Project2 Report

Rongsheng Qian 301449387

Overall:

Folder Structure:

```
.
├── Huffman_Tree.class
├── Huffman_Tree.java     // implement HuffmanTree
├── file_chooser.java   // main file which is a launcher with GUI
├── read_tif.class
├── read_tif.java       // implement all api which is required in Q1
├── read_wave.class
└── read_wave.java      // invoke function in HuffmanTree calculate entropy & avg code length
```

Run:

```
java file_chooser.java
```

Or:

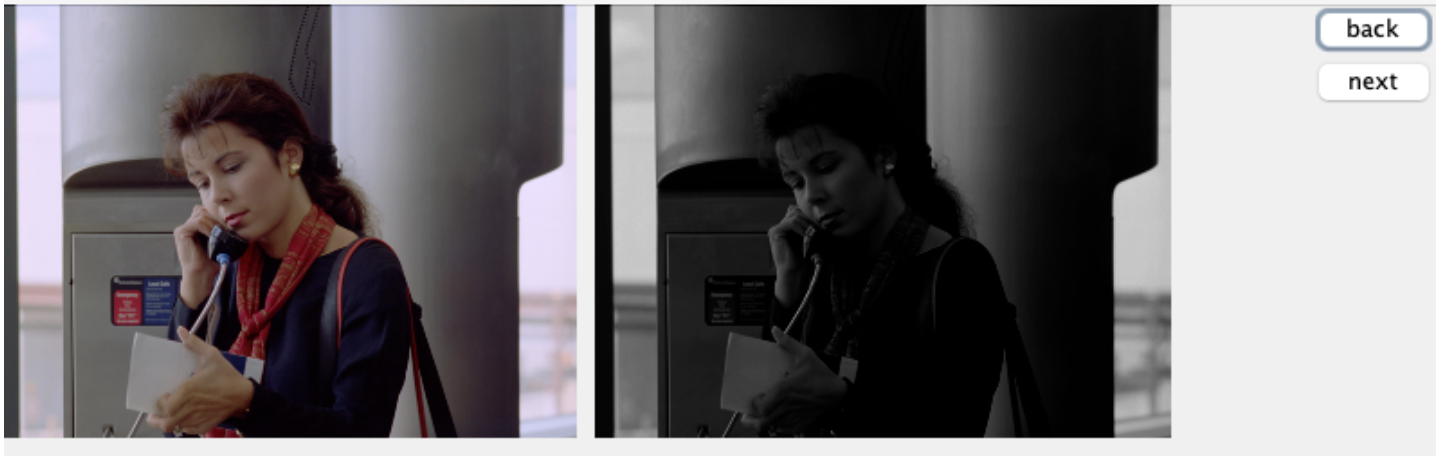
Using Q1.jar & Q2.jar for Part 1&2

Q1.

Step 1:

For the greyscale function, I used `grey = 0.3*r + 0.59*g + 0.11*b` to convert RGB image to grey image.

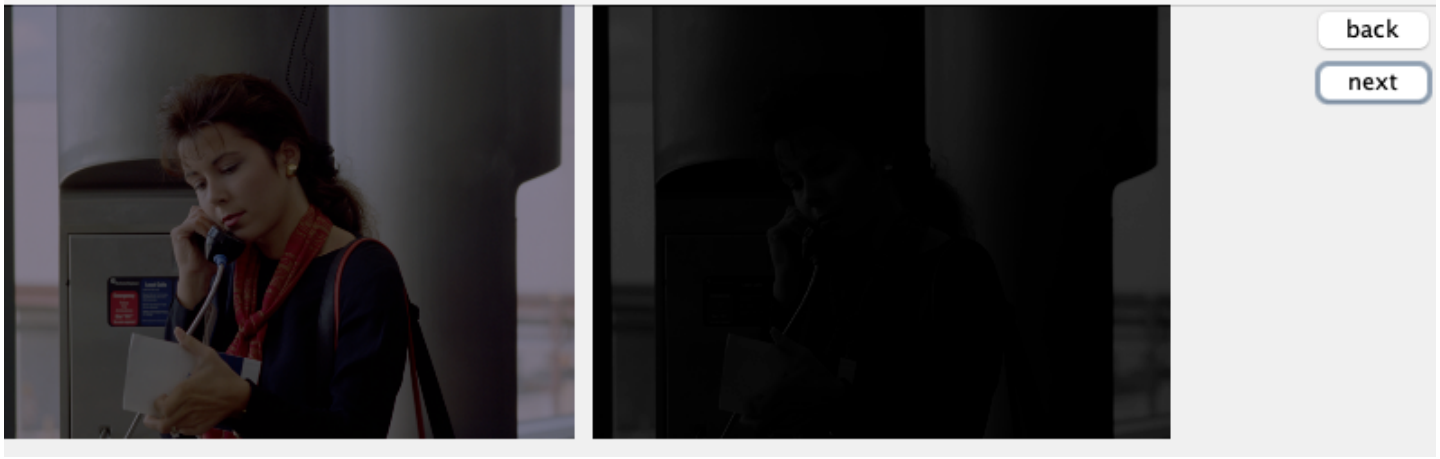
I have used this rather than roughly avg (r,g,b) by `grey = (r + g + b)/3` because it provides a better approximation of how humans perceive grayscale.

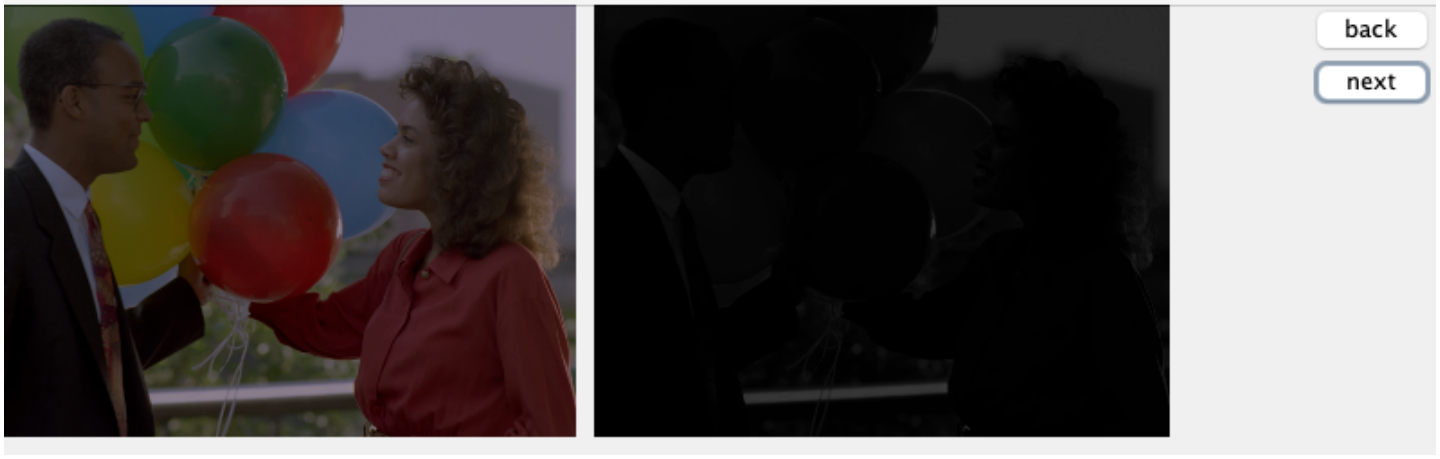


Step 2:

For step 2, I just simply multiply each RGB values by 0.5 to reducing the brightness to 50% for the original colored image (left) and for the grayscale image (right), And get the result below,

```
r = (int)(r * 0.5);
g = (int)(g * 0.5);
b = (int)(b * 0.5);
int result = (r << 16) | (g << 8) | b;
```



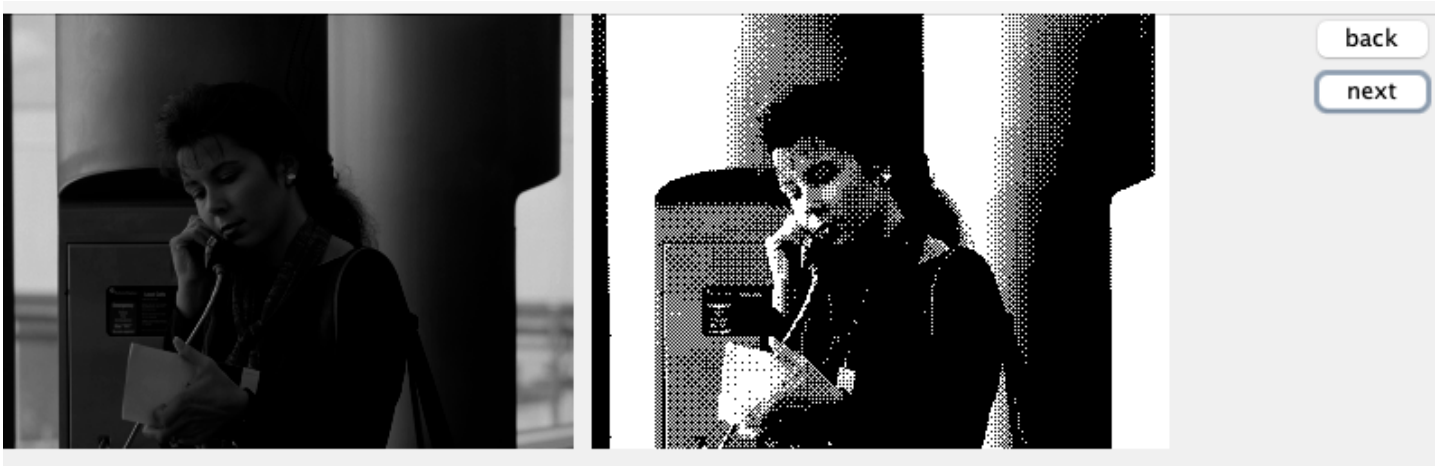


Step 3:

I have used 8x8 Bayer dither matrix to calculate my ordered dithering on the grayscale image which looks like below:

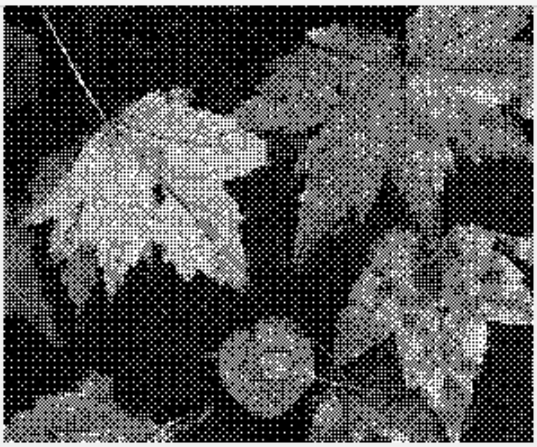
```
int[][] mat = {
                {0, 32, 8, 40, 2, 34, 10, 42},
                {48, 16, 56, 24, 50, 18, 58, 26},
                {12, 44, 4, 36, 14, 46, 6, 38},
                {60, 28, 52, 20, 62, 30, 54, 22},
                {3, 35, 11, 43, 1, 33, 9, 41},
                {51, 19, 59, 27, 49, 17, 57, 25},
                {15, 47, 7, 39, 13, 45, 5, 37},
                {63, 31, 55, 23, 61, 29, 53, 21}
        };
```

The reson why I have choosen 8x8 dimension because the clarity of the display has been significantly improved from `2x2` to `8x8`. The results from different dimensions is shown below: (first is `2x2`, second is `4x4`, third is `8x8`).



The final results is shown below: (I didn't choose 16x16 because there is barely improvement for given samples)


back
next

back
next


back
next

Step 4:

For applying auto level on the original colored image, I used CDF to map the color value in a new color value distribution.

Secondly, I split the whole color value interval into 256 levels (256 pieces) and calculate the CDF based on these 256 levels for each color channel.

Finally, I calculate the new value by `new_value = CDF*(level-1)=CDF*255` for each pixel in each channel, which may assign them into a new level.

The pseudo-code is given below:

```
// indexs in double[] represent the levels
double[] calculateCDF(BufferedImage image,char channel)
// calculate CDF for each channel
double[] cdf_r = calculateCDF(image,'r');
double[] cdf_b = calculateCDF(image,'b');
double[] cdf_g = calculateCDF(image,'g');
// new value
r = (int)(cdf_r[r] * 255.0);
g = (int)(cdf_g[g] * 255.0);
b = (int)(cdf_b[b] * 255.0);
```

The result is shown below:

Q2

Overall result:

audio1

audio2

Implementation Detail:

Step 1:

Calculating the entropy of the audio samples, the pseudo-code is below:

1. **store the frequency for each value**
2. **Calculate prob and entropy by frequency table**

```
Map<Integer, Integer> table = new HashMap<>(); // Table to store the frequency of occurrence of each value

int value = ...; // int value is read from .wav

if (table.containsKey(value)) {
  table.put(value, table.get(value) + 1);
```

```
} else {
  table.put(value, 1);
}

// for each value calculate prob and then calculate entropy
for (int sample : table.keySet()) {
  double probability = (double) table.get(sample) / sample_num_total;
  entropy -= probability * (Math.log(probability) / Math.log(2)); //
}
```

Step 2:

Calculating the average code word length, the steps are below:

1. Create Huffan tree (binary tree)

```
public class Tree {
    private Node root;}
private static class Node {
        int value;
        int frequency;
        Node left;
        Node right;}
```

2. add freq table into nodes pool, sort them and apply huffman code algorithm

```
// generate the node_pool by the freq_table before
// add all of them into pool
// ......
Node right = node_pool.remove(0)
Node left = node_pool.remove(0)
Node interval = merge(right,left)
node_pool.add(interval)
```

3. Encode the huffman code by recursive function:

```
node.left = code + '0'
node.right = code + '1'
```

4. Calculate the avg by the formula

$$\frac{sampleCount*code.length}{totalSampleCount} = \frac{totalCode.length}{totalSampleCount}$$