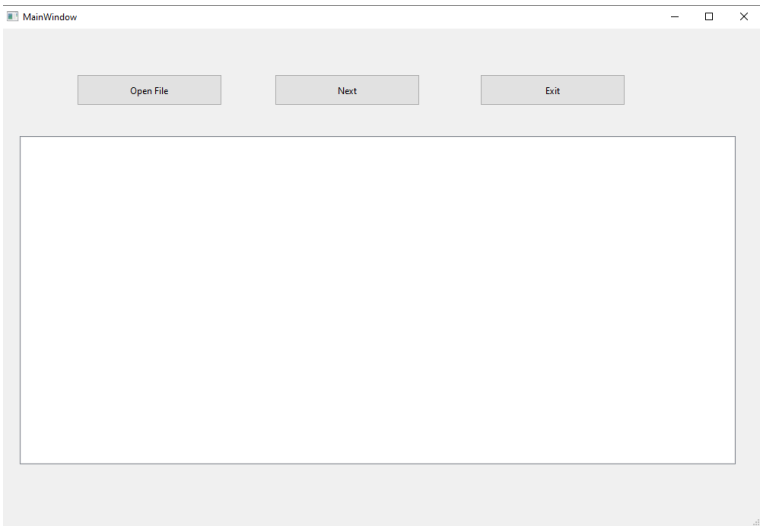
365 Project2

Marco Hiu Yeung Lai

301356237

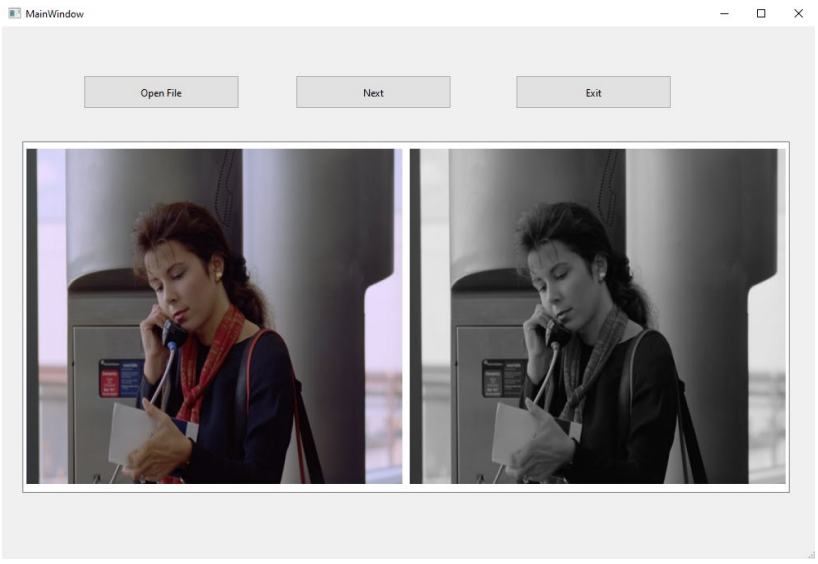
Q1

For project 2 I reuse the code from project 1 which using the qt creator and write in C++. Since it requires the “next” button to show different stage of image processing, I implemented the code within the pushButton_clicked and add a variable of currentStep to determine which stage of image to show.



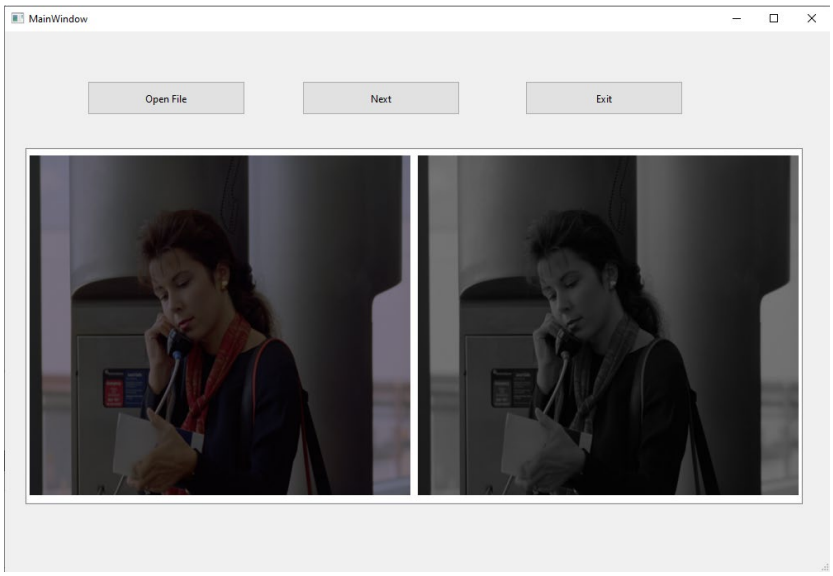
Step 1:

I convert the image to grayscale by simply create a new copy of the original image and loop through the width and height pixels and change its corresponding red, green, and blue pixel colors to gray value using qGray, below is the result of test sample 1.



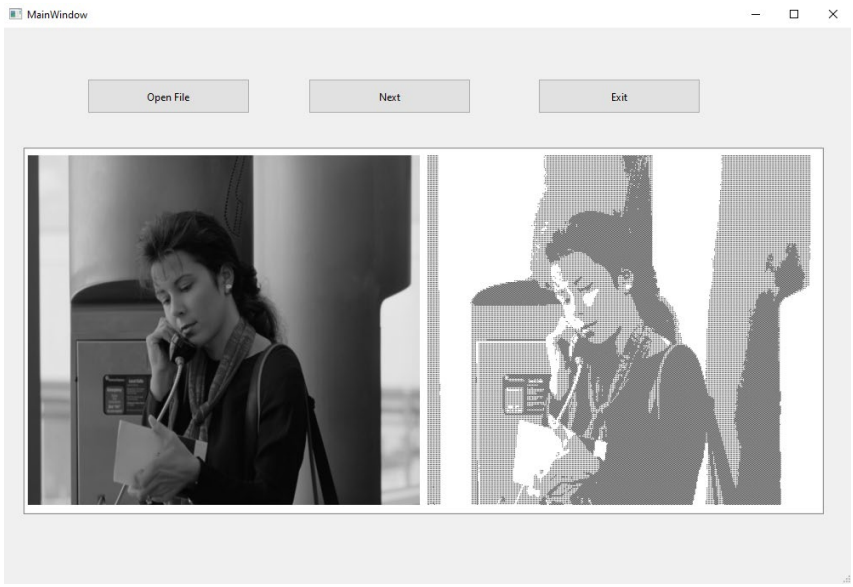
Step 2:

I loop through the pixel and changes its corresponding RGB value by multiplying 0.5. However, when I applied the same technique on gray image it turns out that the image became all white due to the gray value digits collapsed. The simplest way to solve the problem is to convert the new gray image from the 50% brightness image generated, below is the result:



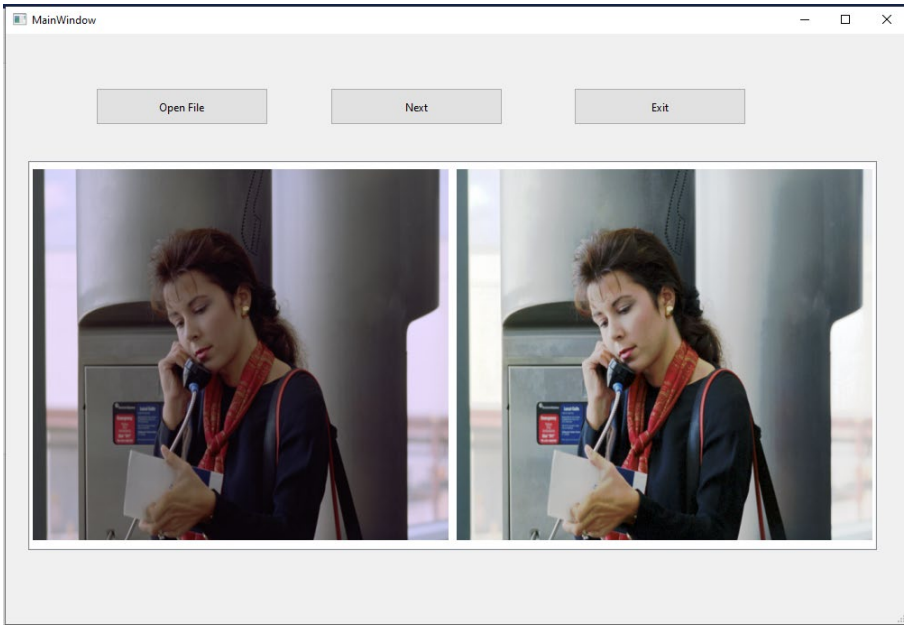
Step 3:

The dithering method I used here same as the lecture slides, I iterates through each pixel, convert to gray value and re-map the values by dividing by 256/5, then I obtain the dither value from the matrix [y%2][x%2] which determines the threshold. Last, I compare if mapValue + ditherValue to 1 and change the pixel in ditherGray to 1 if the result is greater than 1, change to 0 otherwise. I tried both 2X2 and 4X4 dither matrix and it seems 2X2 generate better result:



Step 4:

I first iterate the pixels and count the RGB intensity level. Then I calculate the cumulative distribution function for each color channel, track the cumulative sum of pixel counts with intensity values up to the current level. Last, I apply the contrast enhancement and histogram equalization to each pixel and adjust the color based on the cdf values, therefore improve the image's overall contrast:



Q2

Entropy:

The entropy is the negative of sum of the P_i * log_2 (P_i) where P is the probability of the corresponding sample frequency / the sum of the samples, i from 1 to N samples. The audio data is iterated to count its frequency and store into the map sampleCount. Then the sampleCount is iterated and calculate the corresponding probability by dividing the total samples * number of channels, then entropy is stored in global variable by -= probability * log_2 (probability).

Average code word length:

Since the course lectures did not cover much coding techniques about Huffman coding, I browse online and complete this part of project from online tutorials.
References: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

I rewrite my own version of code by using the min heap approach above to build the Huffman tree. Below is the struct to hold the Huffman node.

```
struct HuffmanNode {
    int16_t sample;
    int frequency;
    HuffmanNode* left;
    HuffmanNode* right;
};
```

Implementation description: the sampleCount generated previously is pass into the build Huffman tree function. Priority queue is created and store the samples and its frequency as the format of Huffman Node struct and compare the nodes with its frequency as minheap. The Huffman tree is built by repeatedly merging the nodes where the Huffman code is generated by transversing the tree root, each left child adds a code “0” and right child adds a code ”1” and therefore is stored in the map huffmanCodes. Last, the average code word length is calculated by iterating the sampleCount, multiply its frequency by that of corresponding Huffman code length matched with the map huffmanCodes, and divided by the total samples.

Below is the test case data:

```
//testing~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
sampleCount.clear();
sampleCount[0] = 5;
sampleCount[1] = 9;
sampleCount[2] = 12;
sampleCount[3] = 13;
sampleCount[4] = 16;
sampleCount[5] = 45;
numChannels = 1;
totalSamples = 100;
```

```
entropy =  2.21988
averageCodeLength =  2.24
```

==========Manuel calculation==========

Probablity[0] = 5/100

Probablity[1] = 9/100

Probablity[2] = 12/100

Probablity[3] = 13/100

Probablity[4] = 16/100

Probablity[5] = 45/100

Entropy = -[5/100 log_2 (5/100) + 9/100 log_2 (9/100) + 12/100 log_2 (12/100) + 13/100 log_2 (13/100) + 16/100 log_2 (16/100) + 45/100 log_2 (45/100)]

Entropy = 2.21988 matched

Huffman code-word:

Sample5: 0

Sample2: 100

Sample3: 101

Sample0: 1100

Sample1: 1101

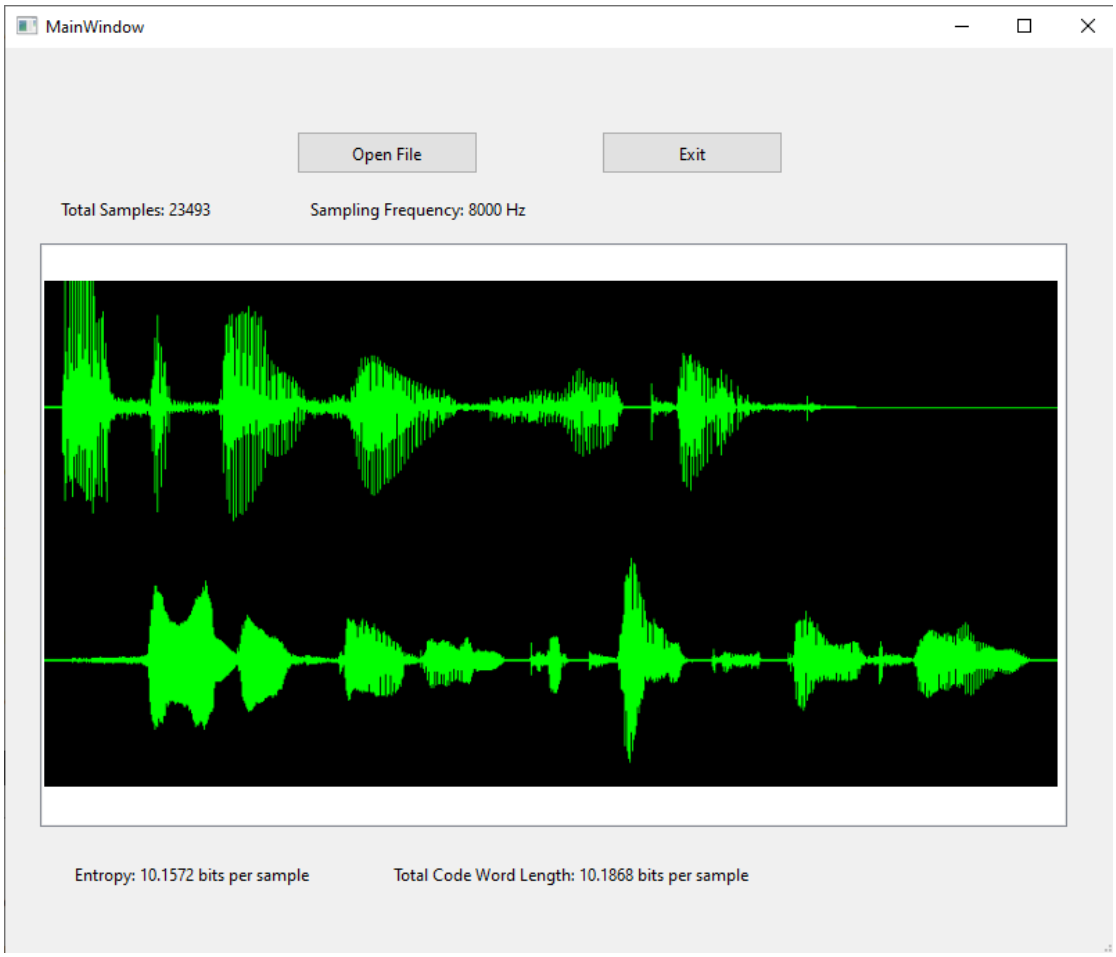
Sample4: 111

Average Code Length = 5/100 * 4 + 9/100 * 4 + 12/100 * 3 + 13/100 * 3 + 16/100 * 3 + 45/100 * 1 = 2.24

Average Code Length = 2.24 matched

==========Manuel calculation ends==========

Test Case from audio1.wav



Entropy is 10.1572 pits pers sample while the average code word length is 10.1868 bits per sample which make sense since their difference is not greater than 1.