CMPT365 – Project 2

Question 1:

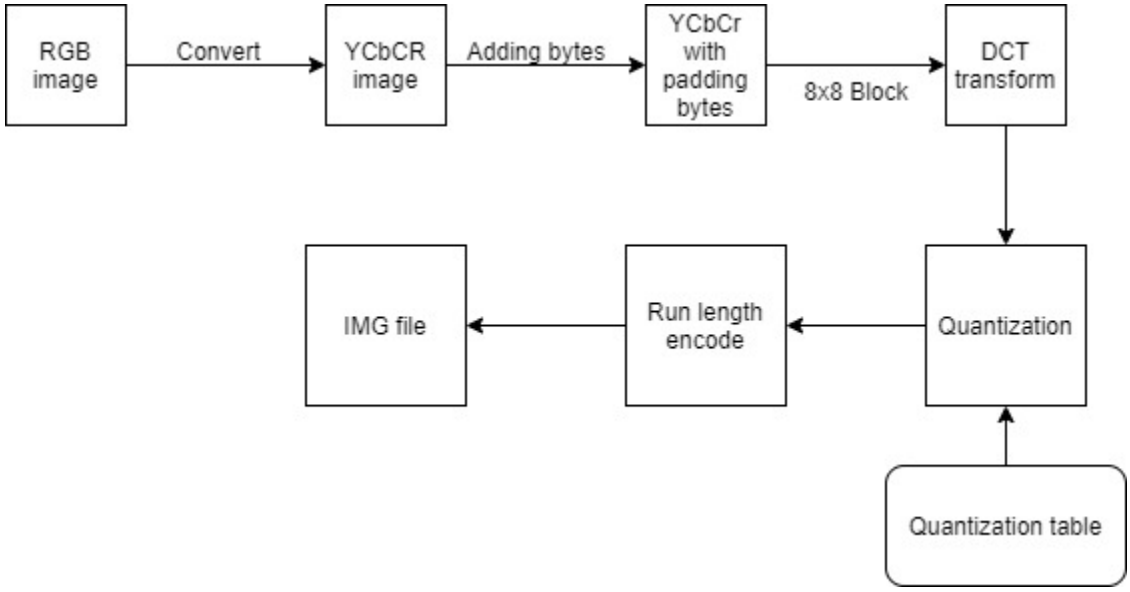
- I managed to implement Huffman and LZW encoding that can achieve roughly from 1.2 to 1.6 compression ratio for the sample inputs. I also encoded both header data and sample data. For each header data, I treated one byte as one symbol and for samples data I treated 1 sample (2 bytes) as 1 symbol
- The compression ratio is computed as:

$$\text{Ratio} = \frac{Number\ of\ bits\ to\ represent\ original\ file}{Number\ of\ bits\ of\ encoded file}$$

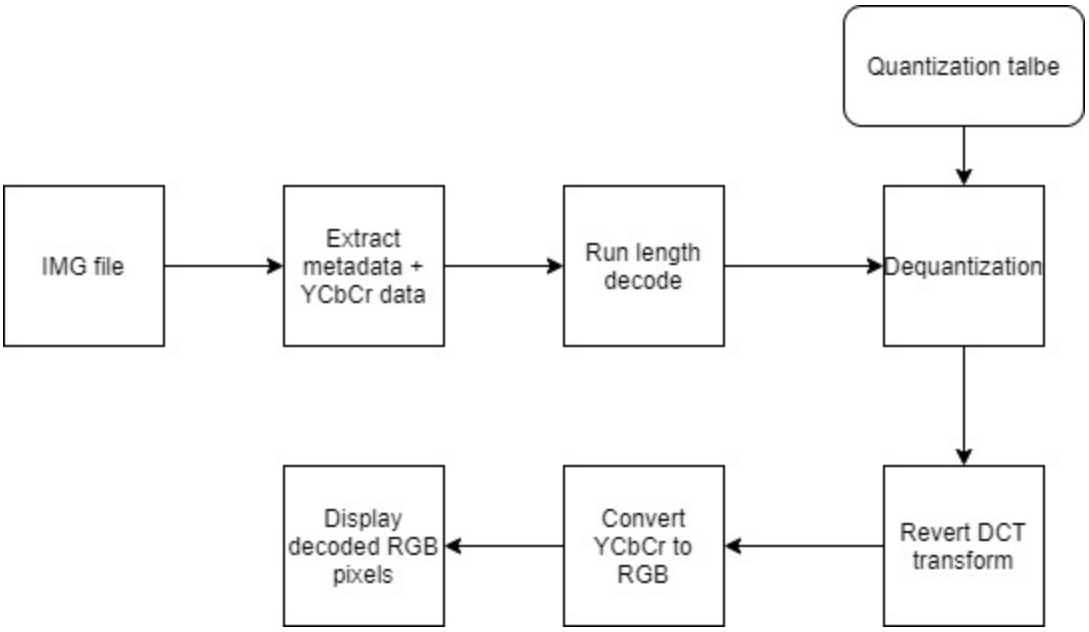
- Compared with FLAC coding, this compression ratios that I achieved are quite lower than FLAC's compressions. The reason is that what I did was just to apply normal Huffman and LZW encoding technique while FLAC coding includes many sophisticated techniques to compress an audio file. For example, FLAC always tries to predict a mathematical description of the audio signal so that it can obtain a better compression pattern. When a WAV file is compressed by FLAC, it only encodes the 'fmt' and 'data' sub-chunks of WAV file so FLAC can avoid encoding unnecessary data from WAV, therefore it can achieve a better compression ratio.

Question 2:

1. IMG encode compression flow diagram



2. IMG decode compression flow diagram



3. Encode and decode performance:

File	Encode/Compression (seconds)	Decode/Uncompression (seconds)	PSNR	Compression ratio
BIOS.bmp	3.19	3.09	27.54 dB	3.91
earth.bmp	1.68	1.60	29.80 dB	3.74
Fall.bmp	3.68	3.58	24.42 dB	2.38
nature.bmp	1.78	1.72	26.43 dB	2.54
nature_2.bmp	3.21	3.13	23.65 dB	2.59

- The performance is much slower than JPEG compression but it is not too bad for this project in my opinion because the quality of compressed images are quite close to scale 4 or 5 of in Photoshop JPEG compression.
- The first two images have high compression ratio is because they contain lots of duplicated colors in consecutive pixels so the Run length encoding does a good job of encoding/compressing them using very few bits.

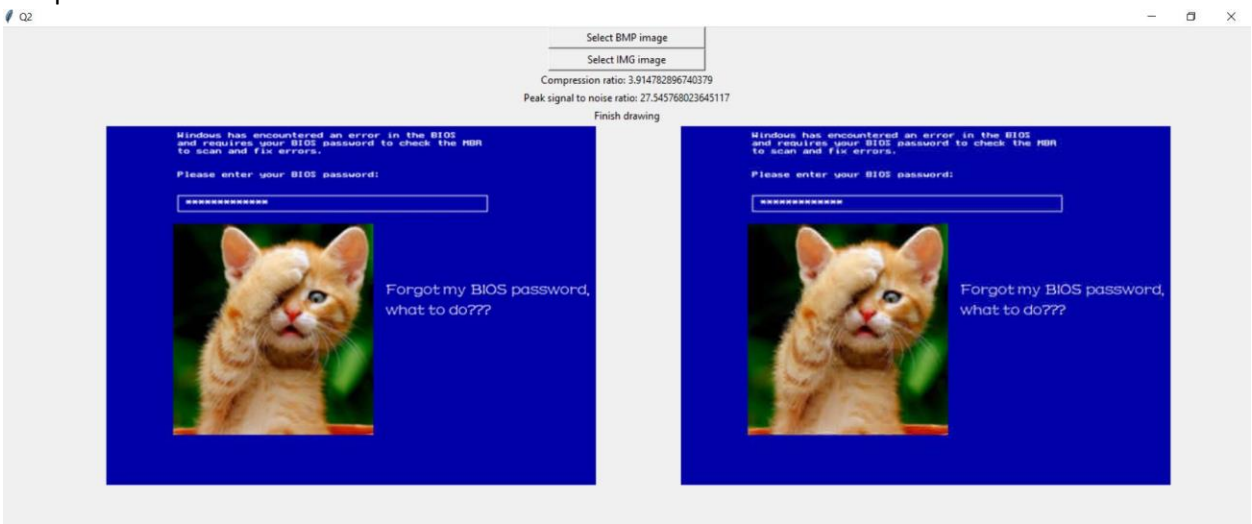
4. Discussion

- Quantization table that is used for quantization step have a big impact on the compression result because quantization is when the loss occurs. With good quantization table, we can achieve a very good compression ratio without losing too much data and therefore the quality of compressed image is maintained.
- Implementing entropy encoding algorithm is also quite challenging. With LZW, it is simple for implementation but it might not work for big images because LZW needs a memory to maintain a concatenated string of symbols, therefore, there might not be enough memory for that with big images. Huffman coding is often the choice for image compression, however, we need to store Huffman table in compressed images and it also complicates the source code. I decided to

use another encoding technique called Run Length Encoding which simply works by record the number of times that a symbol is repeated consecutively. This version of Run Length Encoding is a bit different from the one mentioned in the book which actually records number of zeros that stands in front of a non-zero number. I found that this encoding works quite well to encode small or medium size images and it is straightforward to implement both encoding and decoding.

- One of tricky part when I implemented DCT transformation for encoding images is that I had to reduce the range of values for each color channel because otherwise I will get values that requires more than 1 byte to represent, meaning that the size of compressed images could be even larger then the original ones. I decided to subtract every value by 128 so that the range for every value is only from -128 to 127, which can be well represented by 1 byte.
- Another problem that I encountered is the performance of my program. Using third-party libraries would improve the performance a lot but I decided to implement everything from scratch with only built-in libraries of python3. I realized that the bottleneck of the program is to perform DCT transformation and to draw the image pixel by pixel. DCT transformation involves matrix multiplication of three 8x8 block and there are 3 channels, each channel has a size of widthxheight so the total time complexity of DCT transformation is: O($8^3 * \frac{3*width*height}{8*8}$) = O(24*width*height). In order to improve the speed for this operation, I can use a more cache-friendly data structure like numpy array instead of using normal python list which often cause cache miss.
  Drawing images pixel by pixel is the most expensive because of tkinter's not-so-good performance on drawing images.

5. Sample screenshots:




Q2
Select BMP image
Select IMG image
Compression ratio: 3.740957568408393
Peak signal to noise ratio: 29.80233407305399
Finish drawing


Select BMP image
Select IMG image
Compression ratio:
Peak signal to noise ratio:
Finish drawing