**Multimedia Technologies**

**Project Report**

## Members

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **序号** | **学号** | **专业班级** | **姓名** | **性别** |
|  | 3200300849 | 计算机科学与技术 | RAYMOND SIDHARTA 徐祥龙 | 男 |

## Project Introduction

* 1. Background

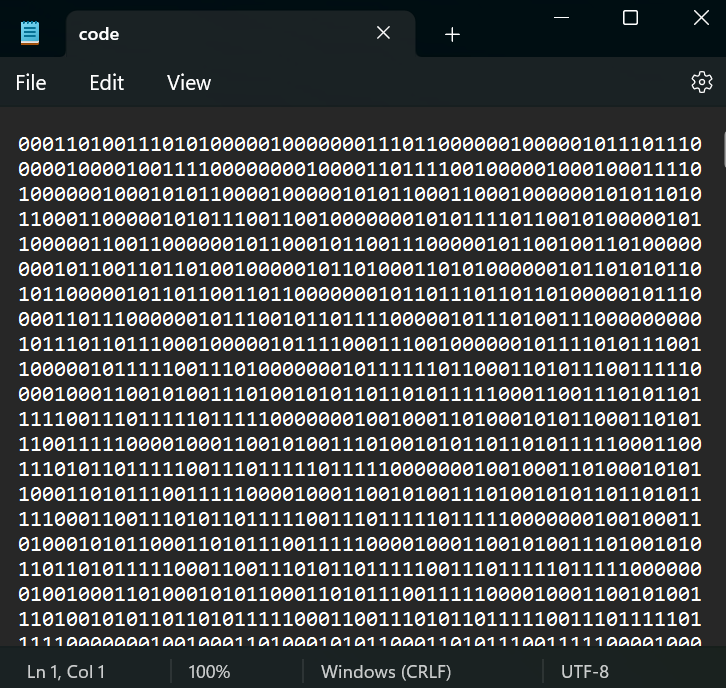
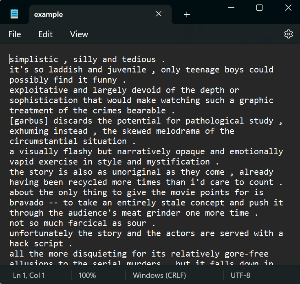
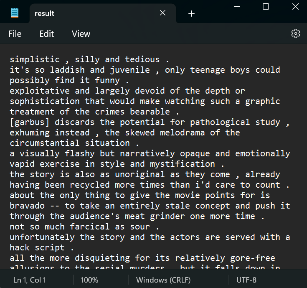
The term *multimedia* comes from 2 words, *multi* and *media*. Multi means many or varying, while media or medium could mean method. Therefore, we can derive multimedia meaning as many and varying methods to represent information to the user, also as a medium of communication. Multimedia provides an attractive and computerized method to represent some information, with the combination of textual data, audio, visual, graphics, animations, etc. By using more than one media, information can be represented in a powerful manner, giving the user deeper comprehensive level than just reading raw text.

The implementation of multimedia nowadays become more massive. It’s been used almost in all sectors of life. For example, in business, multimedia technology provides some facilities such as voice mail, video teleconferencing, animation or video transition for presentation to boost marketing, and other cooperative work environments that can help employees work from anywhere. In entertainment, many computer games nowadays have high definitions graphics that provide a magnificent experience for gamers. Even in education, combining images and interactive animations can enhance student’s perspective about the knowledge, make the information easier to be digested.

Realizing that multimedia plays an important role in present life, this project comes to expand our knowledge and concepts about multimedia and how to learn it by doing practical programming. This project will be focused on text and image compression with JPEG algorithm. We implement JPEG algorithm since it still be commonly used until right now.

* 1. Goals

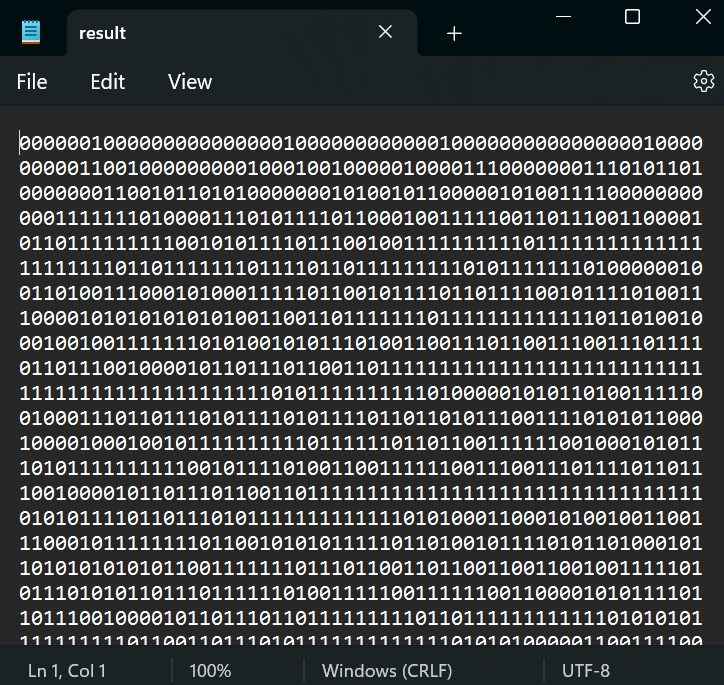
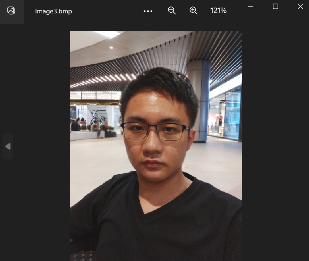
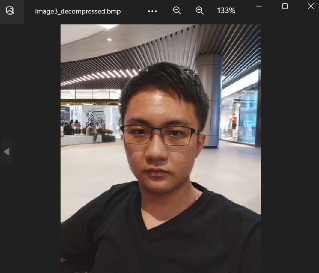
In this project, we shall accomplish 2 tasks. First task is to implement text compression using Huffman coding. Text compression will be done in lossless method.



Encode

Decode

Second task is to implement image compression, using the compression algorithms used in JPEG image. These algorithms consist either lossy and lossless algorithm.



Encode

Decode

* 1. Development Environment

We use Python 3.11.0 as our programming language, Python’s *numpy* to deal with multi-dimension matrices, and Python *cv2* for dealing with images file. Some part of our program also uses Python *multiprocessing* to boost compressing time.

## Technical Details

* 1. Huffman Coding

Huffman coding is a lossless data compression algorithm. It was proposed by David A. Huffman in 1950 and is widely used in text or image compression especially for JPEG (we will explain JPEG soon).

Suppose we deal with a text file. We know that each character is actually a sequence of 0’s and 1’s and stores using 8-bits or 1-byte. This mechanism is called *fixed-length encoding*, because each character uses the same number of fixed-bit storage.

The problem is, can we represent each of those characters with shorter bits, in order to reduce the amount of space required to store the text or save the bandwidth if we want to transmit the data? With Huffman coding, we can do all of these tasks! Huffman coding is so called *variable-length encoding* since each character is represented with different length of bits. We exploit some characters that occur more frequently in comparison to others. The more frequent it occurs, the shorter its bits length, vice versa.

Let’s take an example. Suppose we have a text consists of just 4 characters: a, b, c, d. If we are too greedy, with Huffman we can just represent those characters with bits as short as we can, e.g. a = 0; b = 1; c = 10; d = 01. Seems like we are done, but the problem occurs when we want to decode the bits. Bit stream 0010 can be decoded as aaba, aac, and ada. This leads to ambiguity.

To solve this problem, Huffman coding obeys prefix rule to ensures that the code assigned to the character is not treated as a prefix of the code assigned to any other character. We can assure the prefix rule by creating Huffman tree from input characters. Here is the pseudo-code for Huffman coding algorithm.

**procedure** HuffmanTree (*t*):

*t* is a text file

*d* is python dictionary with format {char : frequency}

*q* is priority queue (could be min heap)

*q*’s item could be integer, could also be a tree node

Tree node consists of value and its left and right child

while *t* != EOF:

read char *c*

*d[c]* += 1

sort *d* by its frequency in increasing order, push all frequencies to *q*

while *q* is not empty:

pop 2 smallest items *i1* and *i2* which has value *v1, v2* from queue

create tree node: value = *v1* + *v2*, left\_child = *i1*, right\_child = *i2*

push that node into *q*

**return** first element of *q*, which is our Huffman tree’s root node

**procedure** HuffmanDict (*root*, *binString* = ‘’):

*root* is tree node, represents Huffman tree’s root

*d* is python dictionary with format {char : binaryString}

if root is a char:

return {root : binString}

update *d* by running HuffmanDict(*root->left*, *binString* += ‘0’)

update *d* by running HuffmanDict(*root->right*, *binString* += ‘1’)

**return** d

In this project, Huffman coding is implemented in both text and image compression.

* 1. JPEG Algorithm

JPEG stands for Joint Photographic Experts Group, an international organization that standardized the format during the late 1980s and early 1990s. It’s a standard image format for containing lossy and compressed image data. Despite the huge reduction in file size, JPEG images maintain reasonable image quality.

The main idea of JPEG compression is we analyze each section of the image and finds and removes elements which eyes can’t easily perceived.

First, it exploits human eyes weakness from recognizing colors. It happens because human eyes have more light receptors (rod cells) than color receptors (cone cells). In other words, our eyes are more sensitive to brightness levels than chrominance levels. The JPEG compression uses this weakness by throwing some color information from images in order to save stored data.

Second, it exploits human eyes weakness from recognizing high frequency change in some part or section of the image. JPEG compression uses this weakness by removing those data which have high frequency change since we missed the details in our image. That’s why JPEG may reduce our image quality, but not that much.

These are 5 common steps for JPEG compression:

* + 1. Color Space Conversion

Image commonly be represented using *RGB* color space. To implement JPEG compression, first we must convert *RGB* color space into *YCbCr* color space to separate image luminance component *Y* from color or chrominance component *Cb* (chrominance blue) and *Cr* (chrominance red).

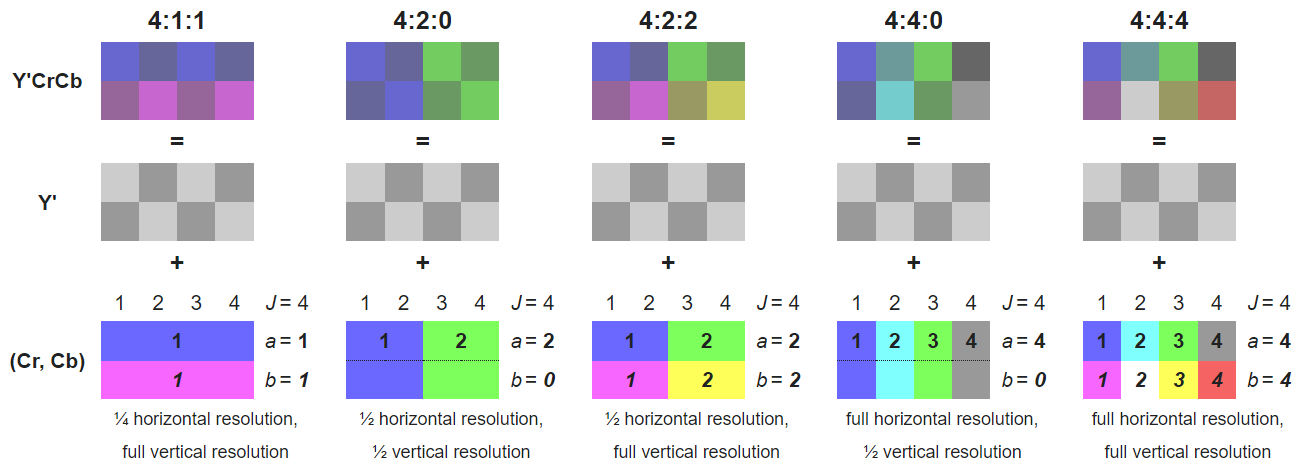
Matrix to convert RGB -> YCbCr

Matrix to convert YCbCr -> RGB

* + 1. Chrominance Down-sampling

Chroma subsampling is the practice of encoding images by implementing less resolution for chroma information than for luminance information, taking advantage of the human visual system's lower acuity for color differences than for luminance. Therefore, it is a lossy compression method.

The subsampling scheme is commonly expressed as a three-part ratio *J:a:b*, where *J* is horizontal sampling reference (width of the conceptual region, usually 4); *a* is number of chrominance samples in the first row of J pixels; *b* is number of changes of chrominance samples between first and second row of *J* pixels.



Source: *https://www.zhihu.com/question/21833195#*

In JPEG and our project, we use 4:2:0 sampling ratio to compress *Cb* and *Cr* component. 4:2:0 means we divided the image by 4 pixels horizontally, then take 2 rows, where the first row has 2 chrominance samples and there is no color difference between first and second row. In other words, each 2 x 2 blocks, all 4 pixels hold same color value. In this project, we take the largest color value among these 4 pixels.

By applying 4:2:0 compression on *Cb* and *Cr* component, this will reduce the chrominance components into half of its original size, while we keep Y component as it is.

* + 1. Discrete Cosine Transform (DCT)

DCT expresses a finite sequence of data points in terms of a sum of cosine functions oscillating at different frequencies. It is widely used transform coding technique, is able to perform decorrelation of the input signal in a data-independent manner.

This phase, together with Quantization exploit the fact that human eyes are not good at perceiving high frequency elements in images. Think about a photo of a tree in forest. Our eyes can easily recognize the edge of the tree or the outline of the woods, but when focusing on high frequency elements such as a single blade of grass or individual leaves, our eyes cannot really pick out the details.

Using the following two-dimension DCT formula, it can help us to get rid of those high frequency elements.

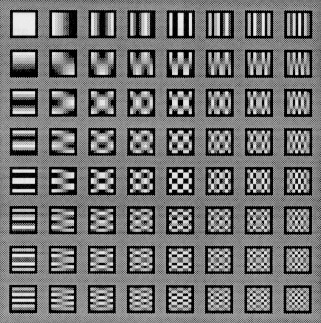
*i, j, u, v =* 0,1,…,7.

For decode, we will use inverse two-dimension DCT formula:

*i, j, u, v =* 0,1,…,7.

Note: we split each image component into 8 x 8 blocks and recenter the values by subtracting all of them by 128 (since cosine range [-1,1]).

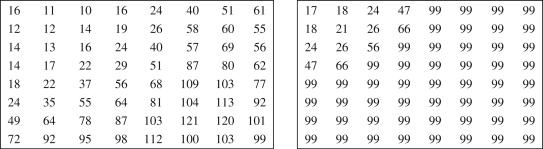
Or we can visualize DCT using these 64 base cosine waves table:

We can represent our 8 x 8 image blocks with these cosine waves by multiply each wave with a coefficient *F(i,j)*. This coefficient will measure the contribution of each wave in our blocks. For example, image with just a plain color will have more *F(0,0)* waves contribution on it.

Source: *https://users.cs.cf.ac.uk/dave/Multimedia/node231.html*

* + 1. Quantization

After DCT, we do quantization to remove high frequency data. Quantization is done by simply divides our 8 x 8 block values with its corresponding value in quantization table.



Left: JPEG luminance quantization table; right: JPEG chrominance quantization table

Source: *https://www.sciencedirect.com/topics/computer-science/quantization-matrix*

Notice that at bottom right of our quantization table we could see the values is much bigger than those in top left. This because the high frequency represented in bottom right section of our block. By divide those values in bottom right with bigger, we ensure to get resulting matrix to have those values close to zero, and round it by floor function to zero. These zeros further will be compressed by lossless compression. The chrominance quantization table also gives more penalties for high frequency data, because if high frequency data is not important in luminance component, it should even less important for chrominance.

* + 1. Different Pulse Code Modulation (DPCM) on DC Coefficients

DC coefficient appear on top left of block, which is *F(0,0)*, so each block only has one. In *F(0,0)* all cosine values are 1, so it actually represent (or proportional) to the average of entire block values.

DC coefficient usually the biggest among other AC coefficients, hence is coded separately from AC.

Suppose we have 5 DC coeffs from 5 different blocks: *DC1, DC2, DC3, DC4, DC5*. The idea of DPCM is to create a list which contains *dci = DCi+1 – DCi*, *dc0 = DC0*. We shall call *dci* as an amplitude and represented by bits (one’s complement for negative and empty bit string for 0). Each DPCM-coded DC coefficient is represented by a pair of symbols (SIZE, AMPLITUDE), where SIZE indicates how many bits are needed for representing the coefficient and AMPLITUDE contains the actual bits.

In JPEG implementation, SIZE is Huffman coded, that is, we assign shorter bit for SIZE if it appeared more frequently. In general, smaller SIZEs occur much more often means the entropy of SIZE is low. On the other hand, AMPLITUDE is not Huffman coded. Since its value can change widely (high entropy), Huffman coding is not suitable.

**procedure** DPCM(*dc\_coef*):

*dc\_coef* is list, consists dc values of all blocks in the image

*dc\_tuple* is list, will be filled by tuples of (SIZE, AMPLITUDE)

*predictor* = 0

for *i* be *dc\_coef* index:

*amplitude* = *dc\_coef*[*i*] – *predictor*

if *amplitude* is 0:

append tuple *(0,’<empty>’)* to *dc\_tuple* #<empty> is empty

else if *amplitude* is negative:

*bits* = one’s complement bits of amplitude in string

append tuple *(len(bits), bits)* to *dc\_tuple*

else:

*bits* = amplitude in normal binary representation

append tuple *(len(bits), bits)* to *dc\_tuple*

for *t* in *dc\_tuple*:

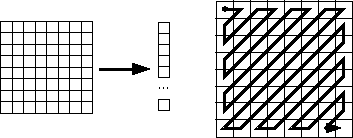
do Huffman coding on *t[0]* #SIZE

create *DPCM* *Huffman\_table*

return *DPCM* *Huffman\_table*

* + 1. Run-Length Encoding (RLE) on AC Coefficients

RLE is implemented because most part of AC coefficients (other coefficients except DC is AC) has value zero after quantization. Before RLE, first we must do a preparation which we called *zigzag scan*. Zigzag scan turns 8 x 8 block into a *64-vector*.



Zigzag scan. Source: *<https://users.cs.cf.ac.uk/dave/Multimedia/node238.html>*

As we mentioned above, DCT and quantization together will zeroed out high-spatial-frequency components, and those components appear on right bottom part of the block. With zigzag scan, our scan order has a good chance of concatenating long runs of zeros.

RLE step replaces values by a pair (RUNLENGTH, VALUE) for each run of zeros in the AC coefficients of certain image block. Suppose we scan our AC list from left to right, RUNLENGTH is the number of zeros strike until we found non-zero number which is VALUE. To further save bits, a special pair (0,0) indicates the end-of-block after last nonzero AC coefficient is reached.

After RLE, we compress its size using Huffman coding. In actual JPEG implementation, VALUE is further represented by SIZE and AMPLITUDE, similar as DCs. To save bits, RUNLENGTH and SIZE are allocated only 4 bits and squeeze it into a single byte, called *Symbol 1.* *Symbol 2* is AMPLITUDE. Since RUNLENGTH can represent only zero-runs of length 0 to 15, in case when the zero-run length exceeds 15 we need special extension code (15,0) for *Symbol 1*. Meanwhile, we keep AMPLITUDE as it is.

**procedure** RLE (*imageBlock*):

*imageBlock*, ndarray which represent 8 x 8 block

do zigzag scan for *imageBlock*, resulting *ac\_coef*

*ac\_coef*, list with 63 AC coefficients inside

*ac\_tuple*, list with RLE tuples: (*symbol1, symbol2*)

*runlength* = 0

for *i* be *ac\_coef* index:

if *ac\_coef[i]* is zero:

if *runlength* == 15:

append ( (15,0), ‘’) to *ac\_tuple*

*runlength* = 0

else:

*runlength* += 1

else:

*amplitude* = *ac\_coef[i]*

if *amplitude* is negative:

*bits* = amplitude one’s complement binary string

append ((*runlength*, len(*bits*)), *bits*) to *ac\_tuple*

else:

*bits* = amplitude original binary string

append ((*runlength*, len(*bits*)), *bits*) to *ac\_tuple*

append special tuple ((0,0),’’)

for *t* in *ac\_tuple*:

create *RLE Huffman table* based on *t[0]* #Symbol 1

return *RLE Huffman table*

* + 1. Combine All

Here we’ve summarized the step of JPEG image compression, which is also implemented in our project.

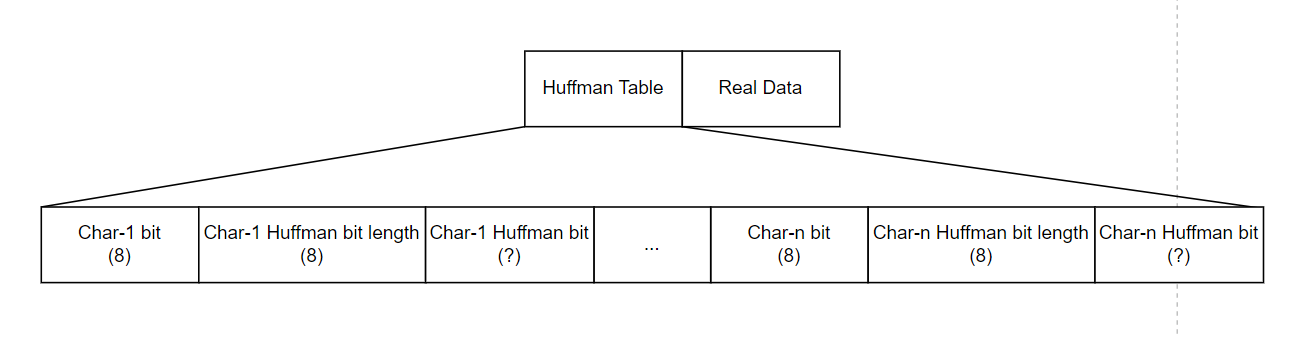
Compression steps:

* Read image file (we use BMP in this project)
* Color space conversion from RGB to YCbCr
* Image padding, so the height and width divisible by 16.
* Chroma Down-sampling 4:2:0 (data loss)
* Slice image into 8 x 8 blocks
* Discrete Cosine Transform for each block
* Quantization (data loss)
* Convert 8 x 8 block into 1D list with zigzag scan
* DPCM for DC and RLE for AC, then do Huffman coding
* Save data

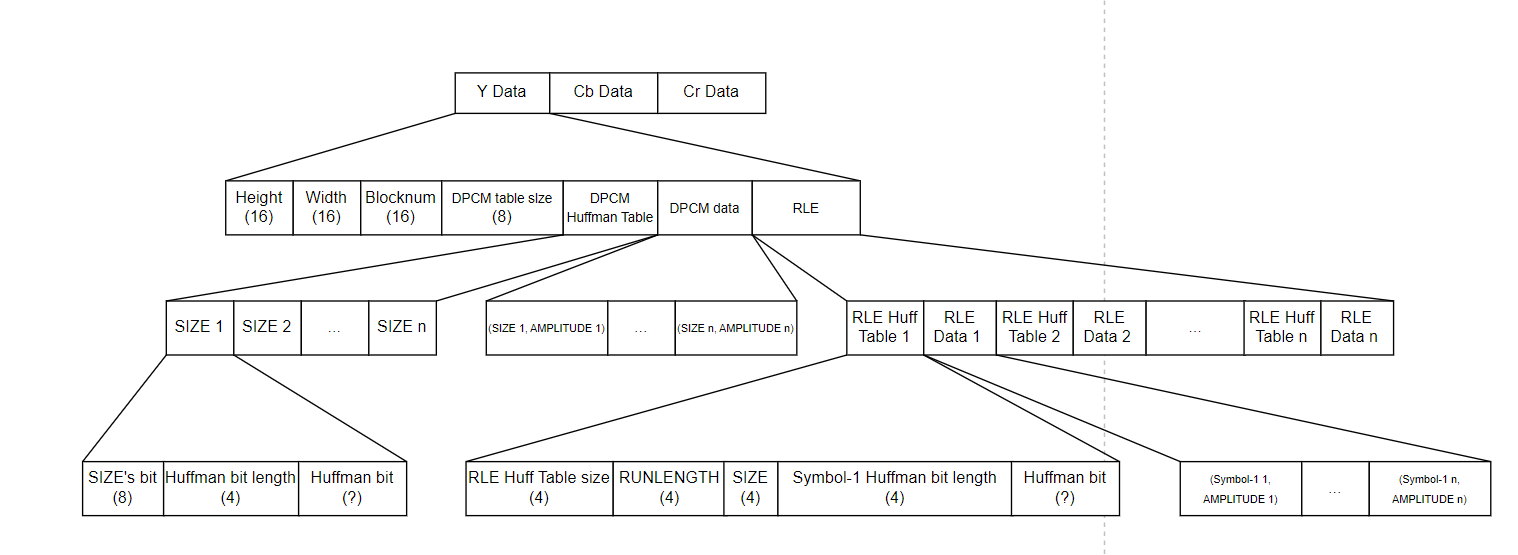
Decompression steps:

* Read data
* Huffman decoding to get 1D list
* Inverse zigzag scan to get 8 x 8 quantized block
* Inverse quantization and inverse DCT
* Rescale Cb and Cr component, combine with Y
* Convert YCbCr to RGB
  1. Text Compression Bit Allocation

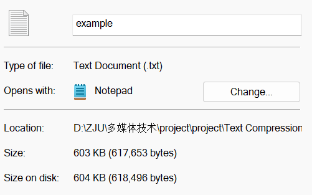
Here we show you bit map for our text compression designed by us.



* 1. Image Compression Bit Allocation

Here we show you bit map for our image compression designed by us.

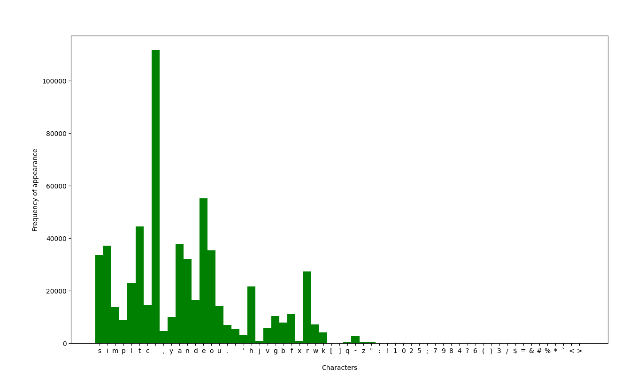
## Experiment Results

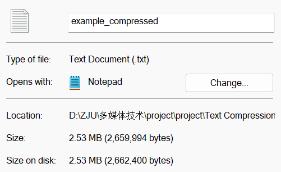
* 1. Text Compression Result

**Case 1**

Original filename: *example.txt*

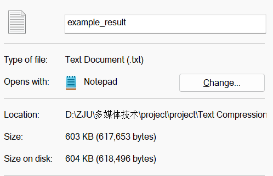
Original file size: **604 KB** (618496 bytes)

 Character distribution:

 Compressed filename: *example\_compressed.txt*

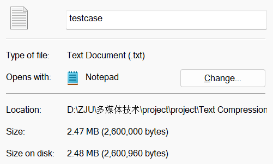
Compressed file size: 2.53 MB / 8\* = 316.25 Kb= **41.60 KB**

Compression ratio: 604/41.60 = **14.52**

 Decompressed filename: *example\_result.txt*

Decompressed file size: 604 KB (618496 bytes), same as the original.

Conclusion: compression ratio > 1; decompressed file same as the original. Success!

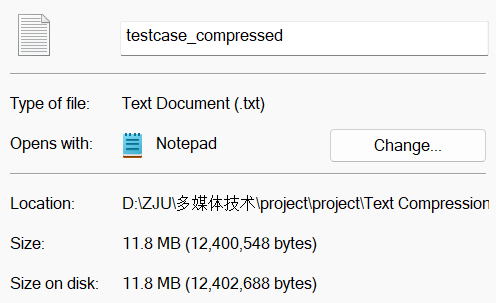
 **Case 2**

Original filename: *testcase.txt*

Original file size: 2.48MB (2600960 bytes)

Character distribution:

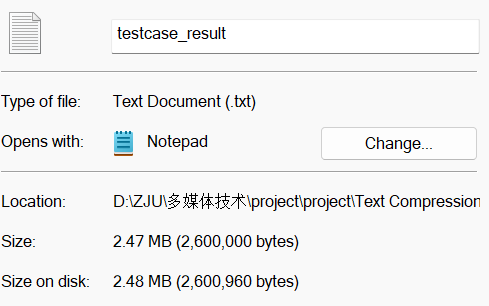
## 



Compressed filename: *testcase\_compressed.txt*

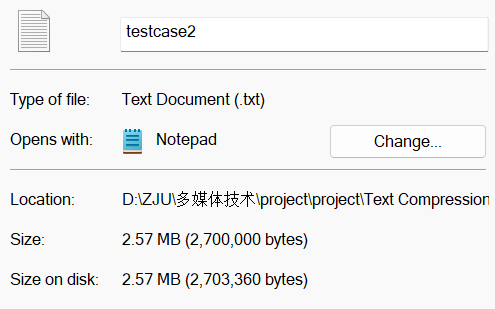
Compressed file size: 11.8 MB / 8\* = 1550.34 Kb= **0.194 MB**

Compression ratio: 2.48/0.19 = **12.72**

** Decompressed filename: *testcase\_result.txt*

Decompressed file size: 604 KB (618496 bytes)

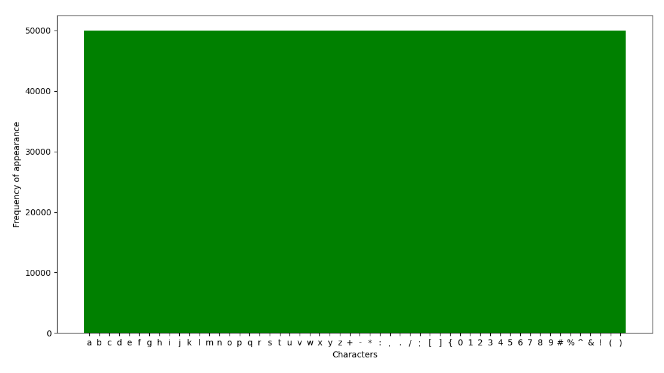
Conclusion: compression ratio > 1, smaller than case 1 because our character distribution now has bigger entropy; decompressed file same as the original. Success!

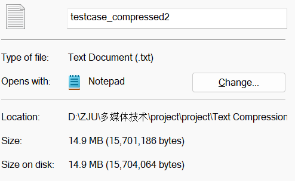
 **Case 3**

Original filename: *testcase2.txt*

Original file size: 2.57 MB (2703360 bytes)

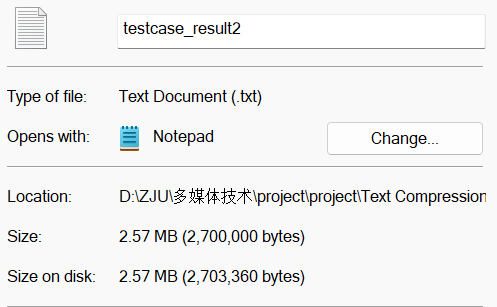
Character distribution:



Compressed filename: *testcase\_compressed2.txt*

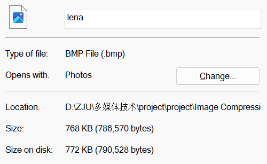
Compressed file size: 14.9 MB / 8\* = 1963.008 Kb= **0.25 MB**

Compression ratio: 2.57/0.25 = **10.49**

Decompressed filename: *testcase\_result2.txt*

Decompressed file size: 2.57 MB (2703360 bytes)

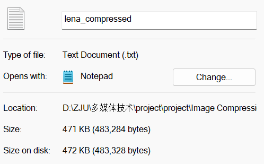
Conclusion: compression ratio > 1, smallest among all because our character distribution now has bigger entropy and more unique chars exist; decompressed file same as the original. Success!

* 1. Image Compression Result

**Case 1**

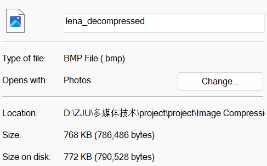
Original filename: *lena.bmp*

Original file size: 772 KB (790528 bytes)

Compressed filename: *lena\_compressed.txt*

Compressed file size: 472 KB / 8\* = 60.42 Kb= **7.55 KB**

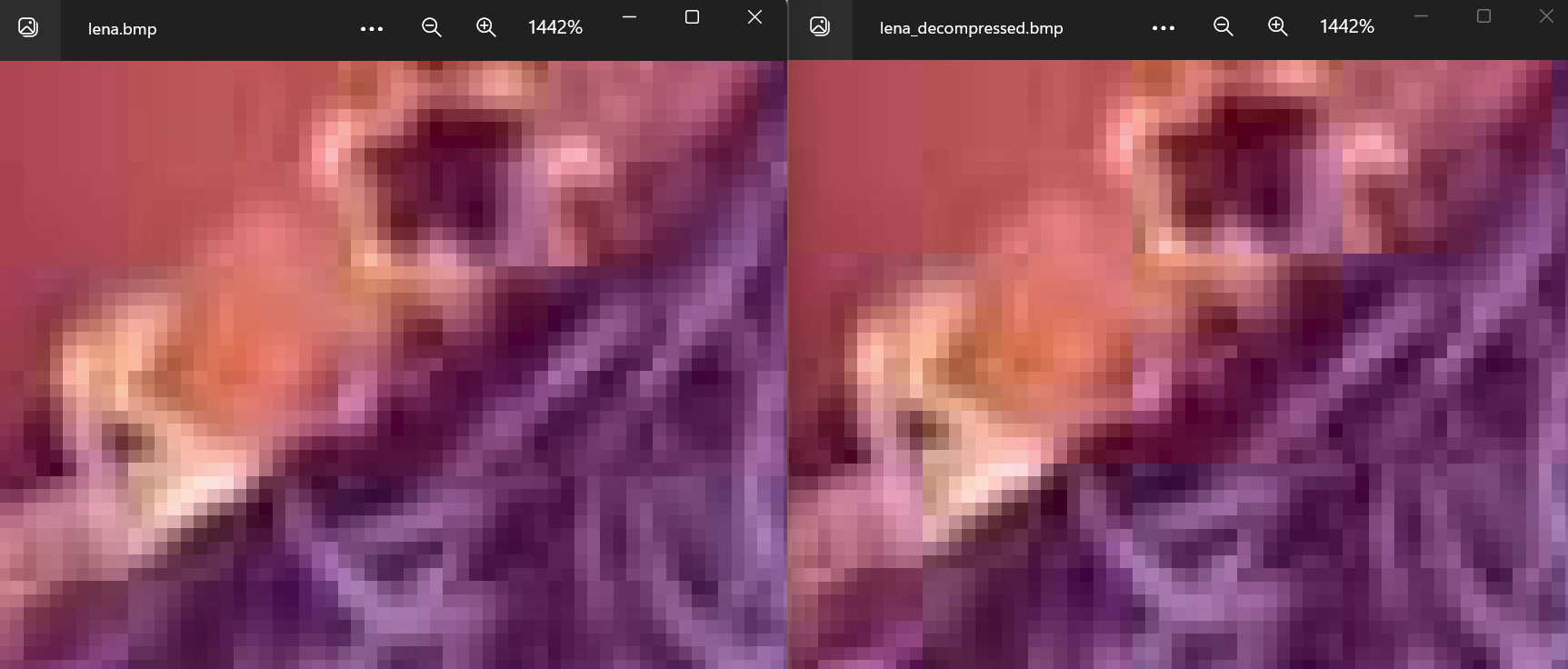
Compression ratio: 772/7.55 = **102.25**



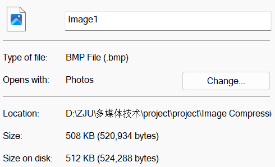
Decompressed filename: *lena\_decompressed.bmp*

Decompressed file size: 772 KB (790528 bytes)

Before vs After:

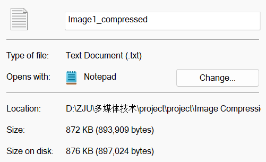
More detail:

Conclusion: The original image (left) actually already in JPEG format before, but we convert it to BMP to see its real size (1 pixel = 3 bytes) and also to compare with our algorithm. We can notice that the image is ‘blockish’ because JPEG split the image into 8 x 8 blocks. Both picture also seems very similar, therefore we conclude that our compression algorithm works well. That’s also why the compression ratio gives a huge number, because we literally have compressed JPEG image.

**Case 2**

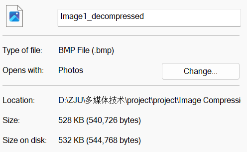
Original filename: *Image1.bmp*

Original file size: 512 KB (524288 bytes)

Compressed filename: *Image1\_compressed.txt*

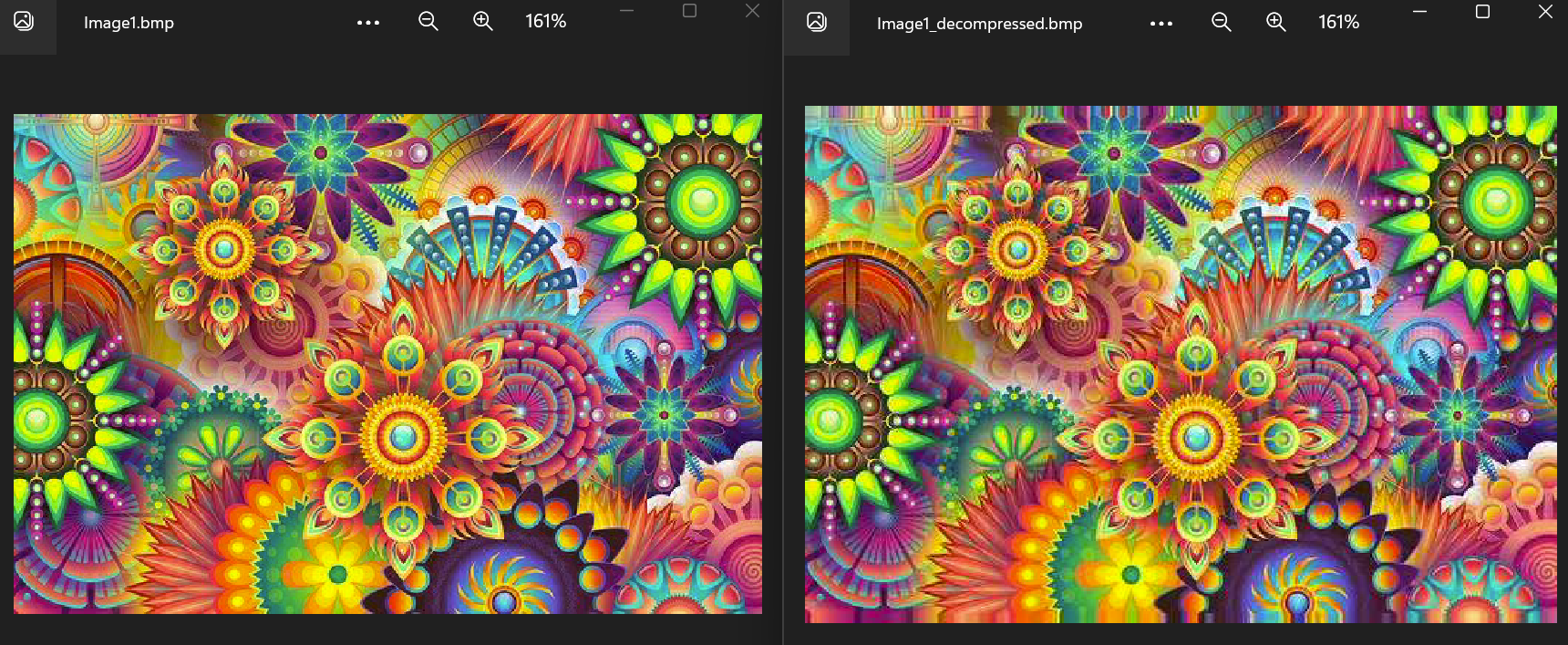
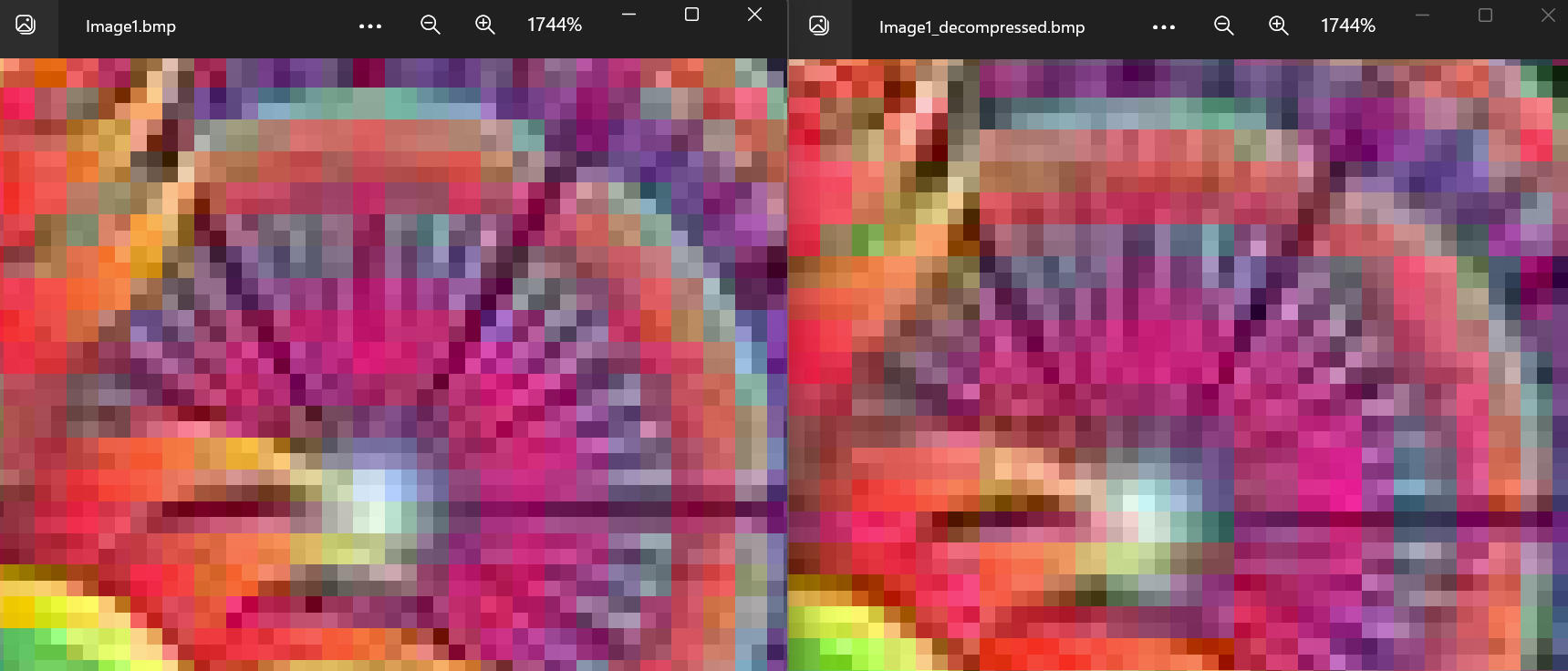
Compressed file size: 876 KB / 8\* = 112.13 Kb= **14.02 KB**

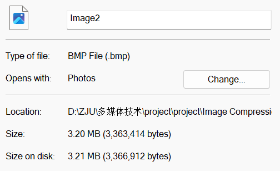
Compression ratio: 512/14.02 = **36.52**

Decompressed filename: *Image1\_decompressed.bmp*

Decompressed file size: 532 KB (544768 bytes)

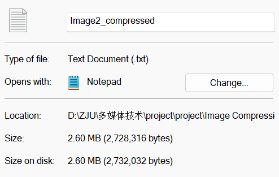
Before vs After:

Conclusion: There’s slightly different from original and decompressed image, especially when we focused on the details of flower petals (red square). This is the example of high frequency data, it appears really tiny until our eyes do not able to recognize it well. Therefore, the compression algorithm remove it. The image in decompressed image looks a bit smoother because quantization makes the color change as minimum as possible. The decompressed image has slightly bigger size because the effect of image padding. We conclude that our algorithm also works well in this case.

**Case 3**

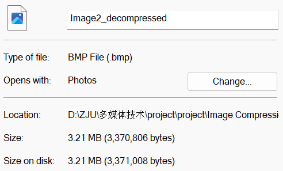
Original filename: *Image2.bmp*

Original file size: 3.21 MB (3366912 bytes)

**Compressed filename: *Image2\_compressed.txt*

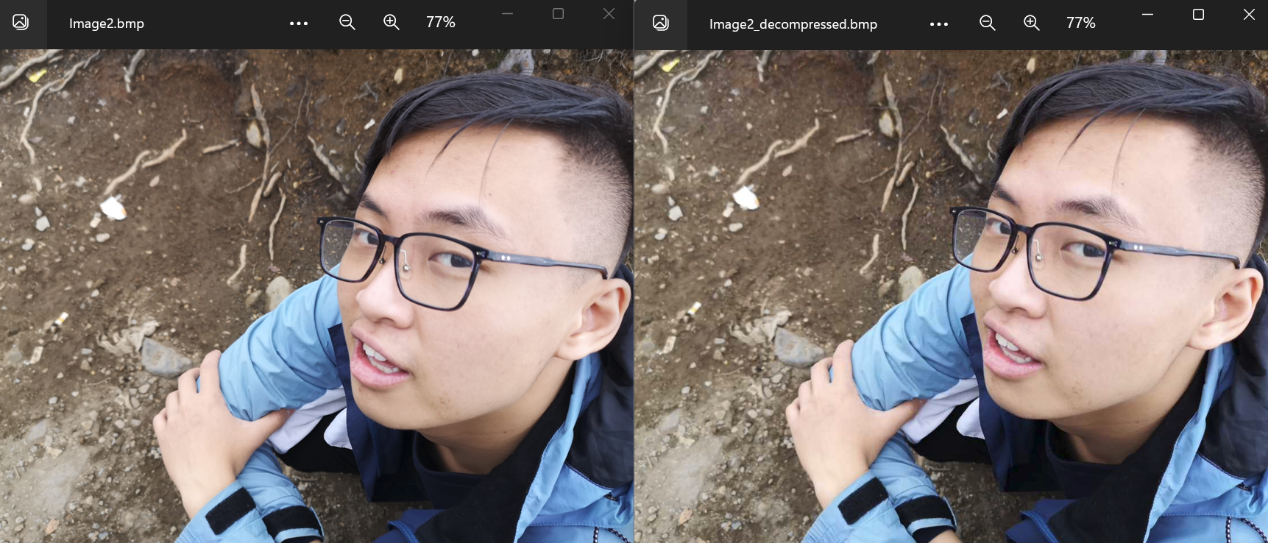
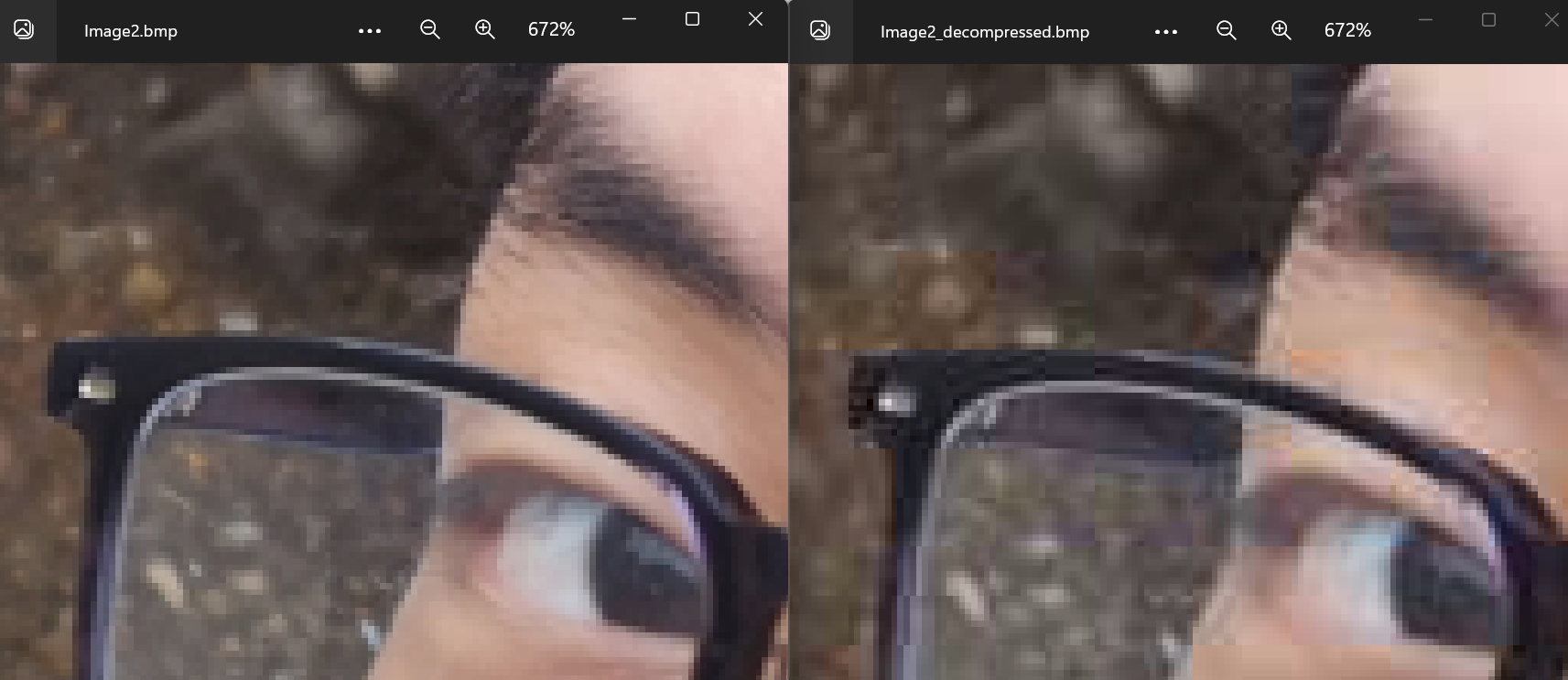
Compressed file size: 2.60 MB / 8\* = 341.50 Kb= **0.043 MB**

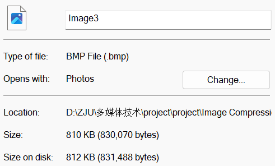
Compression ratio: 3.21/0.043 = **74.65**

Decompressed filename: *Image2\_decompressed.bmp*

Decompressed file size: 3.21 MB (3371008 bytes)

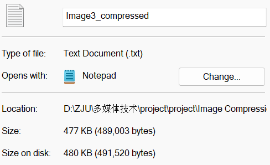
Before vs After:

Conclusion: Our compression algorithm gives bigger compression ration in this picture, since this picture has less entropy or lack of color variants. This increase the effectiveness of Huffman coding. The decompressed image also seems blockish, with a slightly different in high frequency elements (red square) and in image size (effect of padding). Our compression algorithm success to compress this image as well.

**Case 4**

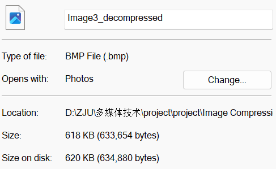
Original filename: *Image3.bmp*

Original file size: 812 KB (831488 bytes)

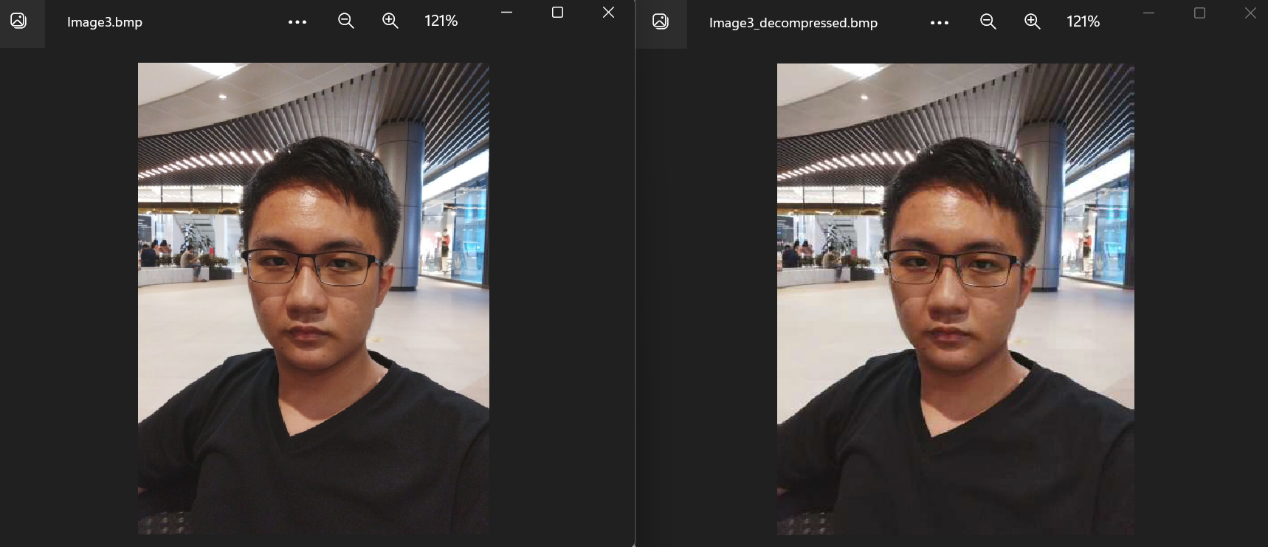
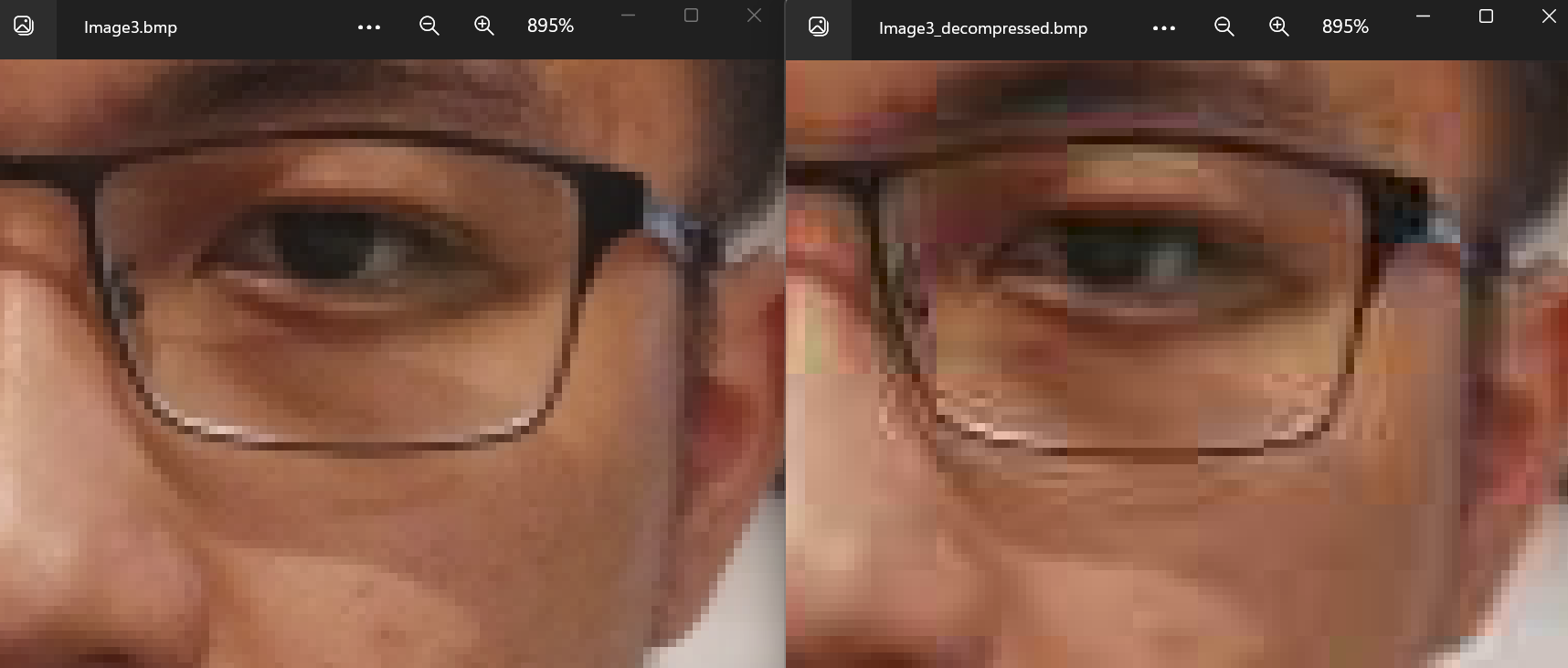
Compressed filename: *Image3\_compressed.txt*

Compressed file size: 480 KB / 8\* = 61.40 Kb= **7.68 KB**

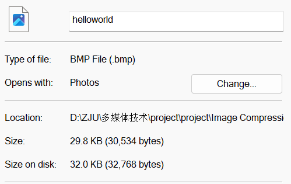
Compression ratio: 812/7.68 = **105.73**

Decompressed filename: *Image3\_decompressed.bmp*

Decompressed file size: 620 KB (634880 bytes)

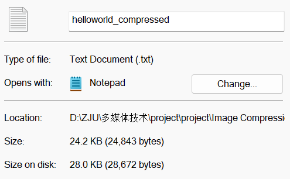


Conclusion: Similar with case 2 and case 3. The problem is we can’t figure out the answer about why our algorithm looks pretty good in this image because the compression ratio gives a crazy huge number! The resulting decompressed image’s size also decreased, while based on other cases it should be increasing because of image padding. Although it decreases the decompressed image’s size, the quality also drops even more (marked by red square).

**Case 5**

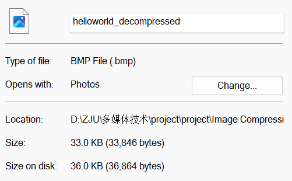
Original filename: *helloworld.bmp*

Original file size: 32.0 KB (32768 bytes)

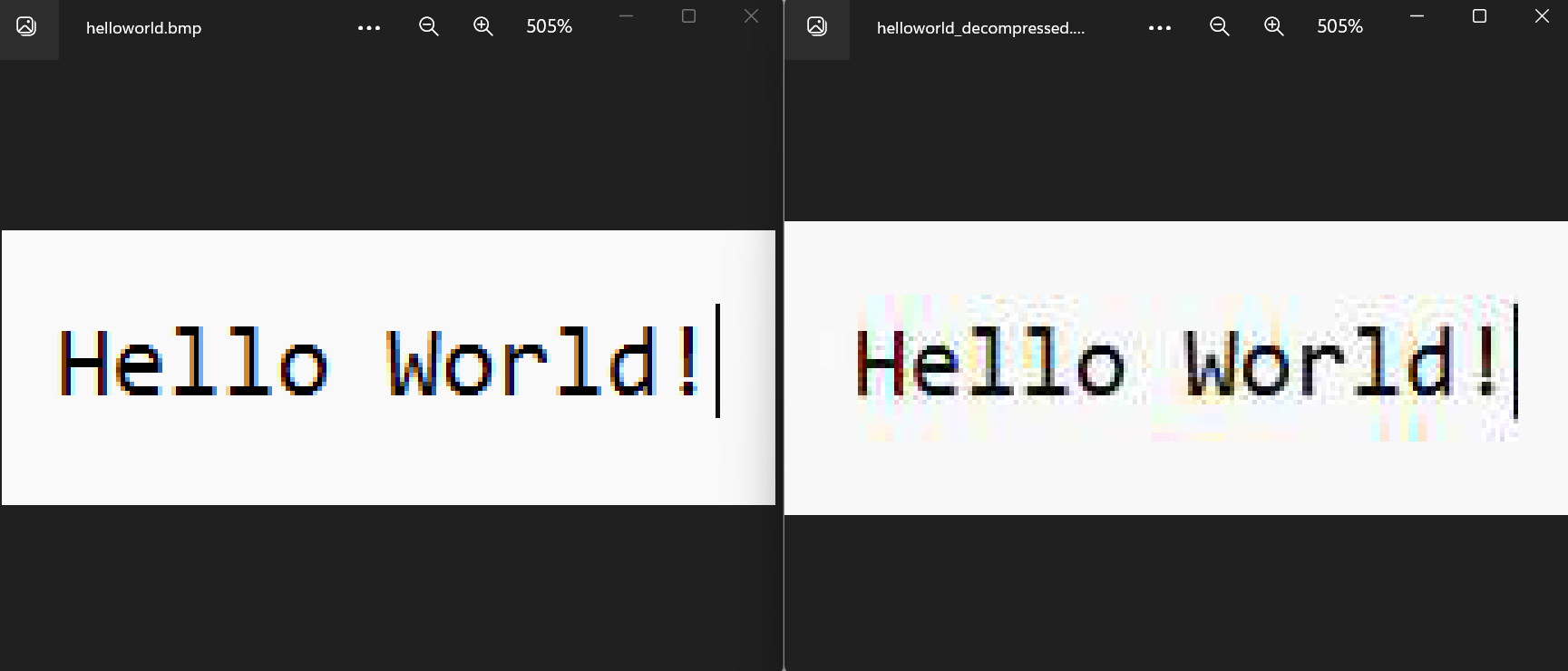
Compressed filename: *helloworld\_compressed.txt*

Compressed file size: 28.0 KB / 8\* = 3.58 Kb= **0.45 KB**

Compression ratio: 32.0/0.45 = **71.11**

Decompressed filename: *helloworld\_decompressed.bmp*

Decompressed file size: 36.0 KB (36864 bytes)

Before vs After:

Conclusion: in this case our compression algorithm doesn’t do its job very well, even in JPEG itself. Although the compression ratio also big, the image quality when we decompress the data is really bad. As you can see, our compression creates new colors around the edge of the text, which do not appear in our original image. This obviously leads to quality drops, and one of JPEG’s drawback. Those colors are result of JPEG compression side effect, because text violates our assumption that high frequency information doesn’t contribute a lot to the image. In text, there are lots of sharp changes especially in luminosity component. This condition which is not something that JPEG handles very well at all.

*\*When we measure compressed data, we divide the size by 8. This because .txt file use 8 bits (1 byte) to represent a character, while our compressed data just contains either 0 or 1, so it should be represented with one bit. Therefore, by dividing the file size by 8, we get the bit length in Kb or kilobits. Then, we divide the result again by 8, in order to convert bit to byte.*

* 1. Summary

Huffman coding is a lossless compression algorithm by reducing the bit length of those characters which appear more frequently. Huffman coding is implemented in text compression and JPEG image compression (DPCM and RLE). Huffman coding has less efffective impact while dealing high-entropy data.

JPEG is an image compression algorithm which implements lossy compression technique, chrominance downsampling and quantization. JPEG provides high compression ratio by exploiting human eyes weakness while recognizing color and high-frequency elements. However, one JPEG drawback is text image, because text image violates our assumption if high-frequency elements are not important.

We have accomplished our project very well. Not only learn about how text and image compression work, this project also makes me know some python libraries such as numpy and cv2. Although, this project is far from perfect. Many places should be improved, such as bit allocation design. By learning to design the bit allocation by ourselves, we can also optimize the compression ratio.

## References:

* *Ze, Nian Li. Drew, Mark S. Fundamentals of Multimedia. 多媒体技术教程英文版.机械工业出版社.2004*
* *<https://www.javatpoint.com/huffman-coding-java>*
* *<https://www.adobe.com/creativecloud/file-types/image/raster/jpeg-file.html>*
* *<https://en.wikipedia.org/wiki/Discrete_cosine_transform>*