Image Processing - Programming Assignment #3

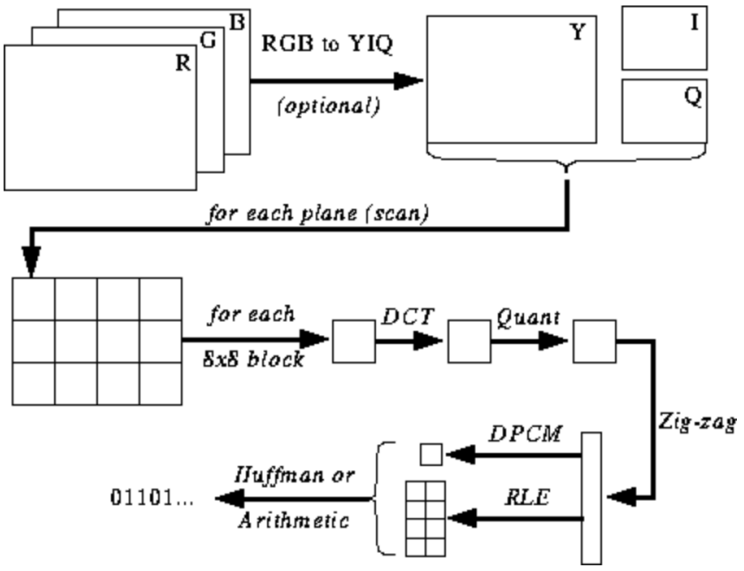
310553056 陳威齊

Introduction / Objectives

This project is about image compression algorithms. However, there are some redundancy in image data, for example, coding redundancy, irrelevant information, spatial and temporal redundancy. We will try to reduce the redundancy property, and make data simpler.

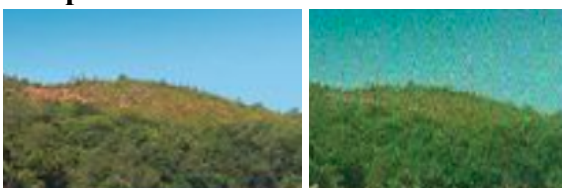
Methods

- Program Language : Python3.8
- Code resource 1 : https://github.com/camilanovaes/jpeg-codec
- Code resource 2 : https://github.com/yasoob/Baseline-JPEG-Decoder
- The major steps in JPEG coding involve :
    - IDCT (Inverse Discrete Cosine Transformation)
    - Quantization
    - Zigzag Scan
    - DPCM on DC component
    - RLE on AC Component
    - Huffman coding
- Work flow :
    -

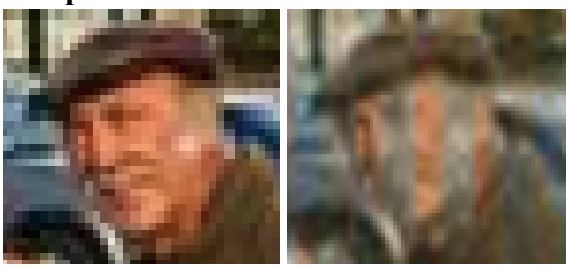


Result

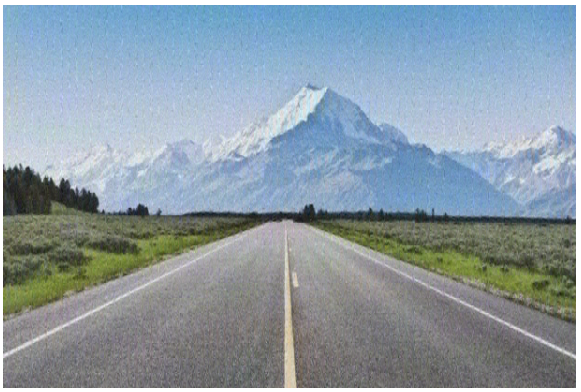
1.

Before 	**After**  Downsampling reduction factor = 2 RMSE: 37.71
After  Downsampling reduction factor = 20 RMSE: 37.71	**Comparison**  壓縮之後(右圖), 遠方山丘的顏色由土色變成綠色

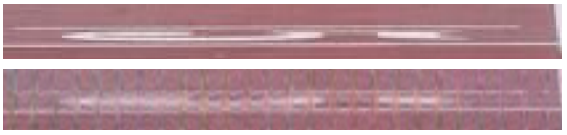
2.

<table>
<tr>
<td>Before
  </td>
<td>After
 
Downsampling reduction factor = 2
RMSE: 18.80</td>
</tr>
<tr>
<td>After
 
Downsampling reduction factor = 20
RMSE: 18.80</td>
<td>Comparison
 
壓縮之後(右圖), 男人臉上出現垂直灰色線條</td>
</tr>
</table>

3.

<table>
<tr>
<td>Before
  </td>
<td>After
 
Downsampling reduction factor = 2
RMSE: 15.80</td>
</tr>
<tr>
<td>After
 
Downsampling reduction factor = 20
RMSE: 15.80</td>
<td>Comparison
 
壓縮之後(右圖), 前面山與後面山的顏色對比變得不明顯</td>
</tr>
</table>

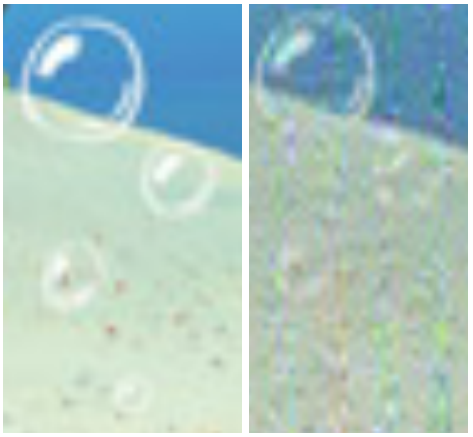
4.

Before	**After**
	 Downsampling reduction factor = 2 RMSE: 17.56
After	**Comparison**
 Downsampling reduction factor = 20 RMSE: 17.56	 壓縮之後(下圖), 跑道號碼變得無法識別

5.

Before	**After**
	 Downsampling reduction factor = 2 RMSE: 24.31

6.

<table>
<tr>
<td>Before
  </td>
<td>After
 
Downsampling reduction factor = 2
RMSE: 30.01</td>
</tr>
<tr>
<td>After
 
Downsampling reduction factor = 20
RMSE: 30.01</td>
<td>Comparison
 
壓縮之後(右圖), 大泡泡下面的三顆泡泡變得不明顯</td>
</tr>
</table>

7.

Before

Y:Cb:Cr = 4:4:4

After - modified zigzag table

```
[[ 0,  1,  5,  6, 14, 15, 27, 28],
 [ 2,  4,  7, 13, 16, 26, 29, 42],
 [ 3,  8, 12, 17, 25, 30, 41, 43],
 [20, 22, 33, 38, 46, 51, 55, 60],
 [21, 34, 37, 47, 50, 56, 59, 61],
 [35, 36, 48, 49, 57, 58, 62, 63],
 [ 9, 11, 18, 24, 31, 40, 44, 53],
 [10, 19, 23, 32, 39, 45, 52, 54]]
```

After - modified zigzag table

After

Y:Cb:Cr = 100:4:4
RMSE = 10.78

`[[12, 19, 26, 33, 40, 48, 41, 34,],`
`[27, 20, 13, 6, 7, 14, 21, 28,],`
`[ 0, 1, 8, 16, 9, 2, 3, 10,],`
`[17, 24, 32, 25, 18, 11, 4, 5,],`
`[35, 42, 49, 56, 57, 50, 43, 36,],`
`[29, 22, 15, 23, 30, 37, 44, 51,],`
`[58, 59, 52, 45, 38, 31, 39, 46,],`
`[53, 60, 61, 54, 47, 55, 62, 63]]`	

After  Y:Cb:Cr = 4:100:4 RMSE = 6.71	**After**  Y:Cb:Cr = 4:4:100 RMSE = 9.31

Discussion

這次的作業可以使用 existing code, 然後要對 code 做參數的修正。

- 第一份 code

  在針對 downsampling reduction factor 去做改變的時候, 發現結果的 RMSE 都一樣, 頗奇怪的。而在調整 chromatic subsampling 中, 想嘗試使用 4:4:4 和 4:2:0 比較結果, 但發現原始碼都寫死了, 所以得不到結果。

- 第二份 code

  對於不同 zigzag 及不同的 chromatic subsampling 比例做修正, 也嘗試對 8x8 的 block 做修正。發現選擇正確的 zigzag 種類對於圖形的壓縮佔有很大的成分, 當路走得不正確 decode 時就會有爆炸或不如預期的結果;在嘗試block的修改時, 發現原始碼都寫死了, 只好一條條的去爬, 但最後沒有很好的結果;而在調整 chromatic subsampling 中, 如果參數按照標準方法 4:4:4 或 4:2:0 很難看出顯著差異, 因此才會以 100 為單位各自去調。在計算 RMSE 時需要將 YCbCr 轉成RGB 之後再轉成灰階, 才可以方便做計算。

Code

第一份

jpeg_codec.py

```
import argparse
import jpeg.encoder, jpeg.decoder
import numpy as np
from PIL import Image

parser = argparse.ArgumentParser(description="JPEG codec")
parser.add_argument('-f', '--filename',
                    help='Image')
args = parser.parse_args()

# Read image
img     = Image.open(args.filename)
src_img = np.asarray(img).astype(np.float64)

img_width, img_height = img.size

# Encoder
encoder    = jpeg.encoder.Encoder(img)
compressed = encoder.process()
header     = compressed['header']

# Decoder
decoder     = jpeg.decoder.Decoder(src_img, header, compressed,
                                   (encoder.width, encoder.height))
decoded_img = decoder.process()

# Print image and calculate the error
img       = np.asarray(decoded_img).copy()
img_final = img[:img_height, :img_width, :]

print(f'Mean Error: {np.mean(src_img -
img_final.astype(np.float64))}')

print(f'RMSE: {np.sqrt(np.mean((src_img -
img_final.astype(np.float64))**2))}')

img_final = Image.fromarray(img_final)
```

```
img_final.show()
```

decoder.py

```
from PIL import Image
from scipy.fftpack import idct
import numpy as np
import jpeg.utils as utils
import skimage.util
import cv2
from cv2 import normalize
from .huffman import H_Encoder, H_Decoder, DC, AC, LUMINANCE,
CHROMINANCE


class Decoder():
    def __init__(self, image, header, compressed, img_info):
        """JPEG Decoder"""
        self.width  = img_info[0]
        self.height = img_info[1]
        self.bits   = compressed['data']
        self.remaining_bits_length = header['remaining_bits_length']
        self.dsls   = header['data_slice_lengths']
        self.image  = image

    def idct(self, blocks):
        """Inverse Discrete Cosine Transform 2D"""
        return idct(idct(blocks, axis=0, norm = 'ortho'), axis=1, norm
= 'ortho')

    def dequantization(self, G, type):
        """Dequantization"""
        if (type == 'l'):
            return(np.multiply(G, utils.Q_y))
        elif (type == 'c'):
            return(np.multiply(G, utils.Q_c))
        else:
            raise ValueError("Type choice %s unknown" %(type))

    def upsampling(self, cb, cr, nrow, ncol):
        """Upsampling function
```

```
        Args:
            cb   :
            cr   :
            nrow :
            ncol :

        """
        up_cb = cv2.resize(cb, dsize=(ncol, nrow))
        up_cr = cv2.resize(cr, dsize=(ncol, nrow))

        return (up_cb, up_cr)

    def entropy_decoding(self, matrix):
        """Entropy decoding

        Args:
            matrix:

        """
        # Rearrange the image components from "zigzag" order
        entropy_mtx = np.zeros((matrix.shape[0], 8, 8))
        for i, block in enumerate(matrix):
            new_block      = [b for _,b in
sorted(zip(utils.zigzag_order, block))]
            new_block      = np.array(new_block).reshape(8,8)
            entropy_mtx[i] = new_block

            if (i != 0):
                entropy_mtx[i][0][0] = matrix[i][0] + matrix[i-1][0]

        return entropy_mtx

    def process(self):

        bits = self.bits.to01()
        remaining_bits_length = self.remaining_bits_length
        dsls = self.dsls  # data_slice_lengths

        # The order of dsls (RGB) is:
        #   LUMINANCE.DC, LUMINANCE.AC, CHROMINANCE.DC, CHROMINANCE.AC
        sliced = {
            LUMINANCE: {
```

```
                DC: bits[:dsls[0]],
                AC: bits[dsls[0]:dsls[0] + dsls[1]]
            },
            CHROMINANCE: {
                DC: bits[dsls[0] + dsls[1]:dsls[0] + dsls[1] +
dsls[2]],
                AC: bits[dsls[0] + dsls[1] + dsls[2]:]
            }
        }
        cb, cr = np.split(H_Decoder(sliced[CHROMINANCE],
CHROMINANCE).decode(), 2)
        y = H_Decoder(sliced[LUMINANCE], LUMINANCE).decode()

        # Dequantization
        dqnt_Y  = self.dequantization(y, 'l')
        dqnt_Cb = self.dequantization(cb, 'c')
        dqnt_Cr = self.dequantization(cr, 'c')

        # Calculate the inverse DCT transform
        idct_Y  = self.idct(dqnt_Y)
        idct_Cb = self.idct(dqnt_Cb)
        idct_Cr = self.idct(dqnt_Cr)

        # Reconstruct image from blocks
        Y  = utils.reconstruct_from_blocks(idct_Y, self.width)
        Cb = utils.reconstruct_from_blocks(idct_Cb, self.width)
        Cr = utils.reconstruct_from_blocks(idct_Cr, self.width)

        # Upsampling Cb and Cr
        Cb, Cr = self.upsampling(Cb, Cr, self.height, self.width)

        img = np.dstack((Y, Cb, Cr)) + 128.0

        # Normalize and convert to uint8
        #
        # The function np.uint8 considers only the lowest byte of the
number,
        # so we need first normalize the image and after that we can
convert to
        # uint8.
        # Ref:
https://stackoverflow.com/questions/46866586/conversion-of-image-type
```

```
-int16-to-uint8
        #
        img = normalize(img, 0, 0, 255, cv2.NORM_MINMAX,
dtype=cv2.CV_8U)
        img = Image.fromarray(img, 'YCbCr').convert('RGB')

        return img

        return np.asarray(img).astype(np.float64)
```

encoder.py

```
from PIL import Image
from scipy.fftpack import dct, idct
import numpy as np
import jpeg.utils as utils
import skimage.util
from bitarray import bitarray, bits2bytes
from .huffman import H_Encoder, H_Decoder, DC, AC, LUMINANCE,
CHROMINANCE

class Encoder():
    def __init__(self, image):
        """JPEG Encoder"""
        self.image  = image
        self.width  = None
        self.height = None

    def dct(self, blocks):
        """Discrete Cosine Transform 2D"""
        return dct(dct(blocks, axis=0, norm = 'ortho'), axis=1, norm =
'ortho')

    def quantization(self, G, type):
        """Quantization"""
        if (type == 'l'):
            return(np.divide(G, utils.Q_y).round().astype(np.float64))
        elif (type == 'c'):
            return(np.divide(G, utils.Q_c).round().astype(np.float64))
        else:
            raise ValueError("Type choice %s unknown" %(type))
```

```
    def downsampling(self, matrix, k=20, type=2):
        """ Downsamplig function

        Args:
            img_ycbcr : Image matrix with 3 channels: Y, Cb and Cr
            nrow      : Number of rows
            ncol      : Number of columns
            k         : Downsampling reduction factor
            type      : Downsampling types. Type 0, represents no
downsampling.
                        Type 1, represents columns reduction, and type
2, rows
                        and columns reduction.

        Returns:
            Downsampling matrix

        """
        if type == 1:
            ds_img = matrix[:,0::k]
        elif type == 2:
            ds_img = matrix[0::k,0::k]
        else:
            ds_img = matrix

        return ds_img

    def entropy_coding(self, matrix):
        """Entropy encoding

        Args:
            matrix:

        """
        # Arrange the image components in a "zigzag" order employing
run-length
        #  encoding (RLE) algorithm.
        entropy_mtx = np.zeros((matrix.shape[0], 64))
        for i, block in enumerate(matrix):
            new_block      = block.reshape([64])[utils.zigzag_order]
            entropy_mtx[i] = new_block
```

```
            if (i != 0):
                entropy_mtx[i][0] = entropy_mtx[i][0] -
entropy_mtx[i-1][0]

        return entropy_mtx

    def process(self):

        # Image width and height
        src_img_height, src_img_width = self.image.size
        print(f'Image: H = {src_img_height}, W = {src_img_width}')

        # Convert to numpy matrix
        src_img_mtx = np.asarray(self.image)

        # Convert 'RGB' to 'YCbCr'
        img_ycbcr = Image.fromarray(src_img_mtx).convert('YCbCr')
        img_ycbcr = np.asarray(img_ycbcr).astype(np.float64)

        # Convert to numpy array
        Y   = img_ycbcr[:,:,0] - 128
        Cb  = img_ycbcr[:,:,1] - 128
        Cr  = img_ycbcr[:,:,2] - 128

        # Apply downsampling to Cb and Cr
        Cb = self.downsampling(Cb, src_img_height, src_img_width)
        Cr = self.downsampling(Cr, src_img_height, src_img_width)

        # Add zero-padding if needed
        Y  = utils.zero_padding(Y)
        Cb = utils.zero_padding(Cb)
        Cr = utils.zero_padding(Cr)

        # Save new size
        self.height, self.width = Y.shape

        # Transform channels into blocks
        Y_bck  = utils.transform_to_block(Y)
        Cb_bck = utils.transform_to_block(Cb)
        Cr_bck = utils.transform_to_block(Cr)

        # Calculate the DCT transform
```

```
        Y_dct  = self.dct(Y_bck)
        Cb_dct = self.dct(Cb_bck)
        Cr_dct = self.dct(Cr_bck)

        # Quantization
        Y_qnt  = self.quantization(Y_dct, 'l')
        Cb_qnt = self.quantization(Cb_dct, 'c')
        Cr_qnt = self.quantization(Cr_dct, 'c')

        # Entropy Encoder
        encoded = {
            LUMINANCE: H_Encoder(Y_qnt, LUMINANCE).encode(),
            CHROMINANCE: H_Encoder(
                np.vstack((Cb_qnt, Cr_qnt)),
                CHROMINANCE
            ).encode()
        }

        # Combine RGB data as binary in the order:
        #   LUMINANCE.DC, LUMINANCE.AC, CHROMINANCE.DC, CHROMINANCE.AC
        order = (encoded[LUMINANCE][DC], encoded[LUMINANCE][AC],
                 encoded[CHROMINANCE][DC], encoded[CHROMINANCE][AC])

        bits = bitarray(''.join(order))

        return {
            'data': bits,
            'header': {
                # Remaining bits length is the fake filled bits for 8
bits as a
                # byte.
                'remaining_bits_length': bits2bytes(len(bits)) * 8 -
len(bits),
                'data_slice_lengths': tuple(len(d) for d in order)
            }
    }
```

huffman.py

```
import collections
import itertools
```

```
from bidict import bidict
import numpy as np

Y, CB, CR = 'y', 'cb', 'cr'

EOB = (0, 0)
ZRL = (15, 0)
DC = 'DC'
AC = 'AC'
LUMINANCE = frozenset({Y})
CHROMINANCE = frozenset({CB, CR})


class H_Encoder:
    def __init__(self, data, layer_type):
        """Create a encoder based on baseline JPEG Huffman table.

        Args:
            data       : The luminance and chrominance data in
following format:
                         {DC: '..010..', AC: '..010..', }
            layer_type : Specify the layer type of data:
                         {LUMINANCE or CHROMINANCE}

        """
        self.data       = data
        self.layer_type = layer_type

        # List containing differential DCs for multiple blocks.
        self._diff_dc       = None

        # List containing run-length-encoding AC pairs for multiple
blocks.
        self._run_length_ac = None

    @property
    def diff_dc(self):
        if self._diff_dc is None:
            self._get_diff_dc()
        return self._diff_dc

    @diff_dc.setter
```

```
    def diff_dc(self, value):
        self._diff_dc = value

    @property
    def run_length_ac(self):
        if self._run_length_ac is None:
            self._get_run_length_ac()
        return self._run_length_ac

    @run_length_ac.setter
    def run_length_ac(self, value):
        self._run_length_ac = value

    def encode(self):
        """Encode differential DC and run-length-encoded AC with
baseline JPEG
        Huffman table based on `self.layer_type`.

        Returns:
            dict : A dictionary containing encoded DC and AC. The
format is:
                   ret = {DC: '01...', AC: '01...'}

        """
        ret = {}
        ret[DC] = ''.join(encode_huffman(v, self.layer_type)
                          for v in self.diff_dc)
        ret[AC] = ''.join(encode_huffman(v, self.layer_type)
                          for v in self.run_length_ac)
        return ret

    def _get_diff_dc(self):
        """Calculate the differential DC of given data."""
        self._diff_dc = tuple(encode_differential(self.data[:, 0, 0]))

    def _get_run_length_ac(self):
        """Calculate the run-length-encoded AC of given data."""
        self._run_length_ac = []
        for block in self.data:
            self._run_length_ac.extend(
                encode_run_length(tuple(iter_zig_zag(block))[1:])
            )
```

```
class H_Decoder:
    def __init__(self, data, layer_type):
        """Create a decoder based on baseline JPEG Huffman table.

        Args:
            data       : A dictionary containing DC and AC bit string
as
                         following format.
                         {DC: '.01..', AC: '.01..'}
            layer_type : Specify the layer type of data:
                         {LUMINANCE or CHROMINANCE}
        """
        self.data       = data
        self.layer_type = layer_type

        # A list containing all DC of blocks.
        self._dc = None

        # A nested 2D list containing all AC of blocks without zig-zag
        # iteration.
        self._ac = None

    def decode(self):
        if (self.layer_type == CHROMINANCE
                and (len(self.dc) % 2 or len(self.ac) % 2)):
            raise ValueError(f'The length of DC chrominance
{len(self.dc)} '
                             f'or AC chrominance {len(self.ac)} cannot
be '
                             'divided by 2 evenly to seperate into Cb
and Cr.')

        shaped = {}
        if len(self.dc) != len(self.ac):
            raise ValueError(f'DC size {len(self.dc)} is not equal to
AC size '
                             f'{len(self.ac)}.')

        shaped = np.array(tuple(inverse_iter_zig_zag((dc, ) + ac,
size=8)
```

```
                                for dc, ac in zip(self.dc, self.ac)))

        return shaped

    @property
    def dc(self):
        if self._dc is None:
            self._get_dc()
        return self._dc

    @property
    def ac(self):
        if self._ac is None:
            self._get_ac()
        return self._ac

    def _get_dc(self):
        self._dc = tuple(decode_differential(decode_huffman(
            self.data[DC],
            DC,
            self.layer_type
        )))

    def _get_ac(self):
        def isplit(iterable, splitter):
            ret = []
            for item in iterable:
                ret.append(item)
                if item == splitter:
                    yield ret
                    ret = []

        self._ac = tuple(decode_run_length(pairs) for pairs in isplit(
            decode_huffman(self.data[AC], AC, self.layer_type),
            EOB
        ))


def encode_huffman(value, layer_type):
    """Encode the Huffman coding of value.

    Args:
```

```
        value      : Differential DC (int) or run-length AC (tuple).
        layer_type : Specify the layer type of value:
                     {LUMINANCE or CHROMINANCE}

    Raises:
        ValueError : When the value is out of the range.

    Returns:
        Huffman encoded bit array.

    """
    def index_2d(table, target):
        for i, row in enumerate(table):
            for j, element in enumerate(row):
                if target == element:
                    return (i, j)
        raise ValueError('Cannot find the target value in the table.')

    if not isinstance(value, collections.Iterable):  # DC
        if value <= -2048 or value >= 2048:
            raise ValueError(
                f'Differential DC {value} should be within [-2047,
2047].'
            )

        size, fixed_code_idx = index_2d(HUFFMAN_CATEGORIES, value)

        if size == 0:
            return HUFFMAN_CATEGORY_CODEWORD[DC][layer_type][size]
        return (HUFFMAN_CATEGORY_CODEWORD[DC][layer_type][size]
                + '{:0{padding}b}'.format(fixed_code_idx,
padding=size))
    else:   # AC
        value = tuple(value)
        if value == EOB or value == ZRL:
            return HUFFMAN_CATEGORY_CODEWORD[AC][layer_type][value]

        run, nonzero = value
        if nonzero == 0 or nonzero <= -1024 or nonzero >= 1024:
            raise ValueError(
                f'AC coefficient nonzero {value} should be within
[-1023, 0) '
```

```
                'or (0, 1023].'
            )

        size, fixed_code_idx = index_2d(HUFFMAN_CATEGORIES, nonzero)
        return (HUFFMAN_CATEGORY_CODEWORD[AC][layer_type][(run, size)]
                + '{:0{padding}b}'.format(fixed_code_idx,
padding=size))


def decode_huffman(bit_seq, dc_ac, layer_type):
    """Decode a bit sequence encoded by JPEG baseline Huffman table.

    Args:
        bit_seq    : The encoded bit sequence.
        dc_ac      : The type of current: {DC or AC}
        layer_type : The layer type of bit sequence: {LUMINANCE or
CHROMINANCE}

    Raises:
        IndexError : When there is not enough bits in bit sequence to
decode
                     DIFF value codeword.
        KeyError   : When not able to find any prefix in current slice
of bit
                     sequence in Huffman table.

    Returns:
        A generator and its item is decoded value which could be an
        integer (differential DC) or a tuple (run-length-encoded AC).
    """

    def diff_value(idx, size):
        if idx >= len(bit_seq) or idx + size > len(bit_seq):
            raise IndexError('There is not enough bits to decode DIFF
value '
                             'codeword.')
        fixed = bit_seq[idx:idx + size]
        return int(fixed, 2)

    current_idx = 0
    while current_idx < len(bit_seq):
        #   1. Consume next 16 bits as `current_slice`.
```

```
        #   2. Try to find the `current_slice` in Huffman table.
        #   3. If found, yield the corresponding key and go to step 4.
        #      Otherwise, remove the last element in `current_slice`
and go to
        #      step 2.
        #   4. Consume next n bits, where n is the category (size) in
returned
        #      key yielded in step 3. Use those info to decode the
data.
        remaining_len = len(bit_seq) - current_idx
        current_slice = bit_seq[
            current_idx:
            current_idx + (16 if remaining_len > 16 else
remaining_len)
        ]
        err_cache = current_slice
        while current_slice:
            if (current_slice in
                    HUFFMAN_CATEGORY_CODEWORD[dc_ac][layer_type].inv):
                key = (HUFFMAN_CATEGORY_CODEWORD[dc_ac][layer_type]
                       .inv[current_slice])
                if dc_ac == DC:  # DC
                    size = key
                    if size == 0:
                        yield 0
                    else:
                        yield HUFFMAN_CATEGORIES[size][diff_value(
                            current_idx + len(current_slice),
                            size
                        )]
                else:  # AC
                    run, size = key
                    if key == EOB or key == ZRL:
                        yield key
                    else:
                        yield (run,
HUFFMAN_CATEGORIES[size][diff_value(
                            current_idx + len(current_slice),
                            size
                        )])

                current_idx += len(current_slice) + size
```

```
                break
            else:
                current_slice = current_slice[:-1]
        else:
            raise KeyError(
                f'Cannot find any prefix of {err_cache} in Huffman
table.'
            )


def encode_differential(seq):
    return (
        (item - seq[idx - 1]) if idx else item
        for idx, item in enumerate(seq)
    )


def decode_differential(seq):
    return itertools.accumulate(seq)


def encode_run_length(seq):
    groups = [(len(tuple(group)), key)
              for key, group in itertools.groupby(seq)]
    ret = []
    borrow = False  # Borrow one pair in the next group whose key is
nonzero.
    if groups[-1][1] == 0:
        del groups[-1]
    for idx, (length, key) in enumerate(groups):
        if borrow == True:
            length -= 1
            borrow = False
        if length == 0:
            continue
        if key == 0:
            # Deal with the case run (0s) more than 16 --> ZRL.
            while length >= 16:
                ret.append(ZRL)
                length -= 16
            ret.append((length, groups[idx + 1][1]))
            borrow = True
```

```
        else:
            ret.extend(((0, key), ) * length)
    return ret + [EOB]


def decode_run_length(seq):
    # Remove the last element as the last created by EOB would always
be a `0`.
    return tuple(item for l, k in seq for item in [0] * l + [k])[:-1]


def iter_zig_zag(data):
    if data.shape[0] != data.shape[1]:
        raise ValueError('The shape of input array should be square.')
    x, y = 0, 0
    for _ in np.nditer(data):
        yield data[y][x]
        if (x + y) % 2 == 1:
            x, y = move_zig_zag_idx(x, y, data.shape[0])
        else:
            y, x = move_zig_zag_idx(y, x, data.shape[0])


def inverse_iter_zig_zag(seq, size=None, fill=0):
    def smallest_square_larger_than(value):
        for ret in itertools.count():
            if ret**2 >= value:
                return ret

    if size is None:
        size = smallest_square_larger_than(len(seq))
    seq = tuple(seq) + (fill, ) * (size**2 - len(seq))
    ret = np.empty((size, size), dtype=int)
    x, y = 0, 0
    for value in seq:
        ret[y][x] = value
        if (x + y) % 2 == 1:
            x, y = move_zig_zag_idx(x, y, size)
        else:
            y, x = move_zig_zag_idx(y, x, size)
    return ret
```

```
def move_zig_zag_idx(i, j, size):
    if j < (size - 1):
        return (max(0, i - 1), j + 1)
    return (i + 1, j)


HUFFMAN_CATEGORIES = (
    (0, ),
    (-1, 1),
    (-3, -2, 2, 3),
    (*range(-7, -4 + 1), *range(4, 7 + 1)),
    (*range(-15, -8 + 1), *range(8, 15 + 1)),
    (*range(-31, -16 + 1), *range(16, 31 + 1)),
    (*range(-63, -32 + 1), *range(32, 63 + 1)),
    (*range(-127, -64 + 1), *range(64, 127 + 1)),
    (*range(-255, -128 + 1), *range(128, 255 + 1)),
    (*range(-511, -256 + 1), *range(256, 511 + 1)),
    (*range(-1023, -512 + 1), *range(512, 1023 + 1)),
    (*range(-2047, -1024 + 1), *range(1024, 2047 + 1)),
    (*range(-4095, -2048 + 1), *range(2048, 4095 + 1)),
    (*range(-8191, -4096 + 1), *range(4096, 8191 + 1)),
    (*range(-16383, -8192 + 1), *range(8192, 16383 + 1)),
    (*range(-32767, -16384 + 1), *range(16384, 32767 + 1))
)

HUFFMAN_CATEGORY_CODEWORD = {
    DC: {
        LUMINANCE: bidict({
            0:  '00',
            1:  '010',
            2:  '011',
            3:  '100',
            4:  '101',
            5:  '110',
            6:  '1110',
            7:  '11110',
            8:  '111110',
            9:  '1111110',
            10: '11111110',
            11: '111111110'
        }),
```

```
        CHROMINANCE: bidict({
            0:  '00',
            1:  '01',
            2:  '10',
            3:  '110',
            4:  '1110',
            5:  '11110',
            6:  '111110',
            7:  '1111110',
            8:  '11111110',
            9:  '111111110',
            10: '1111111110',
            11: '11111111110'
        })
    },
    AC: {
        LUMINANCE: bidict({
            EOB: '1010',  # (0, 0)
            ZRL: '11111111001',  # (F, 0)

            (0, 1):  '00',
            (0, 2):  '01',
            (0, 3):  '100',
            (0, 4):  '1011',
            (0, 5):  '11010',
            (0, 6):  '1111000',
            (0, 7):  '11111000',
            (0, 8):  '1111110110',
            (0, 9):  '1111111110000010',
            (0, 10): '1111111110000011',

            (1, 1):  '1100',
            (1, 2):  '11011',
            (1, 3):  '1111001',
            (1, 4):  '111110110',
            (1, 5):  '11111110110',
            (1, 6):  '1111111110000100',
            (1, 7):  '1111111110000101',
            (1, 8):  '1111111110000110',
            (1, 9):  '1111111110000111',
            (1, 10): '1111111110001000',
```

```
        (2, 1):  '11100',
        (2, 2):  '11111001',
        (2, 3):  '1111110111',
        (2, 4):  '111111110100',
        (2, 5):  '1111111110001001',
        (2, 6):  '1111111110001010',
        (2, 7):  '1111111110001011',
        (2, 8):  '1111111110001100',
        (2, 9):  '1111111110001101',
        (2, 10): '1111111110001110',

        (3, 1):  '111010',
        (3, 2):  '111110111',
        (3, 3):  '111111110101',
        (3, 4):  '1111111110001111',
        (3, 5):  '1111111110010000',
        (3, 6):  '1111111110010001',
        (3, 7):  '1111111110010010',
        (3, 8):  '1111111110010011',
        (3, 9):  '1111111110010100',
        (3, 10): '1111111110010101',

        (4, 1):  '111011',
        (4, 2):  '1111111000',
        (4, 3):  '1111111110010110',
        (4, 4):  '1111111110010111',
        (4, 5):  '1111111110011000',
        (4, 6):  '1111111110011001',
        (4, 7):  '1111111110011010',
        (4, 8):  '1111111110011011',
        (4, 9):  '1111111110011100',
        (4, 10): '1111111110011101',

        (5, 1):  '1111010',
        (5, 2):  '11111110111',
        (5, 3):  '1111111110011110',
        (5, 4):  '1111111110011111',
        (5, 5):  '1111111110100000',
        (5, 6):  '1111111110100001',
        (5, 7):  '1111111110100010',
        (5, 8):  '1111111110100011',
        (5, 9):  '1111111110100100',
```

```
        (5, 10): '1111111110100101',

        (6, 1):  '1111011',
        (6, 2):  '111111110110',
        (6, 3):  '1111111110100110',
        (6, 4):  '1111111110100111',
        (6, 5):  '1111111110101000',
        (6, 6):  '1111111110101001',
        (6, 7):  '1111111110101010',
        (6, 8):  '1111111110101011',
        (6, 9):  '1111111110101100',
        (6, 10): '1111111110101101',

        (7, 1):  '11111010',
        (7, 2):  '111111110111',
        (7, 3):  '1111111110101110',
        (7, 4):  '1111111110101111',
        (7, 5):  '1111111110110000',
        (7, 6):  '1111111110110001',
        (7, 7):  '1111111110110010',
        (7, 8):  '1111111110110011',
        (7, 9):  '1111111110110100',
        (7, 10): '1111111110110101',

        (8, 1):  '111111000',
        (8, 2):  '111111111000000',
        (8, 3):  '1111111110110110',
        (8, 4):  '1111111110110111',
        (8, 5):  '1111111110111000',
        (8, 6):  '1111111110111001',
        (8, 7):  '1111111110111010',
        (8, 8):  '1111111110111011',
        (8, 9):  '1111111110111100',
        (8, 10): '1111111110111101',

        (9, 1):  '111111001',
        (9, 2):  '1111111110111110',
        (9, 3):  '1111111110111111',
        (9, 4):  '1111111111000000',
        (9, 5):  '1111111111000001',
        (9, 6):  '1111111111000010',
        (9, 7):  '1111111111000011',
```

```
        (9, 8):  '1111111111000100',
        (9, 9):  '1111111111000101',
        (9, 10): '1111111111000110',
        # A
        (10, 1):  '111111010',
        (10, 2):  '1111111111000111',
        (10, 3):  '1111111111001000',
        (10, 4):  '1111111111001001',
        (10, 5):  '1111111111001010',
        (10, 6):  '1111111111001011',
        (10, 7):  '1111111111001100',
        (10, 8):  '1111111111001101',
        (10, 9):  '1111111111001110',
        (10, 10): '1111111111001111',
        # B
        (11, 1):  '1111111001',
        (11, 2):  '1111111111010000',
        (11, 3):  '1111111111010001',
        (11, 4):  '1111111111010010',
        (11, 5):  '1111111111010011',
        (11, 6):  '1111111111010100',
        (11, 7):  '1111111111010101',
        (11, 8):  '1111111111010110',
        (11, 9):  '1111111111010111',
        (11, 10): '1111111111011000',
        # C
        (12, 1):  '1111111010',
        (12, 2):  '1111111111011001',
        (12, 3):  '1111111111011010',
        (12, 4):  '1111111111011011',
        (12, 5):  '1111111111011100',
        (12, 6):  '1111111111011101',
        (12, 7):  '1111111111011110',
        (12, 8):  '1111111111011111',
        (12, 9):  '1111111111100000',
        (12, 10): '1111111111100001',
        # D
        (13, 1):  '11111111000',
        (13, 2):  '1111111111100010',
        (13, 3):  '1111111111100011',
        (13, 4):  '1111111111100100',
        (13, 5):  '1111111111100101',
```

```
        (13, 6):  '1111111111100110',
        (13, 7):  '1111111111100111',
        (13, 8):  '1111111111101000',
        (13, 9):  '1111111111101001',
        (13, 10): '1111111111101010',
        # E
        (14, 1):  '1111111111101011',
        (14, 2):  '1111111111101100',
        (14, 3):  '1111111111101101',
        (14, 4):  '1111111111101110',
        (14, 5):  '1111111111101111',
        (14, 6):  '1111111111110000',
        (14, 7):  '1111111111110001',
        (14, 8):  '1111111111110010',
        (14, 9):  '1111111111110011',
        (14, 10): '1111111111110100',
        # F
        (15, 1):  '1111111111110101',
        (15, 2):  '1111111111110110',
        (15, 3):  '1111111111110111',
        (15, 4):  '1111111111111000',
        (15, 5):  '1111111111111001',
        (15, 6):  '1111111111111010',
        (15, 7):  '1111111111111011',
        (15, 8):  '1111111111111100',
        (15, 9):  '1111111111111101',
        (15, 10): '1111111111111110'
    }),
    CHROMINANCE: bidict({
        EOB: '00',  # (0, 0)
        ZRL: '1111111010',  # (F, 0)

        (0, 1):  '01',
        (0, 2):  '100',
        (0, 3):  '1010',
        (0, 4):  '11000',
        (0, 5):  '11001',
        (0, 6):  '111000',
        (0, 7):  '1111000',
        (0, 8):  '111110100',
        (0, 9):  '1111110110',
        (0, 10): '111111110100',
```

```
        (1, 1):  '1011',
        (1, 2):  '111001',
        (1, 3):  '11110110',
        (1, 4):  '111110101',
        (1, 5):  '11111110110',
        (1, 6):  '111111110101',
        (1, 7):  '1111111110001000',
        (1, 8):  '1111111110001001',
        (1, 9):  '1111111110001010',
        (1, 10): '1111111110001011',

        (2, 1):  '11010',
        (2, 2):  '11110111',
        (2, 3):  '1111110111',
        (2, 4):  '111111110110',
        (2, 5):  '111111111000010',
        (2, 6):  '1111111110001100',
        (2, 7):  '1111111110001101',
        (2, 8):  '1111111110001110',
        (2, 9):  '1111111110001111',
        (2, 10): '1111111110010000',

        (3, 1):  '11011',
        (3, 2):  '11111000',
        (3, 3):  '1111111000',
        (3, 4):  '111111110111',
        (3, 5):  '1111111110010001',
        (3, 6):  '1111111110010010',
        (3, 7):  '1111111110010011',
        (3, 8):  '1111111110010100',
        (3, 9):  '1111111110010101',
        (3, 10): '1111111110010110',

        (4, 1):  '111010',
        (4, 2):  '111110110',
        (4, 3):  '1111111110010111',
        (4, 4):  '1111111110011000',
        (4, 5):  '1111111110011001',
        (4, 6):  '1111111110011010',
        (4, 7):  '1111111110011011',
        (4, 8):  '1111111110011100',
```

```
        (4, 9):  '1111111110011101',
        (4, 10): '1111111110011110',

        (5, 1):  '111011',
        (5, 2):  '1111111001',
        (5, 3):  '1111111110011111',
        (5, 4):  '1111111110100000',
        (5, 5):  '1111111110100001',
        (5, 6):  '1111111110100010',
        (5, 7):  '1111111110100011',
        (5, 8):  '1111111110100100',
        (5, 9):  '1111111110100101',
        (5, 10): '1111111110100110',

        (6, 1):  '1111001',
        (6, 2):  '11111110111',
        (6, 3):  '1111111110100111',
        (6, 4):  '1111111110101000',
        (6, 5):  '1111111110101001',
        (6, 6):  '1111111110101010',
        (6, 7):  '1111111110101011',
        (6, 8):  '1111111110101100',
        (6, 9):  '1111111110101101',
        (6, 10): '1111111110101110',

        (7, 1):  '1111010',
        (7, 2):  '111111110000',
        (7, 3):  '1111111110101111',
        (7, 4):  '1111111110110000',
        (7, 5):  '1111111110110001',
        (7, 6):  '1111111110110010',
        (7, 7):  '1111111110110011',
        (7, 8):  '1111111110110100',
        (7, 9):  '1111111110110101',
        (7, 10): '1111111110110110',

        (8, 1):  '11111001',
        (8, 2):  '1111111110110111',
        (8, 3):  '1111111110111000',
        (8, 4):  '1111111110111001',
        (8, 5):  '1111111110111010',
        (8, 6):  '1111111110111011',
```

```
        (8, 7):  '1111111110111100',
        (8, 8):  '1111111110111101',
        (8, 9):  '1111111110111110',
        (8, 10): '1111111110111111',

        (9, 1):  '111110111',
        (9, 2):  '1111111111000000',
        (9, 3):  '1111111111000001',
        (9, 4):  '1111111111000010',
        (9, 5):  '1111111111000011',
        (9, 6):  '1111111111000100',
        (9, 7):  '1111111111000101',
        (9, 8):  '1111111111000110',
        (9, 9):  '1111111111000111',
        (9, 10): '1111111111001000',
        # A
        (10, 1):  '111111000',
        (10, 2):  '1111111111001001',
        (10, 3):  '1111111111001010',
        (10, 4):  '1111111111001011',
        (10, 5):  '1111111111001100',
        (10, 6):  '1111111111001101',
        (10, 7):  '1111111111001110',
        (10, 8):  '1111111111001111',
        (10, 9):  '1111111111010000',
        (10, 10): '1111111111010001',
        # B
        (11, 1):  '111111001',
        (11, 2):  '1111111111010010',
        (11, 3):  '1111111111010011',
        (11, 4):  '1111111111010100',
        (11, 5):  '1111111111010101',
        (11, 6):  '1111111111010110',
        (11, 7):  '1111111111010111',
        (11, 8):  '1111111111011000',
        (11, 9):  '1111111111011001',
        (11, 10): '1111111111011010',
        # C
        (12, 1):  '111111010',
        (12, 2):  '1111111111011011',
        (12, 3):  '1111111111011100',
        (12, 4):  '1111111111011101',
```

```
            (12, 5):  '1111111111011110',
            (12, 6):  '1111111111011111',
            (12, 7):  '1111111111100000',
            (12, 8):  '1111111111100001',
            (12, 9):  '1111111111100010',
            (12, 10): '1111111111100011',
            # D
            (13, 1):  '11111111001',
            (13, 2):  '1111111111100100',
            (13, 3):  '1111111111100101',
            (13, 4):  '1111111111100110',
            (13, 5):  '1111111111100111',
            (13, 6):  '1111111111101000',
            (13, 7):  '1111111111101001',
            (13, 8):  '1111111111101010',
            (13, 9):  '1111111111101011',
            (13, 10): '1111111111101100',
            # E
            (14, 1):  '11111111100000',
            (14, 2):  '1111111111101101',
            (14, 3):  '1111111111101110',
            (14, 4):  '1111111111101111',
            (14, 5):  '1111111111110000',
            (14, 6):  '1111111111110001',
            (14, 7):  '1111111111110010',
            (14, 8):  '1111111111110011',
            (14, 9):  '1111111111110100',
            (14, 10): '1111111111110101',
            # F
            (15, 1):  '111111111000011',
            (15, 2):  '1111111111110110',
            (15, 3):  '1111111111110111',
            (15, 4):  '1111111111111000',
            (15, 5):  '1111111111111001',
            (15, 6):  '1111111111111010',
            (15, 7):  '1111111111111011',
            (15, 8):  '1111111111111100',
            (15, 9):  '1111111111111101',
            (15, 10): '1111111111111110'
        })
    }
}
```

utils.py

```
import numpy as np


# Quantization tables
Q_y = 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]])

Q_c = np.array([[17, 18, 24, 47, 99, 99, 99, 99],
                [18, 21, 26, 66, 99, 99, 99, 99],
                [24, 26, 56, 99, 99, 99, 99, 99],
                [47, 66, 99, 99, 99, 99, 99, 99],
                [99, 99, 99, 99, 99, 99, 99, 99],
                [99, 99, 99, 99, 99, 99, 99, 99],
                [99, 99, 99, 99, 99, 99, 99, 99],
                [99, 99, 99, 99, 99, 99, 99, 99]])

zigzag_order =
np.array([0,1,8,16,9,2,3,10,17,24,32,25,18,11,4,5,12,19,26,33,

40,48,41,34,27,20,13,6,7,14,21,28,35,42,49,56,57,50,

43,36,29,22,15,23,30,37,44,51,58,59,52,45,38,31,39,
                        46,53,60,61,54,47,55,62,63])

# General Functions
def reconstruct_from_blocks(blocks, img_width):
   """ Inverse Discrete Cosine Transform 2D

   Args:
       blocks     :
```

```
        img_width  :
        img_height :
        H          :
        W          :

    """

    total_lines = []
    N_blocks    = int(img_width / 8)

    for n in range(0, len(blocks) - N_blocks + 1, N_blocks):
        res = np.concatenate(blocks[n : n + N_blocks], axis=1)
        total_lines.append(res)

    return np.concatenate(total_lines)

def transform_to_block(image):
    """ Transform image into N 8x8 blocks

    Args:
        image : n-dimensional array

    """

    img_w, img_h = image.shape
    blocks = []
    for i in range(0, img_w, 8):
        for j in range(0, img_h, 8):
            blocks.append(image[i:i+8,j:j+8])

    return blocks

def zero_padding(matrix):
    """ Add zero-padding

    Args:
        matrix :

    """
    ncol, nrow = matrix.shape[0], matrix.shape[1]

    if (ncol % 8 != 0):
```

```
        img_width = ncol // 8 * 8 + 8
    else:
        img_width = ncol

    if (nrow % 8 != 0):
        img_height = nrow // 8 * 8 + 8
    else:
        img_height = nrow

    # Copy data to new matrix
    new_mtx = np.zeros((img_width, img_height), dtype=np.float64)
    for y in range(ncol):
        for x in range(nrow):
            new_mtx[y][x] = matrix[y][x]

    return new_mtx
```

第二份 **code**

decode.py

```
from struct import unpack
import math

marker_mapping = {
    0xFFD8: "Start of Image",
    0xFFE0: "Application Default Header",
    0xFFDB: "Quantization Table",
    0xFFC0: "Start of Frame",
    0xFFC4: "Huffman Table",
    0xFFDA: "Start of Scan",
    0xFFD9: "End of Image",
}


def PrintMatrix(m):
    """
    A convenience function for printing matrices
    """
    for j in range(8):
        print("|", end="")
```

```
        for i in range(8):
            print("%d  |" % m[i + j * 8], end="\t")
        print()
    print()


def Clamp(col):
    """
    Makes sure col is between 0 and 255.
    """
    col = 255 if col > 255 else col
    col = 0 if col < 0 else col
    return int(col)


def ColorConversion(Y, Cr, Cb):
    """
    Converts Y, Cr and Cb to RGB color space
    """
    R = Cr * (2 - 2 * 0.299) + Y
    B = Cb * (2 - 2 * 0.114) + Y
    G = (Y - 0.114 * B - 0.299 * R) / 0.587
    return (Clamp(R + 128), Clamp(G + 128), Clamp(B + 128))


def DrawMatrix(x, y, matL, matCb, matCr):
    """
    Loops over a single 8x8 MCU and draws it on Tkinter canvas
    """
    for yy in range(8):
        for xx in range(8):
            c = "#%02x%02x%02x" % ColorConversion(
                matL[yy][xx], matCb[yy][xx], matCr[yy][xx]
            )
            x1, y1 = (x * 8 + xx) * 2, (y * 8 + yy) * 2
            x2, y2 = (x * 8 + (xx + 1)) * 2, (y * 8 + (yy + 1)) * 2
            w.create_rectangle(x1, y1, x2, y2, fill=c, outline=c)


def RemoveFF00(data):
    """
    Removes 0x00 after 0xff in the image scan section of JPEG
```

```
    """
    datapro = []
    i = 0
    while True:
        b, bnext = unpack("BB", data[i : i + 2])
        if b == 0xFF:
            if bnext != 0:
                break
            datapro.append(data[i])
            i += 2
        else:
            datapro.append(data[i])
            i += 1
    return datapro, i


def GetArray(type, l, length):
    """
    A convenience function for unpacking an array from bitstream
    """
    s = ""
    for i in range(length):
        s = s + type
    return list(unpack(s, l[:length]))


def DecodeNumber(code, bits):
    l = 2 ** (code - 1)
    if bits >= l:
        return bits
    else:
        return bits - (2 * l - 1)


class IDCT:
    """
    An inverse Discrete Cosine Transformation Class
    """

    def __init__(self):
        self.base = [0] * 64
        self.zigzag = [
```

```
            [0, 1, 5, 6, 14, 15, 27, 28],
            [2, 4, 7, 13, 16, 26, 29, 42],
            [3, 8, 12, 17, 25, 30, 41, 43],
            [9, 11, 18, 24, 31, 40, 44, 53],
            [10, 19, 23, 32, 39, 45, 52, 54],
            [20, 22, 33, 38, 46, 51, 55, 60],
            [21, 34, 37, 47, 50, 56, 59, 61],
            [35, 36, 48, 49, 57, 58, 62, 63],
        ]
        self.idct_precision = 8
        self.idct_table = [
            [
                (self.NormCoeff(u) * math.cos(((2.0 * x + 1.0) * u *
math.pi) / 16.0))
                for x in range(self.idct_precision)
            ]
            for u in range(self.idct_precision)
        ]

    def NormCoeff(self, n):
        if n == 0:
            return 1.0 / math.sqrt(2.0)
        else:
            return 1.0

    def rearrange_using_zigzag(self):
        for x in range(8):
            for y in range(8):
                self.zigzag[x][y] = self.base[self.zigzag[x][y]]
        return self.zigzag

    def perform_IDCT(self):
        out = [list(range(8)) for i in range(8)]

        for x in range(8):
            for y in range(8):
                local_sum = 0
                for u in range(self.idct_precision):
                    for v in range(self.idct_precision):
                        local_sum += (
                            self.zigzag[v][u]
                            * self.idct_table[u][x]
```

```
                            * self.idct_table[v][y]
                        )
                out[y][x] = local_sum // 4
        self.base = out


class HuffmanTable:
    """
    A Huffman Table class
    """

    def __init__(self):
        self.root = []
        self.elements = []

    def BitsFromLengths(self, root, element, pos):
        if isinstance(root, list):
            if pos == 0:
                if len(root) < 2:
                    root.append(element)
                    return True
                return False
            for i in [0, 1]:
                if len(root) == i:
                    root.append([])
                if self.BitsFromLengths(root[i], element, pos - 1) ==
True:
                    return True
        return False

    def GetHuffmanBits(self, lengths, elements):
        self.elements = elements
        ii = 0
        for i in range(len(lengths)):
            for j in range(lengths[i]):
                self.BitsFromLengths(self.root, elements[ii], i)
                ii += 1

    def Find(self, st):
        r = self.root
        while isinstance(r, list):
            print(r)
```

```
            r = r[st.GetBit()]
        return r

    def GetCode(self, st):
        while True:
            res = self.Find(st)
            if res == 0:
                return 0
            elif res != -1:
                return res


class Stream:
    """
    A bit stream class with convenience methods
    """

    def __init__(self, data):
        self.data = data
        self.pos = 0

    def GetBit(self):
        b = self.data[self.pos >> 3]
        s = 7 - (self.pos & 0x7)
        self.pos += 1
        return (b >> s) & 1

    def GetBitN(self, l):
        val = 0
        for _ in range(l):
            val = val * 2 + self.GetBit()
        return val

    def len(self):
        return len(self.data)


class JPEG:
    """
    JPEG class for decoding a baseline encoded JPEG image
    """
```

```
    def __init__(self, image_file):
        self.huffman_tables = {}
        self.quant = {}
        self.quantMapping = []
        with open(image_file, "rb") as f:
            self.img_data = f.read()

    def DefineQuantizationTables(self, data):
        (hdr,) = unpack("B", data[0:1])
        self.quant[hdr] = GetArray("B", data[1 : 1 + 64], 64)
        data = data[65:]

    def BuildMatrix(self, st, idx, quant, olddccoeff):
        i = IDCT()

        code = self.huffman_tables[0 + idx].GetCode(st)
        bits = st.GetBitN(code)
        dccoeff = DecodeNumber(code, bits) + olddccoeff

        i.base[0] = (dccoeff) * quant[0]
        l = 1
        while l < 64:
            code = self.huffman_tables[16 + idx].GetCode(st)
            if code == 0:
                break

            # The first part of the AC key_len
            # is the number of leading zeros
            if code > 15:
                l += code >> 4
                code = code & 0x0F

            bits = st.GetBitN(code)

            if l < 64:
                coeff = DecodeNumber(code, bits)
                i.base[l] = coeff * quant[l]
                l += 1

        i.rearrange_using_zigzag()
        i.perform_IDCT()
```

```
        return i, dccoeff

    def StartOfScan(self, data, hdrlen):
        data, lenchunk = RemoveFF00(data[hdrlen:])

        st = Stream(data)
        oldlumdccoeff, oldCbdccoeff, oldCrdccoeff = 0, 0, 0
        for y in range(self.height // 8):
            for x in range(self.width // 8):
                matL, oldlumdccoeff = self.BuildMatrix(
                    st, 0, self.quant[self.quantMapping[0]],
oldlumdccoeff
                )
                matCr, oldCrdccoeff = self.BuildMatrix(
                    st, 1, self.quant[self.quantMapping[1]],
oldCrdccoeff
                )
                matCb, oldCbdccoeff = self.BuildMatrix(
                    st, 1, self.quant[self.quantMapping[2]],
oldCbdccoeff
                )
                DrawMatrix(x, y, matL.base, matCb.base, matCr.base)

        return lenchunk + hdrlen

    def BaselineDCT(self, data):
        hdr, self.height, self.width, components = unpack(">BHHB",
data[0:6])
        print("size %ix%i" % (self.width,  self.height))

        for i in range(components):
            id, samp, QtbId = unpack("BBB", data[6 + i * 3 : 9 + i *
3])
            self.quantMapping.append(QtbId)

    def decodeHuffman(self, data):
        offset = 0
        (header,) = unpack("B", data[offset : offset + 1])
        print(header, header & 0x0F, (header >> 4) & 0x0F)
        offset += 1

        lengths = GetArray("B", data[offset : offset + 16], 16)
```

```
        offset += 16

        elements = []
        for i in lengths:
            elements += GetArray("B", data[offset : offset + i], i)
            offset += i

        hf = HuffmanTable()
        hf.GetHuffmanBits(lengths, elements)
        self.huffman_tables[header] = hf
        data = data[offset:]

    def decode(self):
        data = self.img_data
        while True:
            (marker,) = unpack(">H", data[0:2])
            print(marker_mapping.get(marker))
            if marker == 0xFFD8:
                data = data[2:]
            elif marker == 0xFFD9:
                return
            else:
                (len_chunk,) = unpack(">H", data[2:4])
                len_chunk += 2
                chunk = data[4:len_chunk]
                if marker == 0xFFC4:
                    self.decodeHuffman(chunk)
                elif marker == 0xFFDB:
                    self.DefineQuantizationTables(chunk)
                elif marker == 0xFFC0:
                    self.BaselineDCT(chunk)
                elif marker == 0xFFDA:
                    len_chunk = self.StartOfScan(data, len_chunk)
                data = data[len_chunk:]
            if len(data) == 0:
                break


if __name__ == "__main__":
    from tkinter import Tk, Canvas, mainloop

    master = Tk()
```

```
w = Canvas(master, width=1600, height=600)
w.pack()
img = JPEG("pets.jpg")
img.decode()
mainloop()
```