[:,:,2])

image_cvt2ycc_422[:,:,0] = subsampling_422(image_cvt2ycc[:,:,0])  # [:,:,1]
image_cvt2ycc_422[:,:,1] = subsampling_422(image_cvt2ycc[:,:,1])  # [:,:,1]
image_cvt2ycc_422[:,:,2] = subsampling_422(image_cvt2ycc[:,:,2])  # [:,:,2]

image_after_idct[:,:,0] = image_process(img1[:,:,0], n)
image_after_idct[:,:,1] = image_process(img1[:,:,1], n)
image_after_idct[:,:,2] = image_process(img1[:,:,2], n)

image_cvt2ycc_420_cvt2rgb = ycbcr2rgb(image_cvt2ycc_420)
image_cvt2ycc_422_cvt2rgb = ycbcr2rgb(image_cvt2ycc_422)
# image_cvt2rgb = ycbcr2rgb(image_after_idct)

print('image_cvt2r_max: ', np.max(image_after_idct[:,:,0]), 'image_cvt2rgb_min:
', np.min(image_after_idct[:,:,0]))
print('image_cvt2g_max: ', np.max(image_after_idct[:,:,1]), 'image_cvt2rgb_min:
', np.min(image_after_idct[:,:,1]))
print('image_cvt2b_max: ', np.max(image_after_idct[:,:,2]), 'image_cvt2rgb_min:
', np.min(image_after_idct[:,:,2]))

# image_after_idct = (image_after_idct - np.min(image_after_idct)) / (np.max(im-
age_after_idct) - np.min(image_after_idct))
# image_after_idct[:,:,0] = (image_after_idct[:,:,0] - np.min(image_af-
ter_idct[:,:,0])) / (np.max(image_after_idct[:,:,0]) - np.min(image_af-
ter_idct[:,:,0]))
# image_after_idct[:,:,1] = (image_after_idct[:,:,1] - np.min(image_af-
ter_idct[:,:,1])) / (np.max(image_after_idct[:,:,1]) - np.min(image_af-
ter_idct[:,:,1]))
# image_after_idct[:,:,2] = (image_after_idct[:,:,2] - np.min(image_af-
ter_idct[:,:,2])) / (np.max(image_after_idct[:,:,2]) - np.min(image_af-
ter_idct[:,:,2]))

img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
plt.subplot(121)  # 231
plt.imshow(img, 'gray')
plt.title('original image')
plt.xticks([]), plt.yticks([])

plt.subplot(122)  # 231
plt.imshow(image_after_idct, 'gray')
plt.title('uncompress image after compressed')
plt.xticks([]), plt.yticks([])

print('done')
plt.show()
```

Reference:

https://tzynwang.github.io/2021/how-jpeg-works/
https://yasoob.me/posts/understanding-and-writing-jpeg-decoder-in-python/#jpeg-color-space
https://medium.com/%E9%9B%BB%E8%85%A6%E8%A6%96%E8%A6%BA/%E9%9B%A2%E6%95%A3%E9%A4%98%E5%BC%A6%E8%BD%89%E6%8F%9B-discrete-cosine-transform-dct-%E7%B0%A1%E4%BB%8B-65e426018264
https://www.itread01.com/content/1546761071.html
https://blog.csdn.net/z506820187/article/details/106187863
https://code.activestate.com/recipes/578997-2d-discrete-fourier-transform/
https://pypi.org/project/huffman/
https://github.com/TiongSun/DataCompression/blob/master/Huffman_Coding.ipynb
https://stackoverflow.com/questions/59150761/convert-8x8-matrix-into-flatten-vector-using-zigzag-scan
https://kknews.cc/zh-tw/news/vlgy4jy.html
https://www.itread01.com/content/1550109981.html
https://ithelp.ithome.com.tw/articles/10231357?sc=rss.qu