Digital Media Homework 2

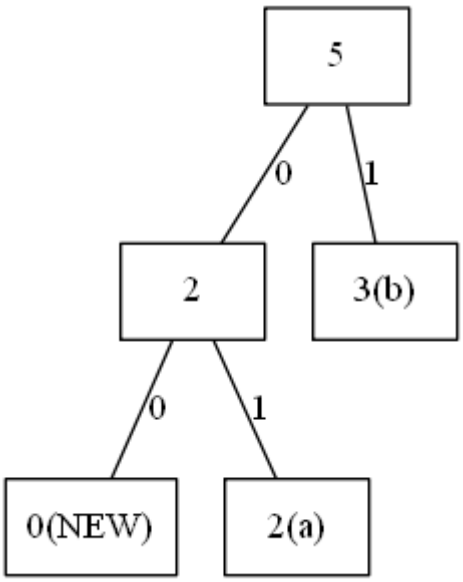
Question 1

Part (a)

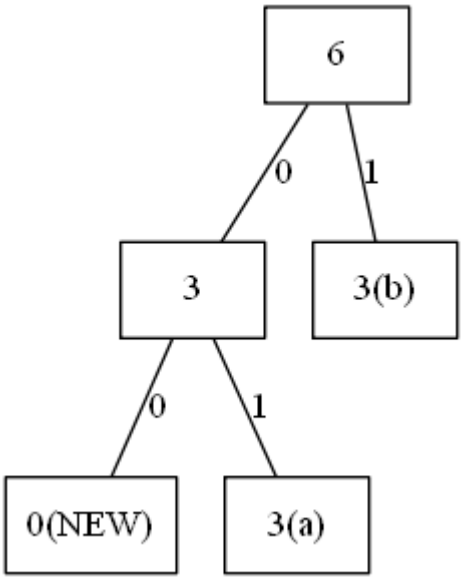
The advantages of Adaptive Huffman Coding are that it can address the problem that the source information is not completely available when coding it. Apart from that, the adaptive Huffman tree won't be saved in the header of files because it can be configured dynamically, which will save the overhead.

Part (b)

- The additional letters received are: **bacc**
- The derivation process is as follows:
  1. In the next few letters(**01**010010101), we first resolve **b** for **01** in `Fig.7.18`. And the tree is like Fig.1.1.
     - *Fig.1.1*: Huffman code now: **00 -> NEW, 01 -> a, 1 -> b**

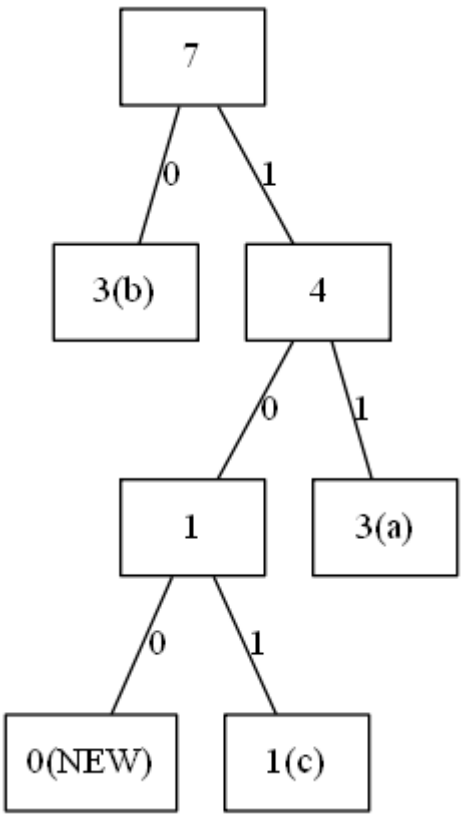


  2. And then(01**01**0010101), **a** can be resolved from **01**. And the tree is like in Fig.1.2.
     - *Fig.1.2*: Huffman code now: **00 -> NEW, 01 -> a, 1 -> b**. (the same with last step)



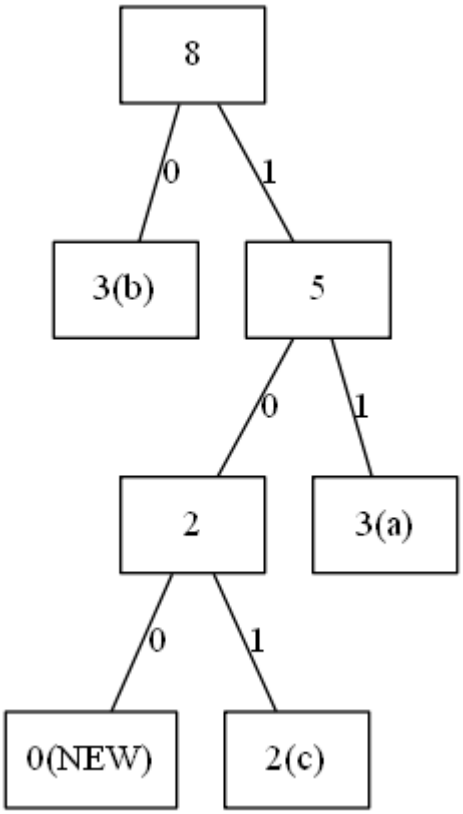
3. And then(0101**00**10101), **NEW** can be resolved from **00**. And the following Huffman code can be resolve to **c** for **10**, according to the initial coding. And the tree is like in Fig.1.3.

- *Fig.1.3*: Huffman code now: **0 -> b, 100 -> NEW, 101 -> c, 11 -> a**.



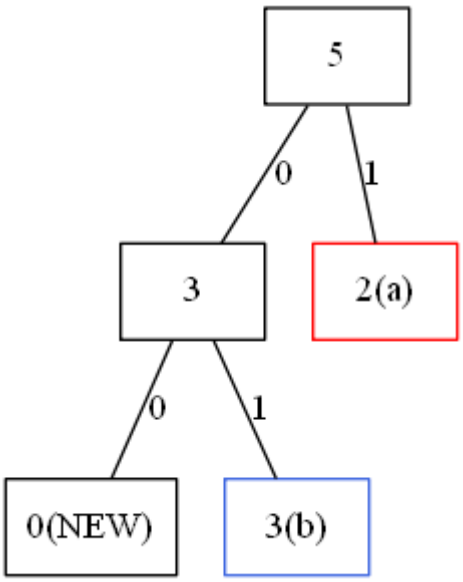
4. And then(01010010**101**), **c** can be resolved from **101**. And the tree is like in Fig.1.4.

   Fig.1.4: Huffman code now: **0 -> b, 100 -> NEW, 101 -> c, 11 -> a**.

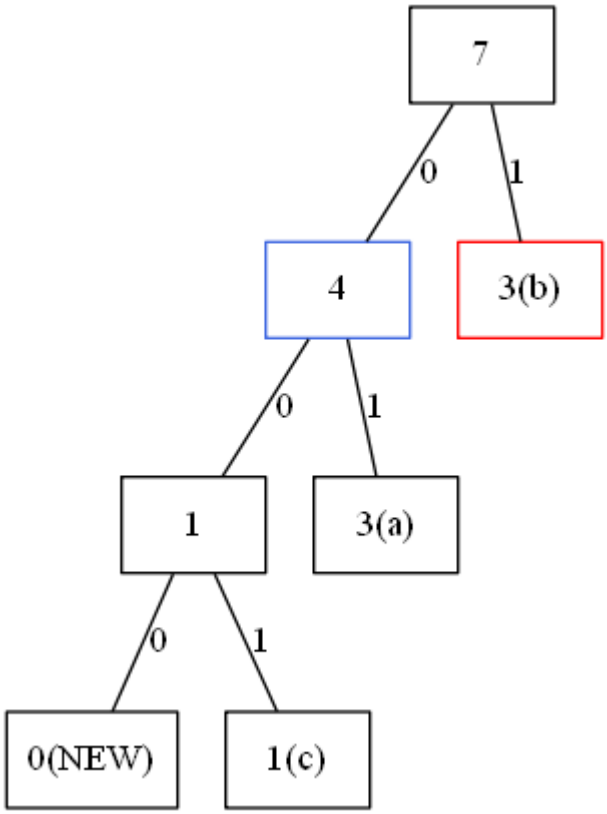


- The adaptive Huffman trees is like above. And it swapped for several times.
  1. From Fig.7.18 to Fig.1.1, another **b** is added. b's count of frequency becomes 3, which is larger than a's(2).

     > Nodes are numbered in order from left to right, bottom to top.

     The tree should be swapped. The farthest node with count 2 (node in **red**) is swapped with the node whose count has just been increased to 3 (node in **blue**). And then it becomes Fig.1.1.
  2. From Fig.1.1 to Fig.1.2, the count of frequency of a increased, but the structure of the tree remains.
  3. From Fig.1.2 to Fig.1.3, the number in the blue node is larger than that in the red node.



The tree should be swapped. The farthest node with count 3 (node in **red**) is swapped with the node whose count has just been increased to 4 (node in **blue**). And then it becomes Fig.1.3. And because the blue node is not a leaf-node, the entire subtree will go with it. Then the tree becomes in Fig.1.3.

4. From Fig.1.3 to Fig.1.4, the count of frequency of c increased, but the structure of the tree remains.

Question 2

理论原因分析

理论上GIF通过减少颜色通道的方法压缩图片，使得颜色通道从24bit变为8bit。这能够将压缩率控制在33%。GIF压缩了色彩通道后会使得颜色颗粒感较之前更为明显，但如果图片分辨率较高则不会有很大的影响。

而JPEG的压缩通过游长编码后能够使得连续的0bit被一个数对表示，同时其所使用的标准的DC, AC和Huffman表格也使得编码被压缩。但是在失真的主要步骤，量化中，JPEG的恢复使得图像会发生总体颜色的细微改变。但是细节之处应该不会向GIF一样产生很严重的不连续像素点。

程序实现 - JPEG压算法

基本步骤（编码）

1. 将图像补全并进行色彩空间转换
2. 将图像分为8*8的小块并进行DCT变换
3. 进行量化
4. 进行ZigZag扫描
5. 将结果使用DC和AC编码为DPCM和RLC
6. 使用Huffman进行熵编码获得数据

7. （编写Header和Tables省略了）

将图像补全并进行色彩空间转换

> `YCbCr.java`

- 如果图像的长或宽不是16的倍数，则将其补全为16的倍数

  > 注意这里如果是8的倍数会导致在计算Cb和Cr亚采样的时候除以2不完整，会导致数组越界

- 根据公式将DCT转换为YCbCr:
  - 对矩阵进行遍历，对每个像素执行下面操作：

```
Ychannel[i][j] = 0.299*r + 0.587*g + 0.114*b;
Cbchannel[i/2][j/2] = -0.169*r - 0.331*g + 0.5*b + 128;
Crchannel[i/2][j/2] = 0.5*r - 0.419*g - 0.081*b + 128;
```

  > 这里需要注意对图像的补全部分外加判断，并将其rgb均设置为1带入公式计算YCbCr对应像素的同道值。

将图像分为8*8的小块并进行DCT变换

> `DCTcoding.java`

- 变换后使用一个三维数组存储，其中第一个维度为宏块的id，后两个维度对应每个宏块的8*8像素
- 使用两层嵌套循环遍历矩阵，其中每次遍历时横坐标和纵坐标均+8，跳到下一个宏块
- 循环体中，根据 `i*width/64 + j/8` 确定宏块在数组内的下标，并遍历8*8宏块，将图像数据对应的矩阵存入三维数组的对应位置。
- DCT变换的步骤根据标准的DCT表对于每个宏块对应的8*8矩阵进行矩阵乘法操作。

  > 我的代码中直接将DCT变换的矩阵和其转置存位常量，在DCT变换的步骤中手工进行了矩阵乘法。

进行量化

> Quatization.java

- 量化的步骤是根据上一步的结果，将每个8*8的矩阵根据所在通道除以对应的标准量化矩阵的对应元素，并用Math.round进行四舍五入得到。

进行ZigZag扫描

> ZigZag.java

- 将ZigZag扫描顺序的矩阵存为常量，读取量化后的三维矩阵，将其后面的8*8宏块根据ZigZag扫描顺序化为一维。
- 注意这里的第一个数字为预测器，为当前宏块的第一个像素变换后的值减去上一个宏块变换后的像素值。后面的数字则直接进行编码。

将结果使用DC和AC编码为DPCM和RLC

进行游长编码

- 对于每一个宏块（每64元素的数组），将其采用数对存储（二维数组），这里由于每个宏块内0的个数与集中程度不一样，所以采用Vector存储
- 遍历64个数组元素：第一个元素直接作为Vector的第一个元素（即该数对只有一个元素），后面的数组元素直接进行统计
    - 如果从当前元素到最后一个元素均为0，则将数对(0,0)存入。
    - 如果从当前元素开始已经有16个元素为0，则将数对(15,0)存入Vector并重置计数器。
    - 否则，第一个数（RUNLENGTH）赋值为截至当前不为0的数，计数器中0的个数，第二个数（AMPLITITUDE）赋值为当前这个不为0的数。

进行熵编码

- 根据游长编码得到的数组，遍历数组元素，获得每个数对的数据。
- 对于第一个元素，将数对中的第一个数直接存入熵编码的AMPLITITUDE部分，计算出AMPLITITUDE对应的标准HuffmanCode的长度，作为数对的第一个数(SIZE)存入。
- 对于剩下的数对元素，判断其RUNLENGTH和AMPLITITUDE不同时为0，并将游长编码后的第二个数（AMPLITITUDE）作为熵编码的第三个数，将AMPLITITUDE对应的标准HuffmanCode的长度存入熵编码的第二个数，第一个数仍然为RUNLENGTH保持不变。

对于熵编码的SYMBOL1进行Huffman编码

- 对于DC编码部分，SIZE直接根据Lum和Color的转换表转为对应的Huffman编码。
- 对于AC编码部分，需要将其RUNLENGTH/SIZE转换为字符串，作为key获得对应Huffman码的Value（在解码的时候将Value作为Key，Key作为Value反向寻找即可。

对于熵编码的SYMBOL2进行标准Huffman编码

- 根据标准Huffman编码的算法计算即可。

解码过程

解码过程和编码过程相反，主要需要注意以下几点：

1. 对于进行Huffman编码过的数据，首先获得宏块第一个数的信息，由于其熵编码为（SIZE, AMP），而SIZE经过标准的DC表格获得，由于该表格对应的哈夫曼表部分，长度为2~9（Lum）和2~11（Lum），且前缀不会重复，则遍历从数组开头开始的2~9长度的字符串，一旦有符合，则根据DC表格解析出该数组对应的值，为当前宏块的DC的SIZE值
2. 读取后面的SIZE位字符串，根据标准哈夫曼表转化出数对的AMP，恢复出宏块第一个像素压缩后的值。
3. 后面的步骤相似，只是遍历长度从2~16的字符串，根据AC表找到哈夫曼表对应的“RUNLENGTH/SIZE”并解析出对应的runlength和size. 如果RUNLENGTH和AMP均为0，则将从当前位置到第64个位置均补0. 否则步RUNLENGTH个0并将之后的一个数字补为AMP。注意这里一边补需要一边计数，当解析出了64个数字后，读取下一个宏块。
4. ZigZag过程将64位按顺序放回8*8数组中
5. 反量化的步骤将原来的除法步骤变为乘法
6. DCT逆变换的过程将矩阵的乘法调换顺序

$$F = A^T f A$$

7. 三个通道计算完毕后将YCbCr转换为RGB并根据Java的BufferedImage写图像。

结果对比

压缩率

- 原始图像大小：
  - 动物图片：

$$1024 \times 682 \times 24 = 16760832$$

  - 卡通图片：

$$1000 \times 715 \times 24 = 17160000$$

- JPEG压缩后直接用代码计算了三通道01串长度的总和为：1342653（动物图片），750846（卡通图片）.
  - 动物图像的压缩率为：8%
  - 卡通图像的压缩率为：4.3%
- 使用画图工具将二者转化为GIF格式后：
  - 动物图片大小为：

$$1024 \times 682 \times 8 = 5586944$$

  - 卡通图片大小为：

$$1000 \times 715 \times 8 = 5720000$$

  - 压缩率均为33%。

失真率

动物图片

三幅图片大体上看没有太大差异，色彩上看，JPEG在压缩后饱和度仿佛低了一度（观察黄色的书叶可以看出GIF对于颜色保持的较好），色彩上的失真上GIF损失较小。

原图：

JPEG编码后：

GIF转换后：

使用画图工具放大到500%，可以看出JPEG压缩的图片相对较为连续，而GIF图片出现了很多明显像素点（猜测是由于GIF色彩深度只有8bit造成的。感官上在细节部分GIF失真率较高。

> 从左到右为：JPEG压缩，GIF压缩，原图

卡通图片

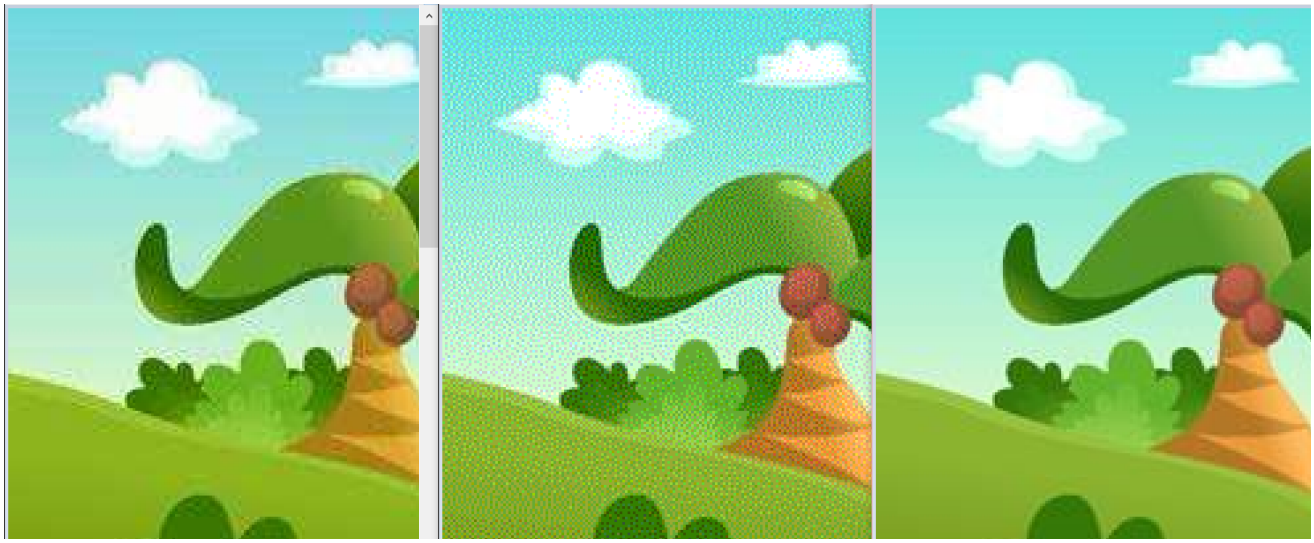
和动物图片相同，但从大图上看，总体色彩上JPEG的失真度较高。

原图：

JPEG编码后：

GIF转换后：

与上文相同，GIF仍然呈现出像素点颗粒点明显的形科，而JPEG与原图的细节之处较为相似。



综上所述，可以得出结论：GIF通过将色彩深度压缩的方式将图片压缩到三分之一，在局部细节上没有JPEG效果好，但是总体颜色与原图更为相近。JPEG图像中，动物图像相比于卡通图像压缩率更高，原因是因为卡通图像中有大范围相同的颜色区域，这使得在进行ZigZag扫描后的游长编码中可以将更多像素值集中到一个数对保存。而动物图像由于细节较多，较少有色彩大范围相同的区域，使得压缩率比如卡通图片。