

Supplement to Chapter 3 of *The Science of Digital Media* – Digital Image Processing

Programming Assignment – Digital Imaging > LZW Compression¹

Introduction:

LZW is a compression algorithm that can be applied to both text and image files. It achieves a good compression rate by recognizing commonly-recurring sequences of pixel values in image files or characters in text files. LZW compression is lossless; that is, no information is lost in the compression, so that the compressed file is identical to the original file. It is also a fixed-length method, which means that all color codes in the compressed file have the same number of bits.

The name "LZW" comes from the algorithm's creators – Lempel and Ziv – who wrote the original version of the algorithm in 1977 – and Welsh – who refined the algorithm in 1984. This method became part of the GIF file format in 1987.

LZW compression is different from other common compression algorithms (e.g. Huffman encoding) in that it does not require an initial processing of the data file to determine the codes to be used. Instead, the code table is generated "on the fly" at the same time that the file is compressed. The algorithm operates so that at each basic step, either the code for a sequence of values already exists in the table, or the code for a new sequence is generated and inserted into the table. Another interesting characteristic of LZW compression is that it is not necessary to send a full code table along with the compressed file. Only a table of the individual colors that exist in the image is needed. Then codes for sequences of colors are generated during the decompression process such that the codes are always available in the table when they are needed.

The first step in LZW is initialization of the code table to include all the colors in the image file. For an 8-bit image, this would be up to 256 colors. If all colors were used, the codes would be the values from 0 through 255, each code requiring eight bits. The algorithm proceeds by processing the pixels in the image file from left to right and top to bottom of the image. The basic idea of the algorithm is that "strings" of colors are put into the code table as they are encountered. Grouping strings of colors together into a single code results in compression.

The basic algorithm for LZW compression is given below. In the pseudocode

¹ This material is based on work supported by the National Science Foundation under Grants No. DUE-0127280 and DUE-0340969. The first version of the programming assignment was written by Dr. Jennifer Burg (burg@wfu.edu). The programming assignment was edited by Annie Lausier.

that follows, *pixelString* is a sequence of pixel values. *pixel = next pixel value* means "read the next pixel out of the image file." *pixelString + pixel* means "take the current *pixelString* value and concatenate *pixel* onto the end of it."

```

algorithm LZW
/*Input:  A bitmap image
 Output:  A table of the individual colors in the image and a
 compressed version of the file
 Note that + is concatenation*/
{
  initialize table to contain the individual colors in bitmap
  pixelString = first pixel value
  while there are still pixels to process {
    pixel = next pixel value
    stringSoFar = pixelString + pixel
    if stringSoFar is in the table then
      pixelString = stringSoFar
    else {
      output the code for pixelString
      add stringSoFar to the table
      pixelString = pixel
    }
  }
  output the code for pixelString
}

```

Basic LZW Compression Algorithm

Let's try this on a simplified example. Assume that only three colors appear in an indexed color image file – red, green, and blue, denoted by r, g, and b. Each color is represented in eight bits.

The code table is initialized as shown below.

Code	0	1	2
Color	r	g	b

Let's say that the image file has only one row of pixels in the sequence of colors given below. (Having only one row of pixels in an image isn't likely, of course, but it serves the purpose for the example.)

rrrggrrggbbbbbbbbbbbbbb

As we trace the algorithm, we'll highlight the pixels that have already been processed, including the ones currently under consideration.

1. Looking at **r**rrgggrrggbbbbbbbbbbbbbb
stringSoFar is r
r is in the table, so gather another pixel.

Table is

Code	0	1	2
Color	r	g	b

Output so far is empty.

2. Looking at **rr**gggrrggbbbbbbbbbbbbbb
stringSoFar is rr
rr is not in the table.

Output the code for r.

Put rr in the table.

Table is

Code	0	1	2	3
Color	r	g	b	rr

Output so far is 0.

3. Looking at **rrr**gggrrggbbbbbbbbbbbbbb
stringSoFar is rr
rr is in the table, so gather another pixel.

Table is

Code	0	1	2	3
Color	r	g	b	rr

Output so far is 0.

4. Looking at **rrrg**gggrrggbbbbbbbbbbbbbb
stringSoFar is rrg
rrg is not in the table.

Output the code for rr.

Put rrg in the table.

Table is

Code	0	1	2	3	4
Color	r	g	b	rr	rrg

Output so far is 0 3.

5. Looking at **rrr**gggrrggbbbbbbbbbbbbbb
stringSoFar is gg
gg is not in the table.

Output the code for g.

Put gg in the table.

Table is

Code	0	1	2	3	4	5
-------------	---	---	---	---	---	---

Color	r	g	b	rr	rrg	gg
--------------	---	---	---	----	-----	----

Output so far is 0 3 1.

6. Looking at **rrrggg**rrggbbbbbbbbbbbbbb
stringSoFar is gg
 gg is in the table, so gather another pixel.

Table is

Code	0	1	2	3	4	5
Color	r	g	b	rr	rrg	gg

Output so far is 0 3 1.

7. Looking at **rrrgggr**rrggbbbbbbbbbbbbbb
stringSoFar is ggr
 ggr is not in the table.

Output the code for gg.

Put ggr in the table

Table is

Code	0	1	2	3	4	5	6
Color	r	g	b	rr	rrg	gg	ggr

Output so far is 0 3 1 5.

8. Looking at **rrrgggr**rrggbbbbbbbbbbbbbb
stringSoFar is rr
 rr is in the table, so gather another pixel.

Table is

Code	0	1	2	3	4	5	6
Color	r	g	b	rr	rrg	gg	ggr

Output so far is 0 3 1 5.

9. Looking at **rrrgggr**rrggbbbbbbbbbbbbbb
stringSoFar is rrg
 rrg is in the table, so gather another pixel.

Table is

Code	0	1	2	3	4	5	6
Color	r	g	b	rr	rrg	gg	ggr

Output so far is 0 3 1 5.

10.

Question 1: What would happen now?

Answer:

11. Looking at **rrrgggr**rrggbbbbbbbbbbbbbb

stringSoFar is gb
 gb is not in the table.
 Output the code for g.
 Put gb in the table.
 Table is

Code	0	1	2	3	4	5	6	7	8
Color	r	g	b	rr	rrg	gg	ggr	rrgg	gb

Output so far is 0 3 1 5 4 1.

12. Looking at **rrrggrrgg**bbbbbbbbbbbbbb
stringSoFar is bb
 bb is not in the table.
 Output the code for b.
 Put bb in the table.
 Table is

Code	0	1	2	3	4	5	6	7	8	9
Color	r	g	b	rr	rrg	gg	ggr	rrgg	gb	bb

Output so far is 0 3 1 5 4 1 2.

13.
Question 2. What would happen now?
Answer:

14. Looking at **rrrggrrgg**bbbbbbbbbbbbb
stringSoFar is bbb
 bbb is not in the table.
 Output the code for bb.
 Put bbb in the table.
 Table is

Code	0	1	2	3	4	5	6	7	8	9	10
Color	r	g	b	rr	rrg	gg	ggr	rrgg	gb	bb	bbb

Output so far is 0 3 1 5 4 1 2 9.

15. Looking at **rrrggrrgg**bbbbbbbbbbbbb
stringSoFar is bb
 bb is in the table, so gather another pixel.
 Table is

Code	0	1	2	3	4	5	6	7	8	9	10
Color	r	g	b	rr	rrg	gg	ggr	rrgg	gb	bb	bbb

Output so far is 0 3 1 5 4 1 2 9.

16. Looking at **rrrggrrgg**bbbbbbbbbbbbb
stringSoFar is bbb
 bbb is in the table, so gather another pixel.

Table is

Code	0	1	2	3	4	5	6	7	8	9	10
Color	r	g	b	rr	rrg	gg	ggr	rrgg	gb	bb	bbb

Output so far is 0 3 1 5 4 1 2 9.

17. Looking at **rrrggrrggbbb****bbbb**bbbbbbbb

stringSoFar is bbbb

bbbb is not in the table.

Output the code for bbb.

Put bbbb in the table.

Table is

Code	0	1	2	3	4	5	6	7	8	9	10	11
Color	r	g	b	rr	rrg	gg	ggr	rrgg	gb	bb	bbb	bbbb

Output so far is 0 3 1 5 4 1 2 9 10.

18. Looking at **rrrggrrggbbbbbb****bb**bbbbbbbb

stringSoFar is bb

bb is in the table, so gather another pixel.

Table is

Code	0	1	2	3	4	5	6	7	8	9	10	11
Color	r	g	b	rr	rrg	gg	ggr	rrgg	gb	bb	bbb	bbbb

Output so far is 0 3 1 5 4 1 2 9 10.

19. Looking at **rrrggrrggbbbbbb****bbb**bbbbbbbb

stringSoFar is bbb

bbb is in the table, so gather another pixel.

Table is

Code	0	1	2	3	4	5	6	7	8	9	10	11
Color	r	g	b	rr	rrg	gg	ggr	rrgg	gb	bb	bbb	bbbb

Output so far is 0 3 1 5 4 1 2 9 10.

20. Looking at **rrrggrrggbbbbbb****bbbb**bbbbbbbb

stringSoFar is bbbb

bbbb is in the table, so gather another pixel.

Table is

Code	0	1	2	3	4	5	6	7	8	9	10	11
Color	r	g	b	rr	rrg	gg	ggr	rrgg	gb	bb	bbb	bbbb

Output so far is 0 3 1 5 4 1 2 9 10.

21. Looking at **rrrggrrggbbbbbb****bbbbbb**bbbbbbbb

stringSoFar is bbbbbb

bbbbbb is not in the table.

Output the code for bbbb.

Put bbbbb in the table.

Table is

Code	0	1	2	3	4	5	6	7	8	9	10	11	12
Color	r	g	b	rr	rrg	gg	ggr	rrgg	gb	bb	bbb	bbbb	bbbbb

Output so far is 0 3 1 5 4 1 2 9 10 11.

22. Looking at **rrrggrrggbbbbbbbbbbbbb**

stringSoFar is bb

bb is in the table, so gather another pixel.

Table is

Code	0	1	2	3	4	5	6	7	8	9	10	11	12
Color	r	g	b	rr	rrg	gg	ggr	rrgg	gb	bb	bbb	bbbb	bbbbb

Output so far is 0 3 1 5 4 1 2 9 10 11.

23. Looking at **rrrggrrggbbbbbbbbbbbbb**

stringSoFar is bbb

bbb is in the table, so gather another pixel.

Table is

Code	0	1	2	3	4	5	6	7	8	9	10	11	12
Color	r	g	b	rr	rrg	gg	ggr	rrgg	gb	bb	bbb	bbbb	bbbbb

Output so far is 0 3 1 5 4 1 2 9 10 11.

24. Looking at **rrrggrrggbbbbbbbbbbbbb**

stringSoFar is bbbb

bbbb is in the table, so gather another pixel.

Table is

Code	0	1	2	3	4	5	6	7	8	9	10	11	12
Color	r	g	b	rr	rrg	gg	ggr	rrgg	gb	bb	bbb	bbbb	bbbbb

Output so far is 0 3 1 5 4 1 2 9 10 11.

25. Looking at **rrrggrrggbbbbbbbbbbbbb**

stringSoFar is bbbbb

bbbbb is in the table.

There are no more pixels.

Output the code for bbbbb.

Table is

Code	0	1	2	3	4	5	6	7	8	9	10	11	12
Color	r	g	b	rr	rrg	gg	ggr	rrgg	gb	bb	bbb	bbbb	bbbbb

Final output is 0 3 1 5 4 1 2 9 10 11 12.

Let's figure out the compression rate for this example.

We noted already that it takes two bits to encode each of the colors in the uncompressed file.

Question 3: If each pixel requires eight bits, what would be the size of an uncompressed file with 25 pixels?

Answer:

LZW uses a fixed-length code encoding scheme. This means that each code must be the same number of bits. The largest code in our example is 12.

Question 4: If the largest code is 12, how many bits would be required for each code?

Answer:

Question 5: What would our encoded file look like in binary (disregarding the code table with the original colors)?

Answer:

Question 6: How many bits would be required for our encoded file?

Answer:

Question 7: What is the compression rate for this example?

Answer:

You can see that when an image has long sequences of colors that are repeated throughout the image, a good compression rate can be achieved. Another advantage of LZW compression is that the full code table does not need to be stored with the compressed image. It is regenerated automatically as the image is decompressed. Let's try decompression on our example.

Decompression begins with a code table initialized to contain the individual colors in the image file. In our case, that would be

Code	0	1	2
Color	r	g	b

The basic algorithm for LZW decompression is given below.

colors(previousCode) means "the color sequence associated with *previousCode*". *firstColor(currentCode)* means the first color in the color sequence associated with *currentCode*. + is used for concatenation, as before. *colors(NULL)* is defined as *NULL*.

```

algorithm LZW_decompress
/*Input:      Compressed bitmap image and table of individual colors in
image
Output:      Decompressed image*/
{
  /*Initialize table*/
  stringSoFar = NULL
  while there are still codes to process in the code string {
    code = next code in the code string
    colors = the colors corresponding to code in the table

    if colors == NULL /*Case where code is not in the table*/
    /*stringSoFar[0] is the first color in stringSoFar*/
    colors = stringSoFar + stringSoFar[0]

    output colors
    if stringSoFar != NULL
    put stringSoFar + colors[0] in the table
    stringSoFar = colors
  }
}

```

Basic LZW Decompression Algorithm

Let's trace the algorithm for a few steps. As we trace the algorithm, we'll highlight the codes that have already been processed, including the one currently under consideration.

1. Looking at **0** 3 1 5 4 1 2 9 10 11 12
 stringSoFar is NULL
 code is 0
 colors is r
 Output r
 Code table is

Code	0	1	2
Color	r	g	b

Output so far is r
 stringSoFar is r

2. Looking at **0 3** 1 5 4 1 2 9 10 11 12
 stringSoFar is r
 code is 3
 colors is NULL
 colors is rr
 Output rr
 Put rr in the table

Code table is

Code	0	1	2	3
Color	r	g	b	rr

stringSoFar to rr
Output so far is rrr

3. Looking at **0 3 1** 5 4 1 2 9 10 11 12

stringSoFar is rr

code is 1

colors is g

Output g.

Put rrg in the table.

Code table is

Code	0	1	2	3	4
Color	r	g	b	rr	rrg

stringSoFar is g
Output so far is rrrg.

Question 8: What would be the next step?

Answer:

You should be able to see from here how the code table is regenerated as the file is decompressed.

Instructions:

The Assignment

Using the programming language of your choice, implement a simulation of the LZW compression algorithm operating on text data.

What do we mean by a "simulation"? Consider what you'd have to do to implement the LZW compression algorithm at the bit level. It isn't until you've compressed the entire file that you know how many items are in your code table. At that point, you know the number of bits you need for the largest code value. Since you initially don't know the largest code value, you could simply use integer values for the codes in your table. After you've gone through the image file and encoded it using variables of type *integer* for the codes, you could figure out, based on the largest code value in the table, the minimum number of bits needed to store the codes. Call this value n . Then you could make another pass through the encoded file and rewrite each code in n bits.

In a language such as C or C++, it's not too hard to do bit-level operations. (For a good implementation of LZW at the bit level, see *Data Structures, Algorithms, and Applications in C++* by Sartaj Sahni). But we think that you'll get the point of the algorithm just as well by implementing a simulation

of the compression. That is, you can use *integer* variables in your code table and print out the codes as integers. You'll still be able to compute the compression rate and see how the algorithm works, although in fact you'll not be making your file any smaller.

Another reason for doing a simulation rather than the real thing is that there are some complications in the implementation of LZW compression that we don't want to deal with here. These complications make LZW compression an interesting problem in data structures and algorithms courses, but they are not the point of our discussion in the context of digital media. The complications are related to the size of the code table and the computational complexity of searching it repeatedly. Optimizations in the implementation of LZW compression (like using hash tables and a clever trick for representing the strings in the code table) make LZW efficient in commercial implementations, but we will not discuss those optimizations here.

We'll use text files rather than image files since they're easier to handle. The concepts are the same. Your program should take as input a string of characters. Prompt the user for the name of the input file. For convenience in verifying correctness, the program should ask the user if he or she wants the code table, the encoded text, and/or the compression ratio written to the screen or to a file on disk.

Put the actions above in a loop in your main function, asking the user if he or she wants to compress another text file.

Before Writing the Program

To run your program, you will need to create some text files to read. Ask your instructor if you should create these yourself or if he/she wants to give you the text files.

Tips for C/C++ Programmers



Use the *string* type for easy concatenation and other string operations.



Initialize your code table to contain only the characters in the text file. This will require a pre-pass over the text file.

Ideas for Further Experimentation and Analysis

- Implement the decompression algorithm. Compare the original file with the file you get after compressing and decompressing. Verify that LZW compression is lossless.
- Do some research on LZW compression. What is a more efficient way of representing the strings in the code table? What is a more efficient way of storing and searching the code table? What are other optimizations to the code?

- Locate places in photographic or video processing software where LZW compression is used or can be chosen as an option.

Answers to Questions:

Question 1. Looking at **rrrggrrgg**bbbbbbbbbbbbbbbb
stringSoFar is rrgg
 rrgg is not in the table.
 Output the code for rrg
 Put rrgg in the table

Table is

Code	0	1	2	3	4	5	6	7
Color	r	g	b	rr	rrg	gg	ggr	rrgg

Output so far is 0 3 1 5 4.

Question 2. Looking at **rrrggrrggbbb**bbbbbbbbbbbbbb
stringSoFar is bb
 bb is in the table, so gather another pixel.

Table is

Code	0	1	2	3	4	5	6	7	8	9
Color	r	g	b	rr	rrg	gg	ggr	rrgg	gb	bb

Output so far is 0 3 1 5 4 1 2.

Question 3: 200 bits (i.e., 25 bytes)

Question 4: 4 bits

Question 5:

0000 0011 0001 0101 0100 0001 0010 1001 1010 1011 1100

Question 6: 44 bits plus a few for initial table

Question 7: 200 to 44 = 50 to 11. The compressed file is about 1/5th its original size.

Question 8:

4. Looking at **0 3 1 5** 4 1 2 9 10 11 12
stringSoFar is g
 code is 5
 colors is NULL
 colors is gg
 Output gg.
 Put gg in the table.
 Code table is

Code	0	1	2	3	4	5
Color	r	g	b	rr	rrg	gg

stringSoFar is gg

Output so far is rrrggg.