Paper: Research on Image Compression Coding Technology

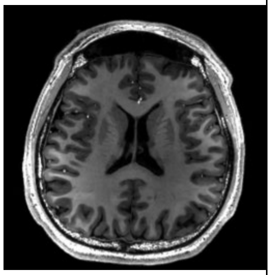
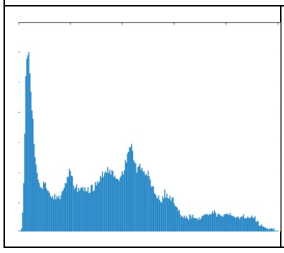
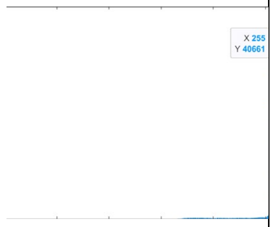
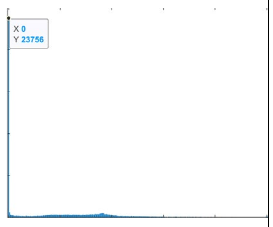
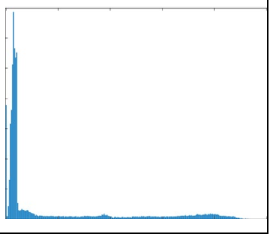
Knowledge of compression coding theory

- **Huffman coding**
  1. First, the probability of source symbols is sorted in descending order.
  2. Then add up the two minimum probabilities and repeat the step, always arranging the high
  3. Next, the path from probability 1 to each source symbol is marked, and the 0 and 1 along the path are recorded in sequence. Thus, the Hoffman codeword of the symbol is obtained.
  4. Finally, the left side of each pair of combinations is designated as 0, and the right side is designated as 1 (or vice versa).
- **Run length coding**

  Usually, computer image is composed of a large number of color blocks of the same color. When storing the image, many consecutive scanning lines or continuous pixels on the same scanning line carry the same color value in the process of scanning the color blocks. Run-length coding is a coding method that stores a pixel value and the number of pixels with the same color instead of storing the same color blocks in the image one by one. Simply put, the color values and count values are used to represent the same color pixel values in a row.
- **Predictive coding**

  The adjacent pixels of the image are correlated in gray level. Predictive coding uses this feature to predict the current signal through the previous signal, and then differential coding through the prediction error.

Experimental Images:

A	B	C	D
	Adaptive pre-OCR cleanup of grayscale document images [illegible] ABSTRACT [illegible] 1. INTRODUCTION [illegible]		
Distribution of pixel value			
			

Experimental Simulation

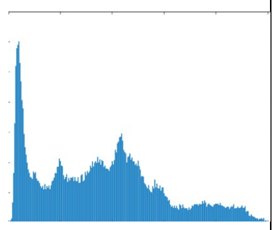
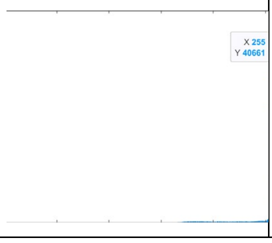
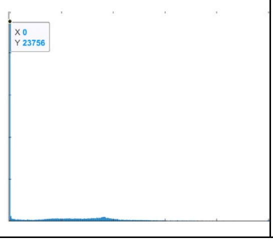
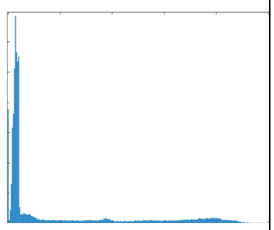
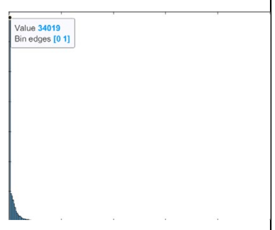
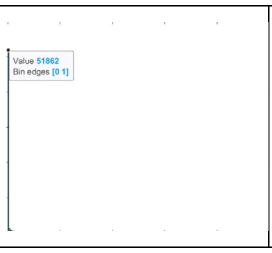
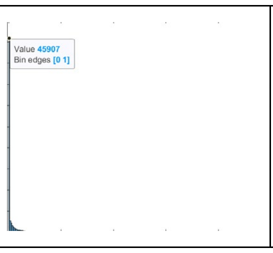
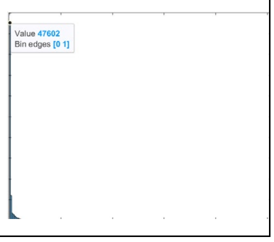
- **Huffman Coding Simulation**

Name	Data Size (bit)	Bit / pixel
Source	256x256x1x8 = 524288 bit	8
After Coding for Image A	500055 bit	7.6302
After Coding for Image B	229095 bit	3.4957
After Coding for Image C	383009 bit	5.8443
After Coding for Image D	381031 bit	5.8141

- **Run-length coding simulation (pixel value, length of same pixel)**

Name	Data Size (bit)	Bit / pixel
Source	256x256x1x8 = 524288 bit	8
After Coding for Image A	118772x8 = 950176 bit	14.4985
After Coding for Image B	62296x8 = 468368 bit	7.6044
After Coding for Image C	78272x8 = 626176 bit	9.5546
After Coding for Image D	57804x8 = 462432 bit	7.0561

- **Predictive coding simulation**

A	B	C	D
			
After Coding			
			

- **Conclusion**

  Huffman Coding 通常能比 Run-length coding 有更好的壓縮。
  Run-length coding 在 X 光影像與文件有比較好的壓縮，可以指出有更多連續相同像素值的影像才適合使用 Run-length coding。
  Predictive coding 可以將影像的轉換成更集中的分布，之後可以接著使用其他 coding 方法。

```matlab
%Input Size: 256x256
data = imread("lena.png");
img = rgb2gray(data);
data = reshape(img, [1, 256*256]);

% Huffman Coding Simulation
[N, symbols] = hist(data,double(unique(data)));
p = N / (256*256);

[dict, avglen] = huffmandict(symbols,p);

lens = zeros(size(symbols));
for v = 1:length(symbols)
    len = length(cell2mat(dict(v,2,:)));
    lens(v) = len;
end

% bar(symbols, N);
Huffman_Coding_N = sum(lens.*N, "all");
Huffman_Coding_P = sum(lens.*p, "all");

Run_Length_Coding=[];
c=1;
for i=1:length(data)-1
    if(data(i)==data(i+1))
        c=c+1;
    else
        Run_Length_Coding=[Run_Length_Coding,c,data(i),];
    c=1;
    end
end
Run_Length_Coding=[Run_Length_Coding,c,data(length(data))];


Predictive_Coding = img(1:256, 2:256) - img(1:256, 1:255);

% histogram(Predictive_Coding)
% xlim([0 255])
```

Paper: A Review of Image Compression Techniques

- **Introduction**
  - Three generally fundamental redundancies
    - Psycho-visual Redundancy
    - Inter-pixel Redundancy
    - Coding Redundancy

- **IMAGE COMPRESSION STANDARD**
  - Lossy Compression methods
    - 通常的步驟: Transform -> Quantization -> Entropy coding

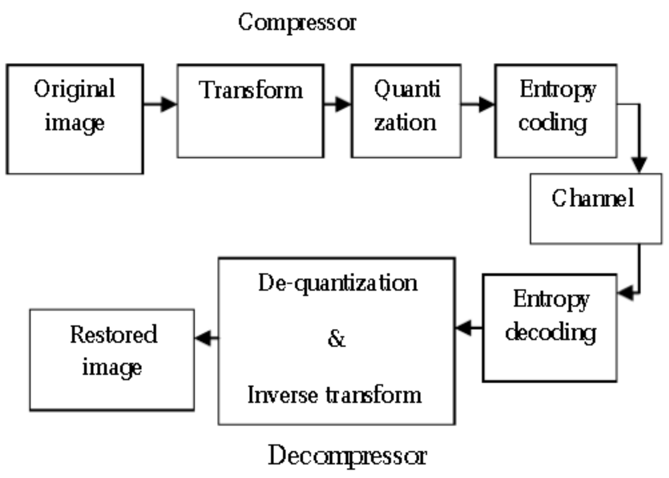


Fig 1: Lossy image compression

  - Lossless image compression
    - 通常的步驟: Transform -> Entropy coding
    - Comment: 對比 Lossy，少了 Quantization 步驟，主要是少了 Psycho-visual Redundancy，所以通常恢復的影像沒有畫值的損失。

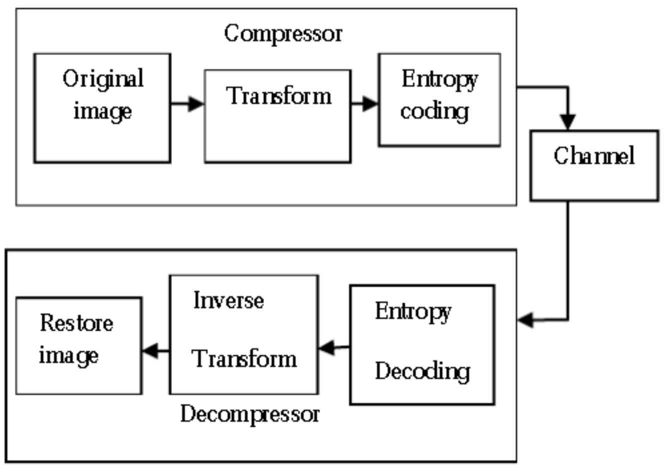


Fig 2: Lossless image compression

- **IMAGE REPRESENTATION FORMATS**
  1. Lossless image representation formats
     - ◆ PNG (Portable Network Graphics)
       - PNG 的成立是為了共同改進並用一種不需要專利許可即可使用的影像格式來替代 GIF 格式
       - 使用 DEFLATE 壓縮演算法(混和 LZ77 algorithm 和 Huffman coding)
       - 有 24 bit 與 32 bit
     - ◆ TIFF (Tagged Image File Format)
  2. Lossy image representation formats
     - ◆ JPEG (Joint Photographic Experts Group)
       - 特點: 可以調整壓縮率(影響影像品質)
       - 基於人類對亮度比色度更敏感的特性，再來是對於邊緣(頻率更高)更敏感
       - 網路儲存常用的格式
     - ◆ JPEG 2000 (Joint Photographic Experts Group 2000)
       - Wavelet-based image compression standard
       - JPEG 2000 的壓縮比優於 JPEG
       - 有非常高壓縮率 (比 JPEG 更模糊)

- **EVALUATION OF COMPRESSED IMAGE**
  1. **MSE**

$$\mathrm{MSE} = \frac{1}{\mathrm{MN}}\sum_{i=1}^{M}\sum_{j=1}^{N}(x(i,j) - y(i,j))^2$$

- **PSNR**

$$PSNR(dB) = 10log_{10}\left(\frac{255^2}{\mathrm{MSE}}\right)$$

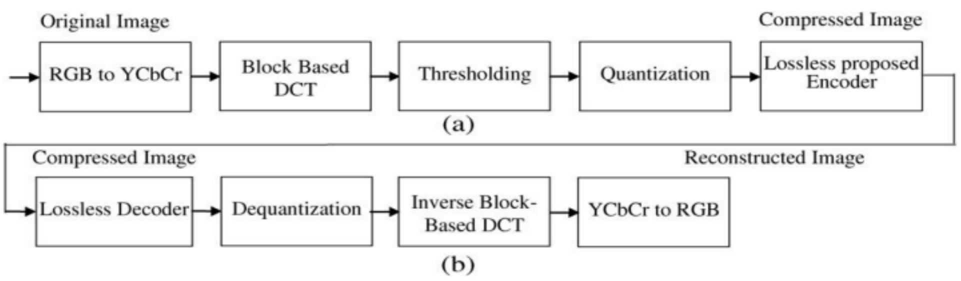
- **LOSSY vs LOSSLESS COMPRESSION**
  1. Lossy methods
     - ◆ 常用於壓縮聲音、影像或影片
     - ◆ 有損影片壓縮幾乎比影像有更好的壓縮率
     - ◆ 通常是為了減少傳輸時間或儲存空間
     - ◆ 基於人類視覺的特性為主要壓縮
  2. Lossless methods
     - ◆ 通常利用統計冗餘來更簡潔的表達
     - ◆ 可以重建實際的影像 (可逆的壓縮)

Paper: Jpeg Image Compression Using Discrete Cosine Transform - A Survey

- **JPEG Compressed Process Steps for color images This section presents jpeg compression steps**
  1. An RGB to YCbCr color space conversion (color specification)
  2. Original image is divided into blocks of 8 x 8.
  3. The pixel values within each block range from [-128 to 127] but pixel values of a black and white image range from [0-255] so, each block is shifted from[0-255] to [-128 to 127].
  4. The DCT works from left to right, top to bottom thereby it is applied to each block.
  5. Each block is compressed through quantization.
  6. Quantized matrix is entropy encoded.

- **Compressed image is reconstructed through reverse process. This process uses the inverse Discrete Cosine Transform (IDCT).**

- 壓縮與解壓流程圖



- **Color Specification (RGB to YCbCr)**

$$\begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.334 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 0 \\ 128 \\ 128 \end{pmatrix}$$

- **Discrete Cosine Transform (Matlab: dct2, idct2)**

This equation gives **the forward 2D_DCT transformation:**

$$F(u,v) = \frac{2}{N} C(u)C(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2N}\right]$$

for $u = 0,\dots,N-1$ and $v = 0,\dots,N-1$

where $N = 8$ and $C(k) = \begin{cases} 1/\sqrt{2} & \text{for } k = 0 \\ 1 & \text{otherwise} \end{cases}$

This equation gives **the inverse 2D_DCT transformation:**

$$f(x,y) = \frac{2}{N} \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} C(u)C(v) F(u,v) \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2N}\right]$$

for $x = 0,\dots,N-1$ and $y = 0,\dots,N-1$ where $N = 8$

- **Quantization in JPEG**

$$F(u,v)_{Quantization} = round\left(\frac{F(u,v)}{Q(u,v)}\right)$$

$$F(u,v)_{deQ} = F(u,v)_{Quantization} \times Q(u,v)$$

  - Matrix Qr and Qc defines the Q matrix for luminance and chrominance components

$$Q_Y = \begin{pmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{pmatrix}$$

$$Q_C = \begin{pmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{pmatrix}$$

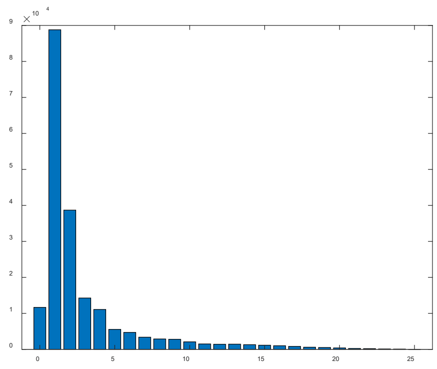
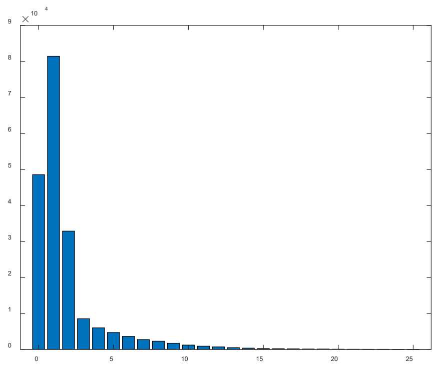
- **Huffman Encoding**
  - 使用與第一篇 paper 一樣的 Code

Experimental Images (Source Image):

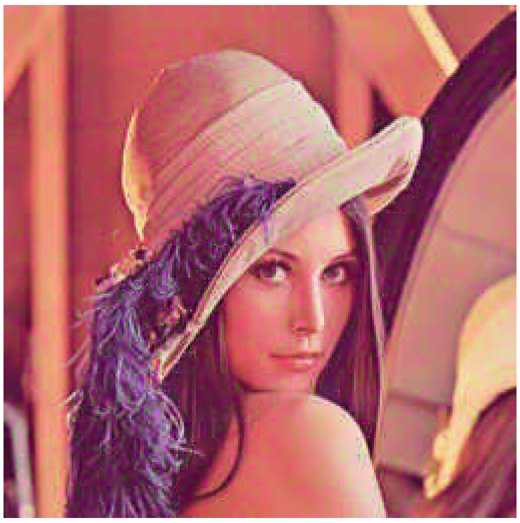
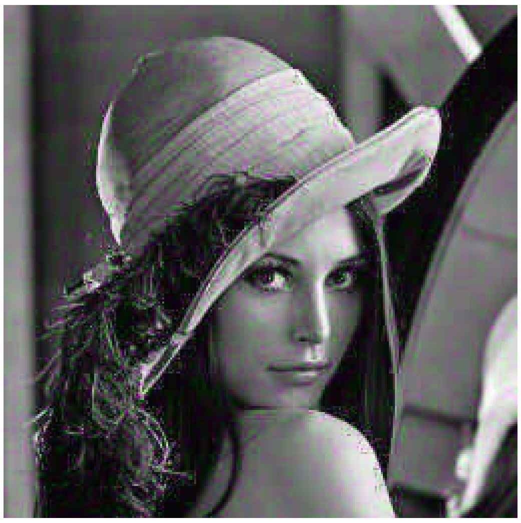
A	B

Name	Data Size (bit)	Bit / pixel
Source for Image A	256x256x3x8 = 1572864 bit	8
Compression for Image A	551304 bit	2.8041
Source for Image B	256x256x3x8 = 524288 bit	8
Compression for Image B	488367 bit	2.4840

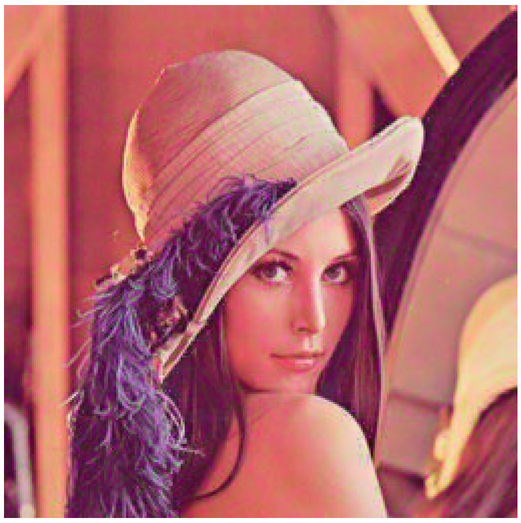
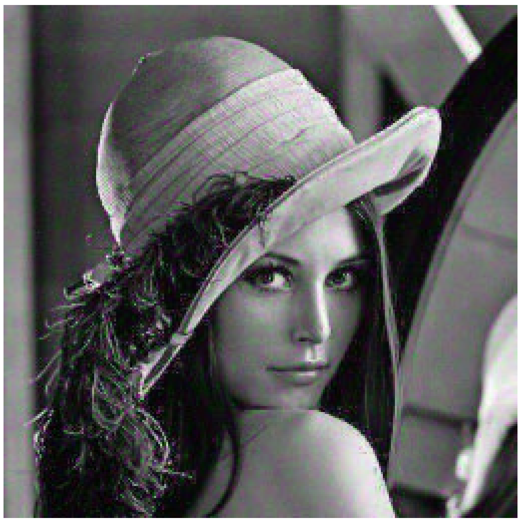
Distribution of Compression Value:

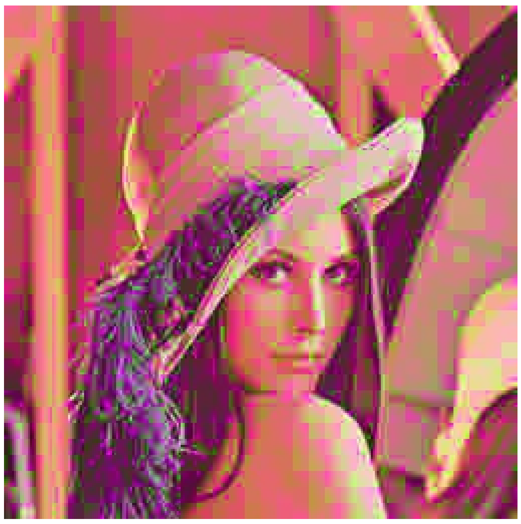
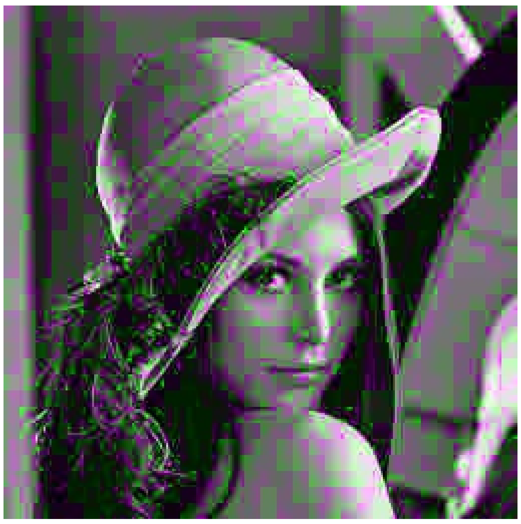
A	B
	

Experimental Images (Reconstructed Image):

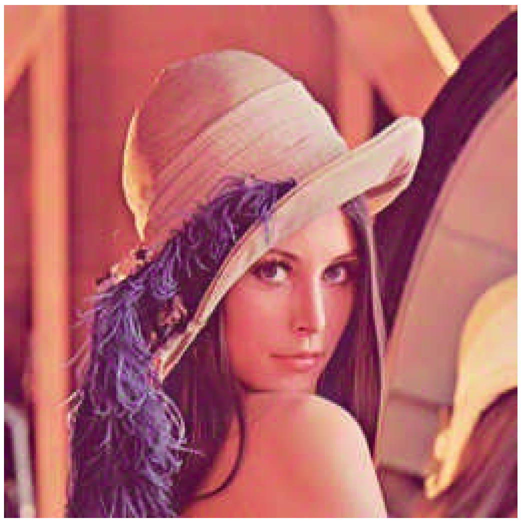
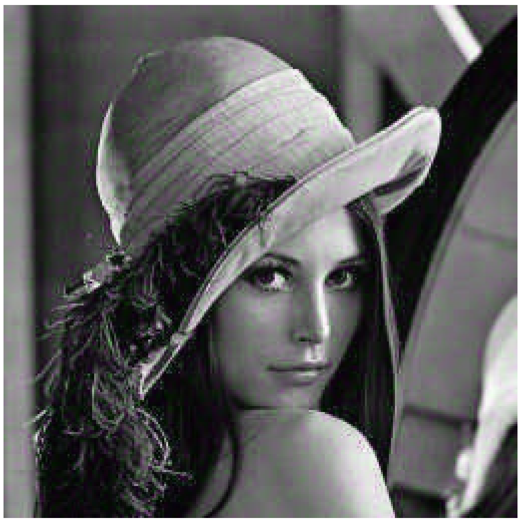
A	B
	

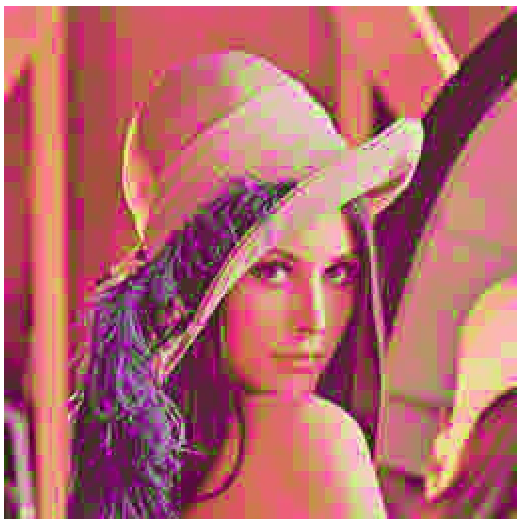
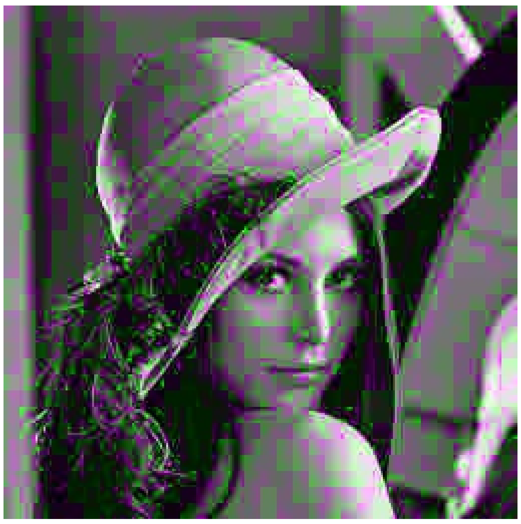
Experimental Images (Reconstructed Image):

將 **Qr** 改成 **1 (**即 **Qr(:) = 1)**	
A	**B**
	
將 luminance 量化程度縮小，可以發現重建影像與原始 **Qr** 的結果無明顯差異 ● **A** 影像可以省下儲存空間 ● **B** 影像需要更多儲存空間 ● **A** 影像中每個 **Pixel** 需要 **2.6624 bit** ● **B** 影像中每個 **Pixel** 需要 **2.8678 bit** ● 此處 **RBG** 比灰階影像的壓縮還更好	

將 **Qr** 改成 **100 (**即 **Qr(:) = 100)**	
A	**B**
	
將 luminance 量化程度加大，可以發現重建影像有多個突兀色塊 ● **A** 影像中每個 **Pixel** 需要 **1.2694 bit** ● **B** 影像中每個 **Pixel** 需要 **1.2906 bit** ● 此處 **RBG** 比灰階影像的壓縮還更好	

Experimental Images (Reconstructed Image):

將 **Qr** 改成 **1 (**即 **Qr(:) = 1)**	
A	**B**
	
將 chrominance 量化程度縮小，可以發現重建影像似乎有更清晰 ● 壓縮率減少 ● 需要更多儲存空間 ● **A** 影像中每個 **Pixel** 需要 **3.8951 bit** ● **B** 影像中每個 **Pixel** 需要 **4.2285 bit** ● 此處 **RBG** 比灰階影像的壓縮還更好	

將 **Qr** 改成 **100 (**即 **Qr(:) = 100)**	
A	**B**
	
將 chrominance 量化程度加大，可以發現重建影像有多個突兀色塊 ● **A** 影像中每個 **Pixel** 需要 **1.2880 bit** ● **B** 影像中每個 **Pixel** 需要 **1.3098 bit** ● 此處 **RBG** 比灰階影像的壓縮還更好	

```
imgRGB=imread('lena.png');
imgRGB = im2double(imgRGB).*255.0;

% An RGB to YCbCr color space conversion ( color specification )
colorT = [0.299 0.587 0.114; -0.169 0.334 0.500; 0.500 0.419 0.081];
bias = [0; 128; 128];
imgYCbCr = zeros(size(imgRGB));
for r = 1:size(imgRGB, 1)
    for c = 1:size(imgRGB, 2)
        vRGB = reshape(imgRGB(r, c, :), [3,1]);
        vYCbCr = colorT*vRGB+bias;
        imgYCbCr(r, c, :) = vYCbCr;
    end
end

% Original image is divided into blocks of 8 x 8.
% The pixel values within each block range from[-128 to 127] but pixel
values of a black and white image range from [0-255] so, each block is
shifted from[0-255] to [-128 to 127].
imgYCbCr = imgYCbCr - 128.0;

% The DCT works from left to right, top to bottom thereby it is applied to
each block.
imgDCT = zeros(size(imgYCbCr));
for r = 1:8:size(imgYCbCr, 1)
    for c = 1:8:size(imgYCbCr, 2)
        for k = 1:size(imgYCbCr, 3)
            vBlock = reshape(imgYCbCr(r:r+7, c:c+7, k), [8, 8]);
            imgDCT(r:r+7, c:c+7, k) = dct2(vBlock);
        end
    end
end

% Each block is compressed through quantization.
Qr = [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];
% Qr(:)=100;
```

```
Qc = [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];
% Qc(:)=100;
imgQDCT = zeros(size(imgDCT));
for r = 1:8:size(imgDCT, 1)
    for c = 1:8:size(imgDCT, 2)
%       luminance
        k = 1;
        vBlock = reshape(imgDCT(r:r+7, c:c+7, k), [8, 8]);
        imgQDCT(r:r+7, c:c+7, k) = round(vBlock./Qr);
%       chrominance(Yb)
        k = 2;
        vBlock = reshape(imgDCT(r:r+7, c:c+7, k), [8, 8]);
        imgQDCT(r:r+7, c:c+7, k) = round(vBlock./Qc);
%       chrominance(Yr)
        k = 3;
        vBlock = reshape(imgDCT(r:r+7, c:c+7, k), [8, 8]);
        imgQDCT(r:r+7, c:c+7, k) = round(vBlock./Qc);
    end
end

% Huffman Coding Simulation
data = reshape((imgQDCT), [1, 256*256*3]);
[N, symbols] = hist(data,double(unique(data)));
p = N / (256*256*3);

[dict, avglen] = huffmandict(symbols,p);

lens = zeros(size(symbols));
for v = 1:length(symbols)
    len = length(cell2mat(dict(v,2,:)));
    lens(v) = len;
end

% Result of compression
code = huffmanenco(data,dict);
```

```
% bar(symbols, N);
Huffman_Coding_N = sum(lens.*N, "all");
Huffman_Coding_P = sum(lens.*p, "all");

%==========================================
% Reconstruction==========================
%==========================================

% Inverse of Huffman Coding
dataInverse = huffmandeco(code,dict);
dataInverse = reshape((dataInverse), [256, 256, 3]);

% Inverse of quantization.
imgQDCTInverse = zeros(size(dataInverse));
for r = 1:8:size(dataInverse, 1)
    for c = 1:8:size(dataInverse, 2)
%       luminance
        k = 1;
        vBlock = reshape(dataInverse(r:r+7, c:c+7, k), [8, 8]);
        imgQDCTInverse(r:r+7, c:c+7, k) = round(vBlock.*Qr);
%       chrominance(Yb)
        k = 2;
        vBlock = reshape(dataInverse(r:r+7, c:c+7, k), [8, 8]);
        imgQDCTInverse(r:r+7, c:c+7, k) = round(vBlock.*Qc);
%       chrominance(Yr)
        k = 3;
        vBlock = reshape(dataInverse(r:r+7, c:c+7, k), [8, 8]);
        imgQDCTInverse(r:r+7, c:c+7, k) = round(vBlock.*Qc);
    end
end


% Inverse of DCT works from left to right, top to bottom thereby it is
applied to each block.
imgDCTInverse = zeros(size(imgQDCTInverse));
for r = 1:8:size(imgQDCTInverse, 1)
    for c = 1:8:size(imgQDCTInverse, 2)
        for k = 1:size(imgQDCTInverse, 3)
```

```
            vBlock = reshape(imgQDCTInverse(r:r+7, c:c+7, k), [8, 8]);
            imgDCTInverse(r:r+7, c:c+7, k) = idct2(vBlock);
        end
    end
end

% Original image is divided into blocks of 8 x 8.
% Inverse of pixel values within each block range from[-128 to 127] but
pixel values of a black and white image range from [0-255] so, each block
is shifted from[0-255] to [-128 to 127].
imgYCbCrInverse = imgDCTInverse + 128.0;

% An YCbCr to RGB color space conversion ( color specification )
imgRGBInverse = zeros(size(imgYCbCrInverse));
for r = 1:size(imgYCbCrInverse, 1)
    for c = 1:size(imgYCbCrInverse, 2)
        vYCbCr = reshape(imgYCbCrInverse(r, c, :), [3,1]);
        vRGB = colorT^(-1)*(vYCbCr - bias);
        imgRGBInverse(r, c, :) = vRGB;
    end
end

% imwrite(uint8(imgRGBInverse),"R.png")
imshow(uint8(imgRGBInverse))
```