

Project2 Report

Rongsheng Qian

Overall:

Folder Structure:

```
1 | .
2 | └─ Huffman_Tree.class
3 | └─ Huffman_Tree.java      // implement HuffmanTree
4 | └─ file_chooser.java      // main file which is a launcher with GUI
5 | └─ read_tif.class
6 | └─ read_tif.java          // implement all api which is required in Q1
7 | └─ read_wave.class
8 | └─ read_wave.java         // invoke function in HuffmanTree calculate entropy & avg code length
```

Run:

```
1 | java file_chooser.java
```

Or:

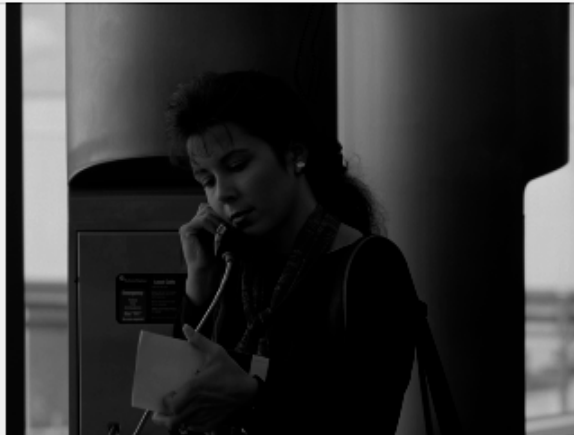
Using Q1.jar & Q2.jar for Part 1&2

Q1.

Step 1:

For the greyscale function, I used $\text{grey} = 0.3*r + 0.59*g + 0.11*b$ to convert RGB image to grey image.

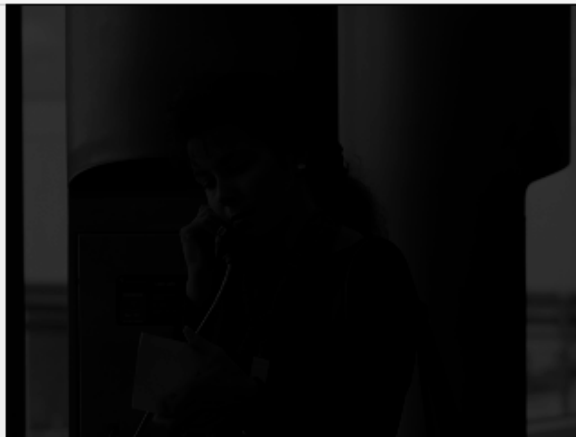
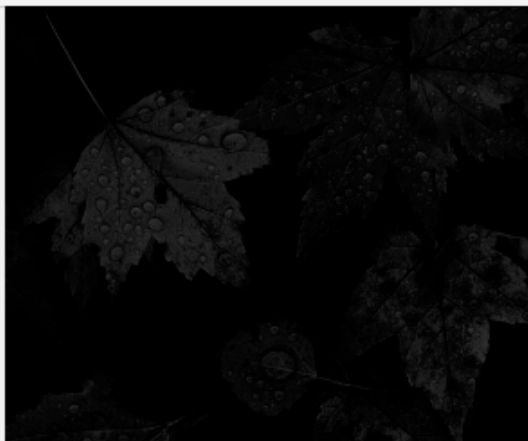
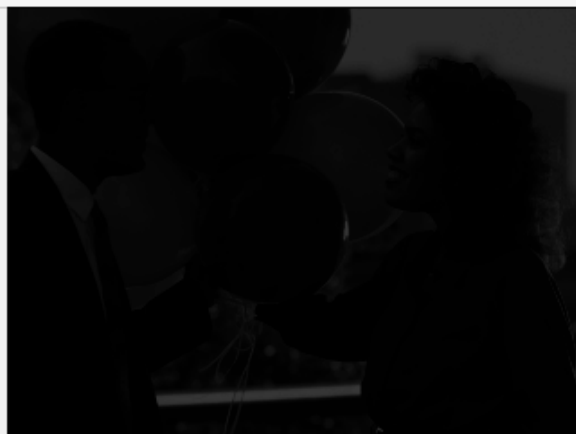
I have used this rather than roughly avg (r,g,b) by $\text{grey} = (r + g + b)/3$ because it provides a better approximation of how humans perceive grayscale.

[back](#)[next](#)[back](#)[next](#)[back](#)[next](#)

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,

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

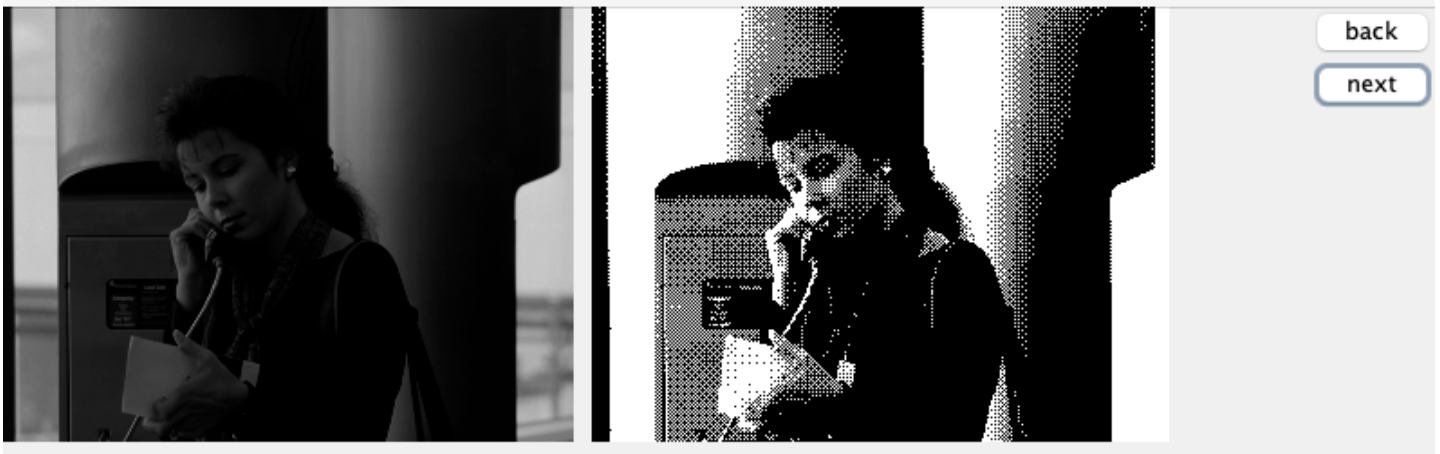
[back](#)[next](#)[back](#)[next](#)[back](#)[next](#)

Step 3:

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

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

The reason why I have chosen 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:

```
1 // indexs in double[] represent the levels
2 double[] calculateCDF(BufferedImage image,char channel)
3 // calculate CDF for each channel
4 double[] cdf_r = calculateCDF(image,'r');
5 double[] cdf_b = calculateCDF(image,'b');
6 double[] cdf_g = calculateCDF(image,'g');
7 // new value
8 r = (int)(cdf_r[r] * 255.0);
9 g = (int)(cdf_g[g] * 255.0);
10 b = (int)(cdf_b[b] * 255.0);
```

The result is shown below:





Q2

Overall result:

audio1



Project 2

the entropy of the audio samples : 10.15701835024495

back

the average code word length: 10.186744423633577 bits

audio2



Project 2

the entropy of the audio samples : 9.143062572317712

back

the average code word length: 9.160709977597794 bits

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

```
1  Map<Integer, Integer> table = new HashMap<>(); // Table to store the frequency of occurrence of
   each value
2
3  int value = ...; // int value is read from .wav
4
5  if (table.containsKey(value)) {
6      table.put(value, table.get(value) + 1);
```

```

7  } else {
8      table.put(value, 1);
9  }
10
11 // for each value calculate prob and then calculate entropy
12 for (int sample : table.keySet()) {
13     double probability = (double) table.get(sample) / sample_num_total;
14     entropy -= probability * (Math.log(probability) / Math.log(2)); //
15 }

```

Step 2:

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

1. Create Huffman tree (binary tree)

```

1  public class Tree {
2      private Node root;
3      private static class Node {
4          int value;
5          int frequency;
6          Node left;
7          Node right;
8      }
9  }

```

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

```

1  // generate the node_pool by the freq_table before
2  // add all of them into pool
3  // .....
4  Node right = node_pool.remove(0)
5  Node left = node_pool.remove(0)
6  Node interval = merge(right, left)
7  node_pool.add(interval)

```

3. Encode the huffman code by recursive function:

```

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

```

4. Calculate the avg by the formula

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