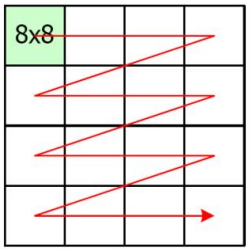
Encoding

```
img = cv2.imread('lena.png')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
cv2.imshow('original', img)
cv2.waitKey(0)
img = np.float32(img)
```

First, the image is converted to grayscale.

```
block = img[cur_block_i:cur_block_i + 8, cur_block_j:cur_block_j + 8]
```

```
if cur_block_j + 8 < dim:
    cur_block_j += 8
elif cur_block_i + 8 < dim:
    cur_block_i += 8
    cur_block_j = 0
else:
    break
```



Block level scan

Get the current 8x8 block of the image.

(cur_block_i, cur_block_j) indicates the (x,y) coordinate of the top left corner of the current processed block.

dim indicates the dimension of the image, 512 (since the image is 512 x 512).

We go through each block of the image row by row.

```
block = block - 128
dct = cv2.dct(block)
dct = np.round(dct)
dct = np.int16(dct)
q_dct = np.round(np.divide(dct,QUANT_MATR)) #quantization
```

```
QUANT_MATR = 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]
    ])
```

Convert the range of the values in the block from [0,255] to [-128,127].

Compute the DCT the block then round them to integers.

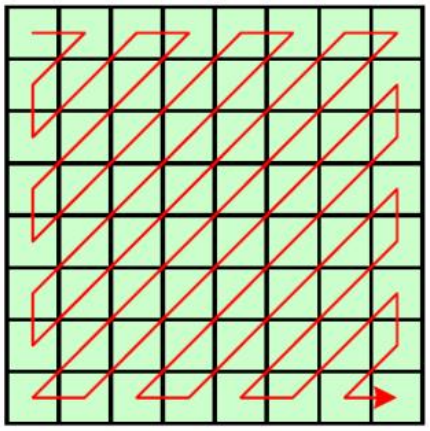
Quantize the DCT coefficients by dividing each coefficient with the value in the same index in the quantization matrix then round them to integers. The quantization matrix is defined above by QUANT_MATR.

```
#run length encoding
cur_i = 0
cur_j = 0
Direction = False # diagonal direction
temp = [] # current block encoding

while True:

    if cur_i == 0 or cur_i == 7:
        Direction = not Direction
        cur_j += 1 #move right
        update_encode()
        if cur_i == cur_j == 7:
            temp.append(sym)
            break
    elif cur_j == 0 or cur_j == 7:
        Direction = not Direction
        cur_i += 1 #move down
        update_encode()

    if Direction == True: # move down left
        cur_i += 1
        cur_j -= 1
    else: #move up right
        cur_i -= 1
        cur_j += 1
    update_encode()
```



Pixel level scan

We do run length encoding using zig-zag scan of the values in the block. (cur_i, cur_j) indicates the current (x,y) coordinate of the pixel in the block.

Direction indicates the direction of the diagonal, we move to the bottom left when Direction == True, or to the upper right otherwise.

When we reach the top or bottom rows (cur_i == 0 or cur_i == 7), we move to the right and change the direction of the diagonal.

When we reach the left or right-most columns (cur_j == 0 or cur_j == 7), we move down and change the direction of the diagonal.

```
temp = [] # current block encoding

# Process first element, q_dct[0][0]
prev = q_dct[0][0] # previous processed element
sym = [prev, 1] # symbol, [element, count]
```

```
def update_encode():
    global prev
    global sym
    if q_dct[cur_i][cur_j] != prev:
        temp.append(sym)
        sym = [q_dct[cur_i][cur_j], 1]
    else:
        sym[1] += 1
    prev = q_dct[cur_i][cur_j]
```

At each pixel (except for the first one), update_encode() is called.

prev indicates the value of the previous pixel.

sym indicates the current symbol the value is encoded as. Each symbol is represented by an array of 2 values, the first element indicates the value of the pixel, and the second element indicates how many of them appear in a row.

When the value of the current pixel is the same as the previous one, we add the second element of sym by one.

When they’re not the same, we append sym to temp (the set of symbols of the current block) then we create a new sym with the current symbol.

```
encoded.append(temp)
```

When we have gone through each pixel in a block, we add the current encoded block to encoded, which is the array that will contain the encoding for the whole image.

```
count = 0

for item in encoded:
    count += np.shape(item)[0]
count *= 2

print(count)
```
✓ 0.2s

```
72946
```

We can see after the encoding, the amount of values in the image is reduced from 512*512 = 262144 to 72946 values.

Decoding

To decode the encoded image, we simply reverse the process of our encoding process. Since the process is essentially the same as the encoding process but reversed, I won't go through too many details.

```
while True:

    if cur_i == 0 or cur_i == 7:
        Direction = not Direction
        cur_j += 1 #move right
        update_decode()
        if cur_i == cur_j == 7:
            break
    elif cur_j == 0 or cur_j == 7:
        Direction = not Direction
        cur_i += 1 #move down
        update_decode()

    if Direction == True: # move down left
        cur_i += 1
        cur_j -= 1
    else: #move up right
        cur_i -= 1
        cur_j += 1
    update_decode()
```

We go through each pixel in the same zig-zag manner and put the values back from the encoding array.

```
enc_block = np.array(encoded[cur_encoded_in])

#run length decoding
cur_i = 0
cur_j = 0
Direction = False # diagonal direction
temp = np.zeros((8,8)) # current block decoding

# Process first element
temp[0][0] = enc_block[0][0]
enc_block[0][1] -= 1
```

```
def update_decode():
    global enc_block
    global temp
    if enc_block[0][1] == 0:
        enc_block = enc_block[1:]

    temp[cur_i][cur_j] = enc_block[0][0]
    enc_block[0][1] -= 1
```

enc_block indicates the set of symbols from the current block. For each symbol, we put the value back to each pixel and when the second value of the symbol (how many times the value appears in a row) reaches 0, we move to the next symbol.

```
temp = np.multiply(temp,QUANT_MATR)
idct = cv2.idct(temp)
idct = idct + 128
decoded[cur_block_i:cur_block_i + 8, cur_block_j:cur_block_j + 8] = idct
```

```
if cur_block_j + 8 < dim:
    cur_block_j += 8
elif cur_block_i + 8 < dim:
    cur_block_i += 8
    cur_block_j = 0
else:
    break
```

After we assembled the block, we do an element wise multiplication with the quantization matrix, calculate the inverse DCT, then add 128 back to the original values.
Then we put the block back to its original position, row by row.

```
decoded = np.round(decoded)
decoded = np.uint8(decoded)
print(decoded)
```

After we assembled the full image, we round the values back to integers.
Here's the result.

Here's the original grayscaled image for comparison.