

## Introduction:

I am currently planning to use a motorcycle to travel around Taiwan, and some seniors recommended the four-pole roundabout in Taiwan. During this round of the island, I hope to help raise funds for the Hsinchu Family Support Center([新竹家扶中心](#)), so that children can have a better quality of life and education.



**Task: The objective of this assignment is for you to experiment with the various components of a JPEG codec. Your implementation should include both an encoder and a decoder.**

A total of four-pole roundabout in Taiwan images were used, and jpeg Encoder 、Decoder and Mse methods were used, namely:

### 1. Encoder







- A. Image read & resize & Block
- B. Quantized
- C. ZigZag
- D. Run length encode
- E. Huffman
- F. Store

### 2. Decoder

- A. Read encoded file & De-Huffman & Run length decode
- B. ZigZag
- C. Quantized

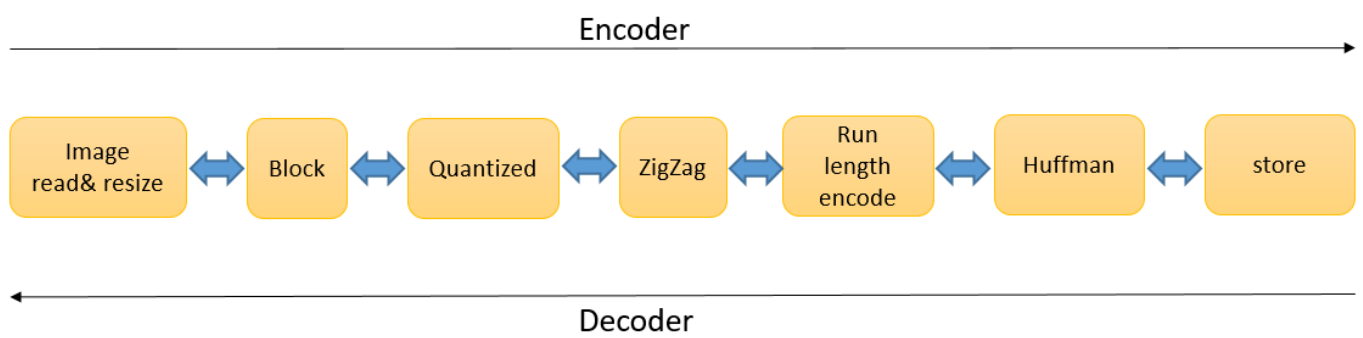
### 3. Mse

**Result:**

	Original image	Decode image	Mse
North			381.27
East			362.14
South			225.25

West			549.42
------	---	--	--------

The main steps:



## Code:

### 1. Encoder

#### A. Image read & resize & Block

```
# parameter
season = "west"

input_file = "../img/{}.jpg".format(season)
output_file = "../img/{}_encoder".format(season)

image = Image.open(input_file)
# resize image width and height to multiples of 8
L, H = image.size
image = image.resize((512,512))
image.save("../img/{}_original.jpg".format(season))
ycbcr = image.convert('YCbCr')

npmat = np.array(ycbcr, dtype=np.uint8)

rows, cols = npmat.shape[0], npmat.shape[1]

# block size: 8x8
if rows % 8 == cols % 8 == 0:
    blocks_count = rows // 8 * cols // 8
    print("blocks_count=", blocks_count)
else:
    raise ValueError(("the width and height of the image "
                      "should both be multiples of 8"))

# dc is the top-left cell of the block, ac are all the other cells
dc = np.empty((blocks_count, 3), dtype=np.int32)
ac = np.empty((blocks_count, 63, 3), dtype=np.int32)
```

#### B. Quantized

```
def quantize(block, component):
    q = load_quantization_table(component)
    return (block / q).round().astype(np.int32)
```

#### C. ZigZag

```
def block2zigzag(block):
    return np.array([block[point] for point in zigzag_points(*block.shape)])
```

#### D. Run length encode

```

def runLengthEncode(arr):
    # determine where the sequence is ending prematurely
    last_nonzero = -1
    for i, elem in enumerate(arr):
        if elem != 0:
            last_nonzero = i

    # each symbol is a (RUNLENGTH, SIZE) tuple
    symbols = []

    # values are binary representations of array elements using SIZE bits
    values = []

    run_length = 0

    for i, elem in enumerate(arr):
        if i > last_nonzero:
            symbols.append((0, 0))
            values.append(int_to_binstr(0))
            break
        elif elem == 0 and run_length < 15:
            run_length += 1
        else:
            size = bits_required(elem)
            symbols.append((run_length, size))
            values.append(int_to_binstr(elem))
            run_length = 0
    return symbols, values

```

#### E. Huffman



```

from queue import PriorityQueue

class HuffmanTree:
    class __Node:
        def __init__(self, value, freq, left_child, right_child):
            self.value = value
            self.freq = freq
            self.left_child = left_child
            self.right_child = right_child
        @classmethod
        def init_leaf(self, value, freq):
            return self(value, freq, None, None)
        @classmethod
        def init_node(self, left_child, right_child):
            freq = left_child.freq + right_child.freq
            return self(None, freq, left_child, right_child)
        def is_leaf(self):
            return self.value is not None
        def __eq__(self, other):
            stup = self.value, self.freq, self.left_child, self.right_child
            otup = other.value, other.freq, other.left_child, other.right_child
            return stup == otup
        def __neq__(self, other):
            return not (self == other)
        def __lt__(self, other):
            return self.freq < other.freq
        def __le__(self, other):
            return self.freq < other.freq or self.freq == other.freq
        def __gt__(self, other):
            return not (self <= other)
        def __ge__(self, other):
            return not (self < other)
    def __init__(self, arr):
        q = PriorityQueue()
        # calculate frequencies and insert them into a priority queue
        for val, freq in self.__calc_freq(arr).items():
            q.put(self.__Node.init_leaf(val, freq))
        while q.qsize() >= 2:
            u = q.get()
            v = q.get()
            q.put(self.__Node.init_node(u, v))
        self.__root = q.get()
        # dictionaries to store huffman table
        self.__value_to_bitstring = dict()
    def value_to_bitstring_table(self):
        if len(self.__value_to_bitstring.keys()) == 0:
            self.__create_huffman_table()
        return self.__value_to_bitstring
    def __create_huffman_table(self):
        def tree_traverse(current_node, bitstring=''):
            if current_node is None:
                return
            if current_node.is_leaf():
                self.__value_to_bitstring[current_node.value] = bitstring
                return
            tree_traverse(current_node.left_child, bitstring + '0')
            tree_traverse(current_node.right_child, bitstring + '1')
        tree_traverse(self.__root)
    def __calc_freq(self, arr):
        freq_dict = dict()
        for elem in arr:
            if elem in freq_dict:
                freq_dict[elem] += 1
            else:
                freq_dict[elem] = 1
        return freq_dict

```

F. Store

```

def write_to_file(filepath, dc, ac, blocks_count, tables):
    try:
        f = open(filepath, 'w')
    except FileNotFoundError as e:
        raise FileNotFoundError(
            "No such directory: {}".format(
                os.path.dirname(filepath))) from e

    for table_name in ['dc_y', 'ac_y', 'dc_c', 'ac_c']:

        # 16 bits for 'table_size'
        f.write(uint_to_binstr(len(tables[table_name]), 16))

        for key, value in tables[table_name].items():
            if table_name in {'dc_y', 'dc_c'}:
                # 4 bits for the 'category'
                # 4 bits for 'code_length'
                # 'code_length' bits for 'huffman_code'
                f.write(uint_to_binstr(key, 4))
                f.write(uint_to_binstr(len(value), 4))
                f.write(value)
            else:
                # 4 bits for 'run_length'
                # 4 bits for 'size'
                # 8 bits for 'code_length'
                # 'code_length' bits for 'huffman_code'
                f.write(uint_to_binstr(key[0], 4))
                f.write(uint_to_binstr(key[1], 4))
                f.write(uint_to_binstr(len(value), 8))
                f.write(value)

        # 32 bits for 'blocks_count'
        f.write(uint_to_binstr(blocks_count, 32))

    for b in range(blocks_count):
        for c in range(3):
            category = bits_required(dc[b, c])
            symbols, values = runLengthEncode(ac[b, :, c])

            dc_table = tables['dc_y'] if c == 0 else tables['dc_c']
            ac_table = tables['ac_y'] if c == 0 else tables['ac_c']

            f.write(dc_table[category])
            f.write(int_to_binstr(dc[b, c]))

            for i in range(len(symbols)):
                f.write(ac_table[tuple(symbols[i])])
                f.write(values[i])

    f.close()

```

## 2. Decoder

### A. Read encoded file & De-Huffman & Run length decode

```
def readImageFile(filepath): # De_huffman + runlenth_decode
    reader = jpgReader(filepath)

    tables = dict()
    for table_name in ['dc_y', 'ac_y', 'dc_c', 'ac_c']:
        if 'dc' in table_name:
            tables[table_name] = reader.read_dc_table()
        else:
            tables[table_name] = reader.read_ac_table()

    blocks_count = reader.read_blocks_count()

    dc = np.empty((blocks_count, 3), dtype=np.int32)
    ac = np.empty((blocks_count, 63, 3), dtype=np.int32)

    for block_index in range(blocks_count):
        for component in range(3):
            dc_table = tables['dc_y'] if component == 0 else tables['dc_c']
            ac_table = tables['ac_y'] if component == 0 else tables['ac_c']

            category = reader.read_huffman_code(dc_table)
            dc[block_index, component] = reader.read_int(category)

            cells_count = 0

            # TODO: try to make reading AC coefficients better
            while cells_count < 63:
                run_length, size = reader.read_huffman_code(ac_table)

                if (run_length, size) == (0, 0):
                    while cells_count < 63:
                        ac[block_index, cells_count, component] = 0
                        cells_count += 1
                else:
                    for i in range(run_length):
                        ac[block_index, cells_count, component] = 0
                        cells_count += 1
                    if size == 0:
                        ac[block_index, cells_count, component] = 0
                    else:
                        value = reader.read_int(size)
                        ac[block_index, cells_count, component] = value
                        cells_count += 1

    return dc, ac, tables, blocks_count
```

### B. ZigZag

```
def zigzag_to_block(zigzag):
    # assuming that the width and the height of the block are equal
    rows = cols = int(math.sqrt(len(zigzag)))

    if rows * cols != len(zigzag):
        raise ValueError("length of zigzag should be a perfect square")

    block = np.empty((rows, cols), np.int32)

    for i, point in enumerate(zigzag_points(rows, cols)):
        block[point] = zigzag[i]

    return block
```

### C. Quantized



```
def dequantize(block, component):
    q = load_quantization_table(component)
    return block * q
```

### 3. Mse

```
def mse(imageA, imageB):
    # the 'Mean Squared Error' between the two images is the
    # sum of the squared difference between the two images;
    # NOTE: the two images must have the same dimension
    err = np.sum((imageA.astype("float") - imageB.astype("float")) ** 2)
    err /= float(imageA.shape[0] * imageA.shape[1])

    # return the MSE, the lower the error, the more "similar" the two images are
    return err

def compare_images(imageA, imageB, season):
    # compute the mean squared error and structural similarity
    # index for the images
    m = mse(imageA, imageB)
    # initialize the figure
    fig = plt.figure("Images")
    images = ("{}_Original".format(season), original), ("{}_Decoder".format(season), decoder)

    # Loop over the images
    for (i, (name, image)) in enumerate(images):
        # show the image
        ax = fig.add_subplot(1, 2, i + 1)
        ax.set_title(name)
        plt.imshow(image, cmap = plt.cm.gray)

        plt.axis("off")

    # show the figure
    plt.show()
    print("MSE: %.2f" % (m))
```