作业二

第一题：

8.

（a）原始的哈夫曼编码需要对符号的出现次数对其进行排序，根据排序好的结果建立哈夫曼树。哈夫曼算法需要有关信息源的先验统计知识，而这样的信息通常很难获得。这在多媒体应用中表现尤为突出，数据在到达之前是未知的，例如在直播（或流式）的音频和视频中。即使能够获得这些统计数字，符号表的传输仍然式一笔相当大的开销。而自适应哈夫曼编码中，统计数字式随着数据流的到达而动态地收集和更新的。概率不再是基于先验知识而是基于到目前为止实际收到的数据。通过自适应哈夫曼编码，我们可以随着新数据的到来而更新哈夫曼树，保证数据的压缩率，同时节省了符号表传输的开销。

（b）

i. 01010010101

①收到 01，收到字符 b，此时 b 的计数为 3，与最远的计数为 2 的 a 交换

②收到 01，收到字符 a，此时 a 的计数为 3

③收到 00，收到字符 NEW，说明有新字符准备接收

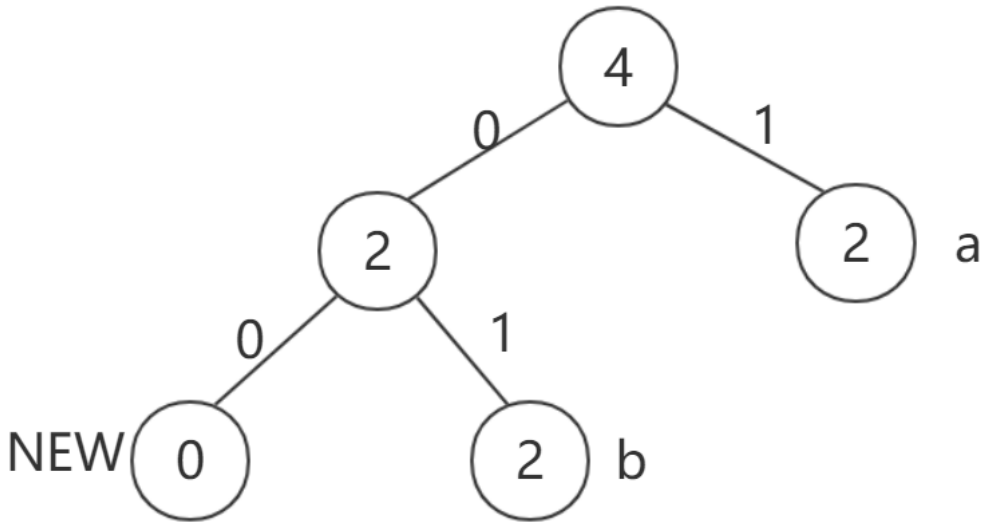
④收到 10，对照初始编码得知收到字符 c，原本 NEW 节点（a 的兄弟节点）变为子节点分别为 NEW 与 c 的子树（计数为 1）。此时 a 的父节点计数变为 4，为 b 计数 3+1。因此交换两节点。

⑤收到 101，收到字符 c，此时 c 的计数为 2。

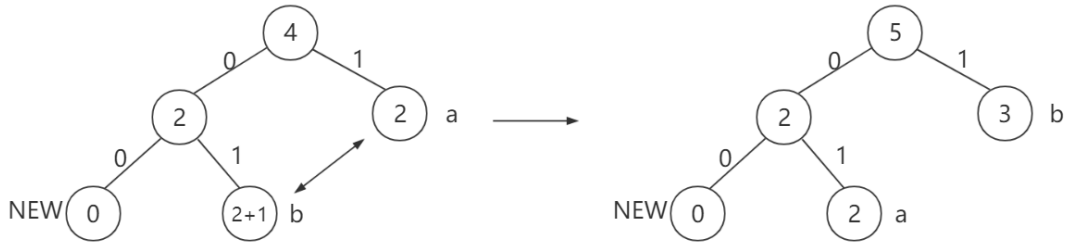
综上所述，收到的后续几个字母为 bacc。

ii.

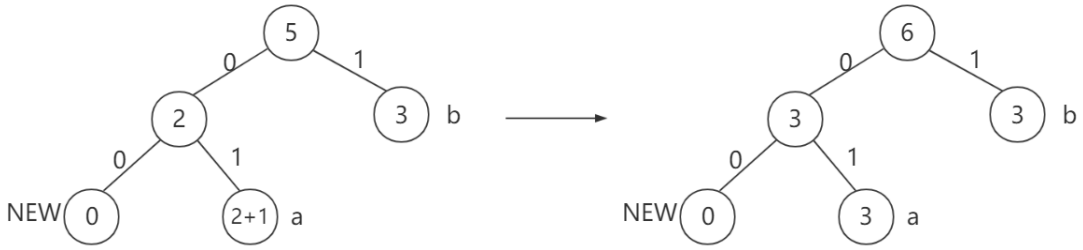
原始图：



① 接收 01，根据树得知接收 b，b 的计数+1 后变为 3，为 2+1，其中最远的 2 为 a 的节点，因此 b 与 a 交换得到右图。

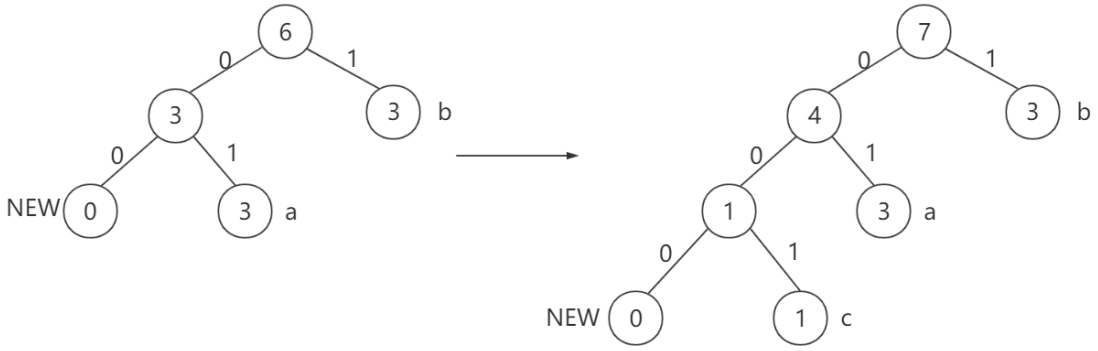


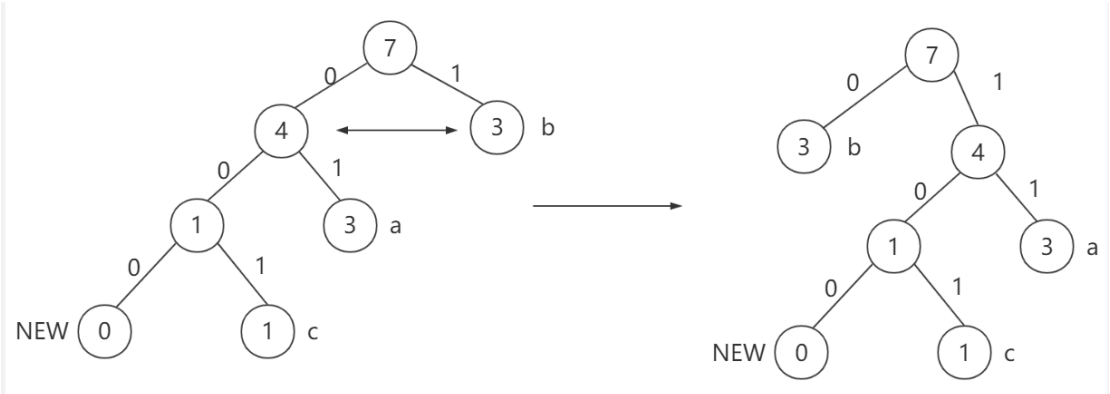
② 接收 01，根据树得知接收 a，a 的计数+1，其所有直系前辈节点都响应+1。



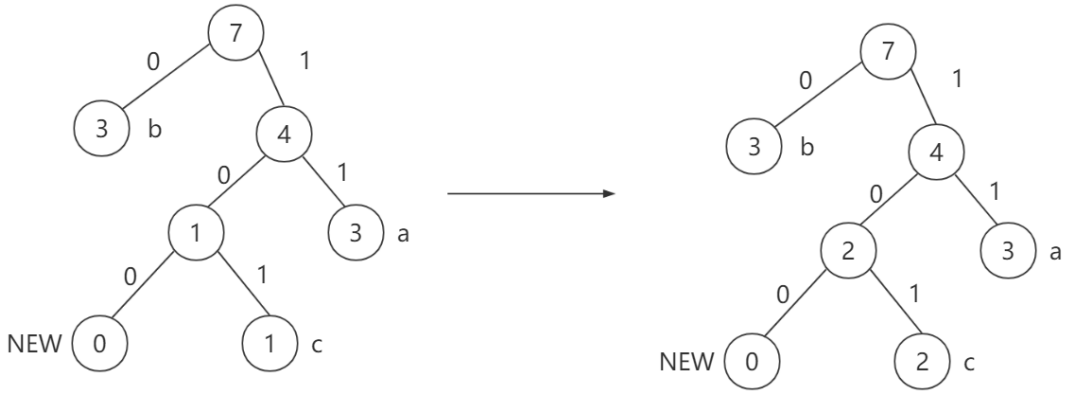
③ 接收 00，根据表可知为 NEW，象征下一字符为新字符

④ 接收 10，根据上一步得知该为新节点，查编码表得知 10 为 c，因此其与 NEW 共同组成原 NEW 节点位的两个子节点。最终更新树后 5 号位（a 的父节点）的计数位 4，为 b 节点的 3+1（只有 b 计数为 3），因此与 b 节点交换。





⑤ 接收 101，根据树得知新增节点为 c，c 节点计数+1，不需要交换。



以上每一步的最后一棵树是该步的自适应哈夫曼树。

第二题:

① 用 GIF 压缩 cartoon picture，用 JPEG 压缩 photograph。

1. GIF 是编码压缩的位图，GIF 是将图像的色彩从 24 位压缩到 8 位（256），每个像素仅用 8 位来存储色彩，通过映射的方式将像素映射成 24 位的 RGB 值。由于编码压缩为 8 位，其色彩的选项将大幅度减少（为 256），更适用于色彩少而独特的图像，比如 cartoon picture。因为 cartoon picture 中色彩鲜明而不丰富。
2. JPEG 的压缩是有损压缩，利用人眼对图片的感知的规律，通过减少高频分量的内容，使在尽可能不影响观看效果的前提下减少空间冗余。人眼对彩色敏感度发生接近变化的敏感度很低，JPEG 正是减少了高频内容，因此对于颜色丰富的 photograph，用 JPEG 更合适。

② JPEG 压缩实现（Python3）

1. RGB 转 YUV

   JPEG 会将彩色图像执行 YUV 或 YIQ 的颜色空间转换，二次采样 JPEG 采用 4：2：0，所以这里使用 YUV420 的颜色空间。在 JPEG 中使用的颜色模型是 YCbCr（由 YUV 调整而来）。对于一个 2*2 的块，我们会保存 4 个 Y 值，1 个 Cb 值（取 0 行 0 列的 Cb）与 1 个 Cr 值（取 1 行 0 列的 Cr），6 个值保存信息，因此 Cb 与 Cr 有一定损失。

   其中我们用函数 rgb2yuv（在 RGB2YUV.py）中来实现颜色模型转换与二次采样。并分别用三个二维数组保存采取的 Y、U、V 值。于是我们能得到三张分别用 Y、U、V 生成的灰度图（由于 U 和 V 损失为原来的 1/4，因此其图像的长宽也分别为原来的 1/2）（为了减少绝对值将 Y 值减去 128）：

```python
def rgb2yuv(img, width, height):
    Y = [[0 for j in range(width)] for i in range(height)]
    U = [[0 for j in range((width - 1) // 2 + 1)] for i in range((height - 1) // 2 + 1)]
    V = [[0 for j in range((width - 1) // 2 + 1)] for i in range((height - 1) // 2 + 1)]
    for i in range(height):
        flag = False
        if i % 2 == 1 or i == height - 1:
            flag = True
        for j in range(width):
            B = img[i][j][0]
            G = img[i][j][1]
            R = img[i][j][2]
            Y[i][j] = (0.299 * R + 0.587 * G + 0.144 * B - 128)
            if i % 2 == 0 and j % 2 == 0:
                U[i // 2][j // 2] = (-0.168736 * R + (-0.331264) * G + 0.5 * B)
            if flag:
                V[i // 2][j // 2] = (0.5 * R + (-0.418688) * G + (-0.081312) * B)
    return Y, U, V
```

2. 图像边长填充为 8 的倍数并等分

用 DCT.fill(img)函数对二次采样得到的 Y、U、V 图像分别用 0 填充知道其矩阵的 height 和 width 都是 8 的倍数，因为 DCT 函数的参数是一个 8*8 的矩阵。同样用 DCT.split(img)函数将图像以左到右，上到下的顺序分成多个 8*8 矩阵，并返回这些矩阵连成的数组。

```
def fill(self, img):
    width = len(img[0])
    height = len(img)
    if height % 8 != 0:
        for i in range(8 - height % 8):
            img.append([0 for j in range(width)])
    if width % 8 != 0:
        for row in img:
            for i in range(8 - width % 8):
                row.append(0)

    return img

def split(self, img):
    width = len(img[0])
    height = len(img)
    blocks = []
    for i in range(height // 8):
        for j in range(width // 8):
            temp = [[0 for i in range(8)] for i in range(8)]
            for r in range(8):
                for c in range(8):
                    temp[r][c] = img[i * 8 + r][j * 8 + c]
            blocks.append(temp)
    return blocks
```

3. 离散余弦变换

用 DCT.FDCT(block)函数对一个 8*8 矩阵进行二维离散余弦变换，保存得到的矩阵。

```
def FDCT(self, block):
    temp = [[0 for j in range(8)] for j in range(8)]
    for u in range(8):
        for v in range(8):
            n = 0
            for i in range(8):
                for j in range(8):
                    n += math.cos((2 * i + 1) * u * math.pi / 16) * math.cos((2 * j + 1) * v * math.pi / 16) * \
                        block[i][j]
            temp[u][v] = round(self.C(u) * self.C(v) / 4 * n)
    return temp
```

4. 量化

在类 Quantization 中保存成员变量 table0 与 table1 作为亮度和色度的量化表，调用 Quantization.quanY(img)与 Quantization.quanUV(img)分别用于对 Y 图像与 U、V 图像量化。

```
table0 = \
    [
        [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]
    ]

table1 = \
    [
        [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]
    ]
```

```
def quanY(self, img):
    temp = [[0 for j in range(8)] for j in range(8)]
    for i in range(8):
        for j in range(8):
            temp[i][j] = round(img[i][j] / self.table0[i][j])
    return temp

def quanUV(self, img):
    temp = [[0 for j in range(8)] for j in range(8)]
    for i in range(8):
        for j in range(8):
            temp[i][j] = round(img[i][j] / self.table1[i][j])
    return temp
```

5. AC 系数

   用 AC 类中的 ZScan(img)对一个图像进行 Z 型扫描，得到一个长度为 63 的数组（图像第一个像素并不需要，它将在 DC 系数中保存）。之后通过 RLC（array）对上面得到的数组 array 进行 RLC 得到他们的游长编码。

```
def ZScan(self, img):
    Z = []
    i = j = 0
    while i < 8 and j < 8:
        if j < 7:
            j = j + 1
        else:
            i = i + 1
        while j >= 0 and i < 8:
            Z.append(img[i][j])
            i = i + 1
            j = j - 1
        i = i - 1
        j = j + 1
        if i < 7:
            i = i + 1
        else:
            j = j + 1
        while j < 8 and i >= 0:
            Z.append(img[i][j])
            i = i - 1
            j = j + 1
        i = i + 1
        j = j - 1
    return Z
```

```
def RLC(self, a):
    temp = []
    numof0 = 0
    for num in a:
        if num == 0:
            numof0 += 1
        else:
            temp.append([numof0, num])
            numof0 = 0
    if numof0 != 0:
        temp.append([0, 0])
    return temp
```

6. DC 系数

用 DC 类中的 DPCM(blocks)函数对所有图像的 DC 系数提取并返回它们的 DPCM 编码数组。

```
def DPCM(self, blocks):
    temp = []
    temp.append(blocks[0][0][0])
    for i in range(1, len(blocks)):
        temp.append(blocks[i][0][0] - blocks[i-1][0][0])
    return temp
```

7. 熵编码

压缩部分是无损压缩，对 DC 系数（一个数组）采用可变字长整数编码。将一个 DC 系数分成 size 和 amplitude 两部分，配合 VLI(num)和 toB(num)函数将一个 DC 系数转换为一个[size(num), s(string(B))]。

```python
def toB(self, num):
    num = int(num)
    s = bin(abs(num)).replace('0b', '')
    if num < 0:
        s2 = ''
        for c in s:
            s2 += '0' if c == '1' else '1'
        return s2
    else:
        return s
```

```python
def VLI(self, num):
    if num == 0:
        return [0, '']
    s = self.toB(num)
    return [len(s), s]
```

这里的 s 已经是二进制串了（暂且用字符串存储方便操作），size 需要用哈夫曼编码压缩。这里使用 JPEG 推荐的哈夫曼编码（亮度与色度两个表）。

```python
DC_Y = {
    0: '00',
    1: '010',
    2: '011',
    3: '100',
    4: '101',
    5: '110',
    6: '1110',
    7: '11110',
    8: '111110',
    9: '1111110',
    10: '11111110',
    11: '111111110'
}
```

```python
DC_UV = {
    0: '00',
    1: '01',
    2: '10',
    3: '110',
    4: '1110',
    5: '11110',
    6: '111110',
    7: '1111110',
    8: '11111110',
    9: '111111110',
    10: '1111111110',
    11: '11111111110'
}
```

对于 AC 系数，我们知道 AC 系数采用有偿编码，由两个数 runlength 与 value 组成。先将 value 如同 DC 系数一样采用可变字长整数编码拆分成 size 与 amplitude，然后 runlength 与 size 合并为 symbol1，smplitude 独立为 symbol2。对于 runlength 大于 15 的数情况，在 symbol1 添加（15，0）表示（为了解码时能识别，应在前端添加）。然后对 symbol1 采用哈夫曼编码，对于 symbol2 则直接用上文可变字长整数编码得到的二进制码。Symbol1 采用的哈夫曼编码同样用 JPEG 推荐的哈夫曼编码表，由于过长不在报告中贴出。最终可用函数 AllCompressY 与 AllcompressUV（未贴出）将一个表格的数据转换为二进制字符串。参数为 DC 系数（一个）与 AC 系数（数组）。

```python
def AllCompressY(self, DC, arr):
    s = ''
    DC = self.VLI(DC)
    s += self.DC_Y[DC[0]] + DC[1]
    for num in arr:
        runlength = num[0]
        value = num[1]
        temp = self.VLI(value)
        while runlength > 15:
            runlength -= 15
            s += self.AC_Y[(15, 0)]
        s += self.AC_Y[(runlength, temp[0])]
        s += temp[1]
    return s
```

在 Test.py 中将对所有函数测试（实际上每个类的 python 文件的下方注释部分都是对这个类中函数的单元测试，由于不方便在报告中列出因此采用这种方式）。
这是 Y 图像的第一个 8*8 矩阵：

```
The first block of Y:
[108.10399999999996, 61.643, 2.8439999999999994, 11.016999999999996, 16.819999999999993, -27.271, 24.682999999999993, -28.876999999999995]
[101.20399999999998, 122.43499999999997, -17.445999999999998, -94.043, -90.3, -111.742, -125.608, -59.788]
[94.769, 88.024, 80.12699999999998, 63.55799999999999, -49.986000000000004, -126.206, -83.799, -112.65]
[82.72999999999999, 73.338, 51.59699999999998, 54.43199999999999, 81.38899999999998, -88.849, -13.37100000000001, -68.415]
[14.63900000000001, -83.2, -85.238, -64.915, 67.07999999999998, 44.708, 103.24099999999999, 23.62099999999998]
[-15.375000000000014, -81.87100000000001, -77.297, -117.467, -14.921000000000006, 101.00399999999999, 48.286, 63.27099999999999]
[-3.7680000000000007, -125.264, -94.39699999999999, -95.438, -123.813, 11.028999999999996, 97.64899999999997, 94.38199999999998]
[85.06700000000001, 8.84699999999998, -98.273, -94.17500000000001, -75.81200000000001, -120.025, -35.73000000000002, -24.876000000000005]
```

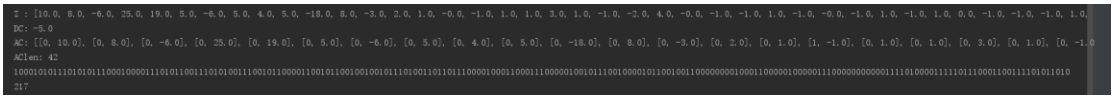
这是该矩阵的 DCT 变换结果（取整）：

```
The DCT of the block:
[-81.0, 111.0, 187.0, 84.0, 38.0, 30.0, -44.0, 52.0]
[96.0, 299.0, -87.0, -65.0, -1.0, -41.0, -48.0, -2.0]
[-87.0, 61.0, 132.0, -17.0, -1.0, -1.0, -1.0, -46.0]
[55.0, -301.0, 18.0, 116.0, -39.0, -1.0, 63.0, 2.0]
[83.0, 19.0, -60.0, 46.0, 54.0, 4.0, -80.0, 60.0]
[82.0, -26.0, -50.0, -52.0, 4.0, 77.0, -4.0, 4.0]
[36.0, 51.0, -61.0, -1.0, -2.0, 4.0, 7.0, -6.0]
[2.0, -75.0, -2.0, 2.0, 1.0, 78.0, -1.0, 1.0]
```

量化结果（取整）：

```
The Quantization of the DCT:
[-5.0, 10.0, 19.0, 5.0, 2.0, 1.0, -1.0, 1.0]
[8.0, 25.0, -6.0, -3.0, -0.0, -1.0, -1.0, -0.0]
[-6.0, 5.0, 8.0, -1.0, -0.0, -0.0, -0.0, -1.0]
[4.0, -18.0, 1.0, 4.0, -1.0, -0.0, 1.0, 0.0]
[5.0, 1.0, -2.0, 1.0, 1.0, 0.0, -1.0, 1.0]
[3.0, -1.0, -1.0, -1.0, 0.0, 1.0, -0.0, 0.0]
[1.0, 1.0, -1.0, -0.0, -0.0, 0.0, 0.0, -0.0]
[0.0, -1.0, -0.0, 0.0, 0.0, 1.0, -0.0, 0.0]
```

Z 字形扫描，DC 系数，AC 系数（太长未列全），二进制字符串，串长：



可以看到此处二进制长 217 位，意思是已经将一个 8*8*8（512）bit 的图像压缩为 217bit。

对于译码，我们需要事先保存图片的长宽（正如图片位流里会保存一样），以此计算出 Y，U，V 图像的矩阵数，才能对整个二进制流正确分割（在译码过程中分割）。在 Compress 类的 encoding 函数中，参数是位流（字符串形式）与宽，高（整型）。我们根据宽高得到 Y、U、V 图像的 8*8 矩阵的数量（我们知道 U 和 V 是一样多的），然后从位流头部开始移动两个指针。我们需要先得到上面使用的四个哈夫曼编码表的反向映射（这里用字典）。我们知道两个指针之间的二进制码的含义必定在几个状态之间转换：读取 DC 系数的 size（通过不断比较两个指针之间的位流是否为字典的 key，是的话得到其 value（这里指字典的 value），即 size，不是则移动尾指针）；通过 size 得到新的头尾指针，得到 amplitude；然后开始读取 AC 系数的(runlength,size)，如同上面得到 DC 系数的 size 一样，获得 size 后以此得到 amplitude；循环读取 AC 系数直到读取翻译到的(runlength，value)为(0,0)或得到 63 个 AC 系数为止，将 DC 系数与 AC 系数都加进各自的列表中（这两列表将存储全部 Y 矩阵的 DC 系数与 AC 系数）。U 与 V 同理，最终我们得到 Y、U、V 的 DC 系数与 AC 系数的数据（由于这段代码比较长不贴出，在 Compress.py 中）

```
def encoding(self, s, height, width):
```

```
return DCY, DCU, DCV, ACY, ACU, ACV
```

然后通过 DC 类中的 DPCM2(DC)函数，传入各自的 DC 系数，得到三个其元素是一个矩阵的列表，每个矩阵的[0][0]都通过 DC 系数还原为编码前

```
def DPCM2(self, arr):
    blocks = []
    for i in range(len(arr)):
        temp = [[0 for i in range(8)] for i in range(8)]
        if i == 0:
            temp[0][0] = arr[0]
        else:
            temp[0][0] = arr[i] + blocks[i-1][0][0]
        blocks.append(temp)
    return blocks
```

然后对每个矩阵分别通过对应的 DC 系数还原为 63 个数（AC.RLE(array)）

```
def RLE(self, a):
    temp = []
    for s in a:
        zero = s[0]
        num = s[1]
        if num == 0:
            for i in range(63 - len(temp)):
                temp.append(0)
            break
        for i in range(zero):
            temp.append(0)
        temp.append(num)
    return temp
```

再 Z 形填进矩阵中(AC.Z2Tab 函数)（这段较长不贴出）

```
def Z2Tab(self, a, table):
    arr = self.RLE(a)
    iter = 0
    i = j = 0
    while i < 8 and j < 8:
        if j < 7:
            j = j + 1
        else:
            i = i + 1
        while j >= 0 and i < 8:
            table[i][j] = arr[iter]
            i = i + 1
            j = j - 1
            iter += 1
        i = i - 1
        j = j + 1
        if i < 7:
            i = i + 1
        else:
            j = j + 1
        while j < 8 and i >= 0:
            table[i][j] = arr[iter]
            i = i - 1
            j = j + 1
            iter += 1
        i = i + 1
        j = j - 1
```

分别逆量化（Quantization.reY(img)与 Quantization.reUV(img)）（因为亮度与色度量化表不一样因此要分别操作）

```
def reY(self, img):
    temp = [[0 for j in range(8)] for i in range(8)]
    for i in range(8):
        for j in range(8):
            temp[i][j] = round(img[i][j] * self.table0[i][j])
    return temp

def reUV(self, img):
    temp = [[0 for j in range(8)] for i in range(8)]
    for i in range(8):
        for j in range(8):
            temp[i][j] = round(img[i][j] * self.table1[i][j])
    return temp
```

再二维逆离散余弦变换(DCT.IDCT(img)。

```
def IDCT(self, block):
    temp = [[0 for j in range(8)] for i in range(8)]
    for i in range(8):
        for j in range(8):
            n = 0
            for u in range(8):
                for v in range(8):
                    n += math.cos((2 * i + 1) * u * math.pi / 16) * math.cos((2 * j + 1) * v * math.pi / 16) * \
                         block[u][v] * self.C(u) * self.C(v) / 4
            temp[i][j] = round(n)
    return temp
```

这段操作每个函数的对象都是单个 8*8 矩阵

通过对所有矩阵进行同样的操作我们能得到所有当初刚分割完的 8*8 矩阵（不算损失的话）。

同样通过对这些矩阵进行合拼并将当初填充的 0 割掉，得到 Y、U、V 图像（这段代码较长不贴出，在 DCT.py 的 DCT.merge()中，注意需要图像的长宽作为参数）

```
def merge(self, imgY, imgU, imgV, height, width):
```

```
return Y, U, V
```

经此我们能得到 YUV420 的原图像。

关于测试代码，可以直接运行 Test.py，会打印得到对每个函数的测试，或者运行 Main.py，将执行对图像从压缩到解压的全部操作。Main.py 会打印压缩得到的位流长，并在代码的上一层目录中创建一个 txt.txt 文件用字节形式保存每个位的数据（主要是用于测试解压代码时可以免去压缩过程迅速开始测试）。

运行 Main.py 会打印位流的长度，图片的高度与宽度（因此推荐在终端窗口中运行而不是直接运行 py 文件），并显示解压后的图片

```
1388439
682 1024
```

我们直到 682*1024 个像素，如果直接用 RGB 的 24 位保存将要 16760832bit

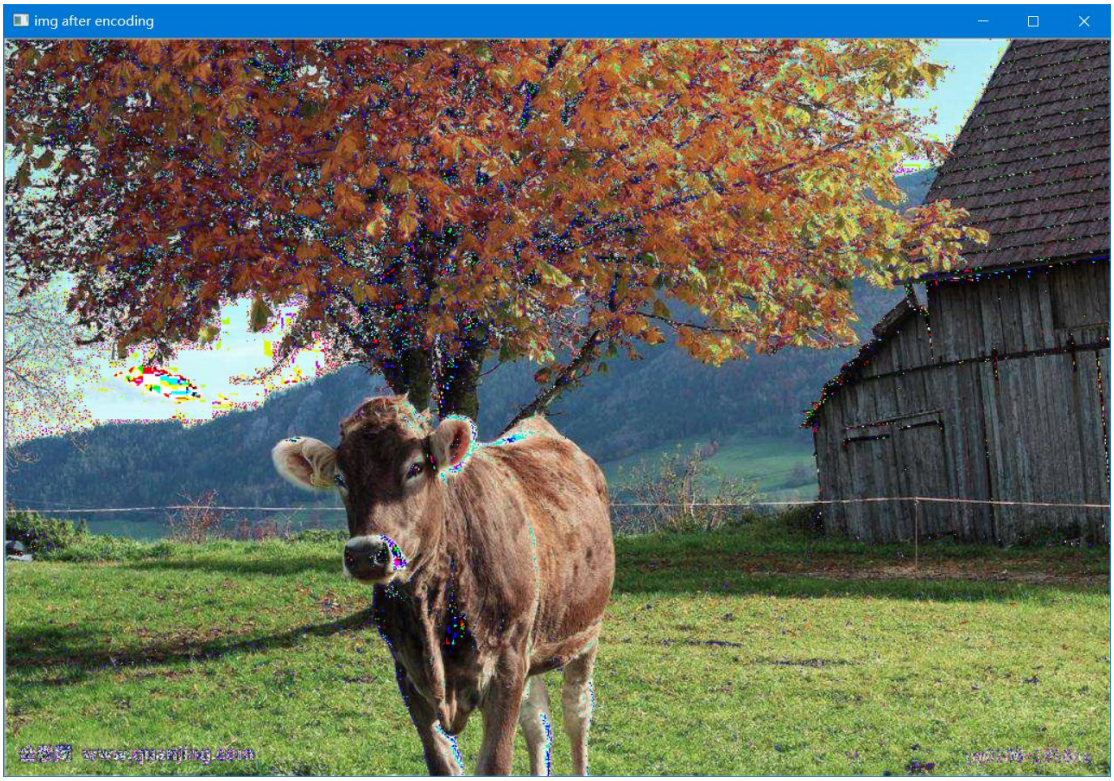
682*1024*8*3

16 760 832

而这里压缩后得到的位流是 1388439bit（虽然这里是字符串形式而非位流形式，但为了方便省去了以位流写文件的步骤，我们直接通过字符串的长度来对比）。压缩率是 8.3%。

然而压缩的代价也十分明显（如图）。其中黑色变为其他颜色的点主要是 YUV420 中大量色度损失导致的（直接将图片从 RGB 转为 YUV420 再直接转为 RGB 就会有这些点的损失，因此如此推断），颜色也同样有不少损失，远处的云能看到明显的格子化。

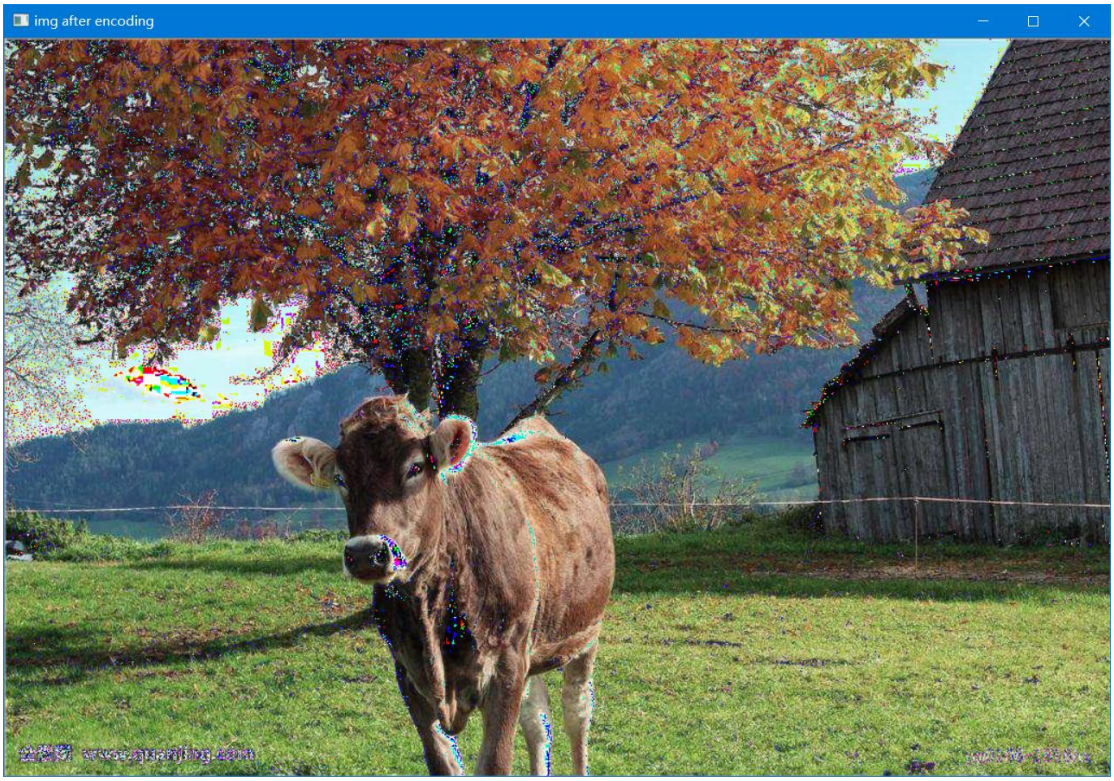
解压后：



原图：

③ 结果对比：

压缩率约为 8.3%的 JPEG 图（忽略文件信息等真正 JPEG 文件可能会造成的误差）



体积大约为 170KB。与原文件接近，或许是我二度采样哪里理解错了或者写错了导致失真严重。

用 PS 使压缩的 GIF 也接近 170KB（调整损失），得到的图像明显颜色（只有 256）不足导致颜色接近的区块容易模糊融在一起。但整体色彩比起 jpeg 更接近原图，但是缺少细节。

因此像照片这种细节多的图片，在体积相同的情况下还是选择 JPEG 更合适。或许是我的代码或者算法不够好导致的损失，也可能是我的二度采样是在原 JPEG 已经压缩过的二度采样的基础上的再次二度采样导致的损失，实际上原图 jpg 也是 170KB 体积，但表现效果十分好，比起我的压缩解压以及 PS 的转换为 GIF 都是如此。

GIF 与 JPEG 文件保存在根目录下。代码保存在 py 文件夹中。