LZW (Lempel - Ziv - Welch) compression is a lossless compression algorithm. It is a dictionary-based algorithm. You could even call it a palette-based algorithm since the dictionary works like a palette. This algorithm takes an established dictionary and builds upon that dictionary as you read the compressed information. This will primarily talk about decompression, but we will move into compression of data as well.

When reading any LZW compressed data, generally you will be given a dictionary that you must expand upon however, that is not always the case. This is also just to give a basic perspective on these types of problems. Information that is generally required is how many bits are used for the minimum code. Other information that is useful to know is how you read the bytes and bits. The bytes are read from left to right, but the bits will most likely be read from right to left. The size of the dictionary must be a power of 2.

When reading using the start number of bits for a code, we must allocate 2 values for the clear dictionary code and the end of data code. These are not at the end of the dictionary, but they are the first 2 new values added to the dictionary. When you reach the max value of what the number of bits you have can store, you add another bit. This will be shown in the example. Every time something is read, a new item is added into the dictionary regardless of whether what was added will ever occur. The rule when reading data is:

* If the number is in the dictionary, that is the next value.
  + The next dictionary entry is the next available value.
  + The data for the next dictionary entry is the last data + the first part of new data. In that order. Last + first part of new. (Remember this)
* If the number is not in the dictionary, the next value is the last + the first part of the last
  + The next dictionary entry is the next available value. In this case, it is the number we read.
  + The data for the next dictionary entry is the value of this entry since this is the next entry.
  + If the number we read is the max we can read with the available bits, add another bit.

Let’s give an example using a table of A, B, C

This is the final string A B A A B A C B C C B A B C

Spacing was added to make it easier to read.

Let’s make the encoded data. We will use 3 bits to start with since we have 3 values and we need 2 for the clear dictionary and end of data codes.

|  |  |
| --- | --- |
| Value | Code |
| A | 000 |
| B | 001 |
| C | 010 |
| Clear Dictionary | 011 |
| End of Data | 100 |

Now let’s use the string ABAABACBCCBABC to build our encoded data string

Continue to read until you find a value that is not in your dictionary.

For example, starting from the front. If AB is in the dictionary, check if ABA is. If not, AB is the value and ABA is the next dictionary value. If ABA is in the value, check if ABAA is in the dictionary. Repeat the process until you reach the end or a value that is in the dictionary. If the next code hits the max,

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Value | Code | Next Value | Next Code | Next dictionary value | Next dictionary code | Final Code |
| Clear Dictionary | 011 / 3 | N/A | N/A | N/A | N/A | 011 / 3 |
| A | 000 / 0 | B | 001 / 1 | AB | 101 / 5 | 000 / 0 |
| B | 001 / 1 | A | 000 / 0 | BA | 110 / 6 | 001 / 1 |
| A | 000 / 0 | A | 000 / 0 | AA | 111 / 7 | 000 / 0 |
| AB | 101 / 5 | A | 000 / 0 | ABA | 1000 / 8 \*\*\* | 101 / 5 |
| A | 0000 / 0 | C | 0010 / 2 | AC | 1001 / 9 | 0000 / 0 |
| C | 0010 / 2 | B | 0001 / 1 | CB | 1010 / 10 | 0010 / 2 |
| B | 0001 / 1 | C | 0010 / 2 | BC | 1011 / 11 | 0001 / 1 |
| C | 0010 / 2 | C | 0010 / 2 | CC | 1100 / 12 | 0010 / 2 |
| CB | 1010 / 10 | A | 0000 / 0 | CBA | 1101 / 13 | 1010 / 10 |
| AB | 0101 / 5 | C | 0010 / 2 | ABC | 1110 / 14 | 0101 / 5 |
| C | 0010 / 2 | N/A | N/A | N/A | N/A | 0010 / 2 |
| End of data | 0100 / 4 | N/A | N/A | N/A | N/A | 0100 / 4 |

Using the final code column, we can create the final encoded data. It should look similarly to this.

We start from end of data and go up. Padding is added to the first byte which will become the last byte.

0100 0010 0101 1010 0010 0001 0010 0000 101 000 001 000 011

0100001 00101101 00010000 10010000 01010000 01000011

We reverse the bytes and not the bits. It should look more like this with additional padding. The padding goes in front of the last byte in this list. In this case, it is 1 bit of padding.

01000011 01010000 10010000 00010000 00101101 00100001

When turned into hexadecimal it looks like this (We use the reverse data as that is typically how the data is stored in many files):

0x43 0x50 0x90 0x10 0x2D 0x21

If the data was stored with just bytes and no encoding, it would look like this

0x41 0x42 0x41 0x41 0x42 0x41 0x43 0x42 0x43 0x43 0x42 0x41 0x42 0x43

So, we saved 8 bytes of data of the 14 bytes it originally took. If we used just a palette with no padding or anything, we would have 4 bytes instead. Since you cannot have 3.5 bytes

Now let’s try to decompress the final data using the predefined table to get back where we were.

Keeping in mind that we use the last data instead of the next data.

|  |  |
| --- | --- |
| Value | Code |
| A | 000 |
| B | 001 |
| C | 010 |
| Clear Dictionary | 011 |
| End of Data | 100 |

0x43 0x50 0x90 0x10 0x2D 0x21

First, move the bytes around so we can just read from right to left and not left to right with the bits being right to left. Just makes it easier to see, however your data will not be sorted in this way by default.

0x21 0x2D 0x10 0x90 0x50 0x43

Now convert into binary. There is an underscore used to separate every 4 bits. A space is used to separate bytes.

0010\_0001 0010\_1101 0001\_0000 1001\_0000 0101\_0000 0100\_0011

Now let’s make a new dictionary using the previous dictionary kind of like we did when creating the encoded data. We read from the right to the left and start with 3 bits

For reference: A B A A B A C B C C B A B C

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Value | Code | Last Value | Last code | Next Dictionary value | Next Dictionary Code |
| Clear Dictionary | 011 | N/A | N/A | N/A | N/A |
| A | 000 | N/A | N/A | N/A | N/A |
| B | 001 | A | 000 | AB | 101 |
| A | 000 | B | 001 | BA | 110 |
| AB | 101 | A | 000 | AA | 111 \*\*\* |
| A | 0000 | AB | 0101 | ABA | 1000 |
| C | 0010 | A | 0000 | AC | 1001 |
| B | 0001 | C | 0010 | CB | 1010 |
| C | 0010 | B | 0001 | BC | 1011 |
| CB | 1010 | C | 0010 | CBC | 1100 |
| AB | 0101 | CB | 1010 | ABC | 1101 |
| C | 0010 | AB | 0101 | CA | 1110 |
| End of data | 0100 | N/A | N/A | N/A | N/A |

Final String is A B A A B A C B C C B A B C

If we compare our data with the reference data, we see that it is the same meaning that we have decompressed correctly.

As you saw, we switched from using 3 bits to 4 bits in both instances. When you see the star, you add one bit to your current amount.

Something that is note-worthy is that you are not required to have to dictionary to decompress. When you decompress without a dictionary, you don’t get the values, but their location in the dictionary. You are just required to know how many elements are in the dictionary. If we repeat the decompression with a fake dictionary and apply are normal dictionary afterwards and replace, we should have the same value.

Our dictionary has 3 value not including the 2 special symbols. The special symbols are always known.

In this case, we are referring to numbers in the dictionary and not the value. Concatenation is specified by a ‘,’.

For reference, here is the compressed data as a binary string with \_ separating every 4 bits and a space separating every byte. We start at the end of the data and start with 3 bits of data because of our dictionary size.

0010\_0001 0010\_1101 0001\_0000 1001\_0000 0101\_0000 0100\_0011

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Value | Code | Last Value | Last code | Next Dictionary value | Next Dictionary Code |
| Clear Dictionary | 011 - 3 | N/A | N/A | N/A | N/A |
| 1 | 000 - 0 | N/A | N/A | N/A | N/A |
| 2 | 001 - 1 | 1 | 000 | 1,2 | 101 |
| 1 | 000 - 0 | 2 | 001 | 2,1 | 110 |
| 1,2 | 101 - 5 | 1 | 000 | 1,1 | 111 \*\*\* |
| 1 | 0000 - 0 | 1,2 | 0101 | 1,2,1 | 1000 |
| 3 | 0010 - 2 | 1 | 0000 | 1,3 | 1001 |
| 2 | 0001 - 1 | 3 | 0010 | 3,2 | 1010 |
| 3 | 0010 - 2 | 2 | 0001 | 2,3 | 1011 |
| 3,2 | 1010 - 10 | 3 | 0010 | 3,2,3 | 1100 |
| 1,2 | 0101 - 5 | 3,2 | 1010 | 1,2,3 | 1101 |
| 3 | 0010 - 2 | 1,2 | 0101 | 3,1 | 1110 |
| End of data | 0100 - 4 | N/A | N/A | N/A | N/A |

1, 2, 1, 1, 2, 1, 3, 2, 3, 3, 2, 1, 2, 3

Then using our original dictionary above,

A B A A B A C B C C B A B C

The result we previously got:

A B A A B A C B C C B A B C

Both match like we expect.

When implementing this, it is important to be able to separate the bits by unknown values that could exceed the size of a byte. Creating a data structure that can contain all the bits and convert them in to a series of bytes is very helpful.

Also, you may choose to limit you dictionary size as well. An important aspect is whether the dictionary values can be more than one byte and whether the base dictionary is able to be more than a byte in size. Both aspects determine your compression ratio and how you should implement it.

Corrections may be made to this document to correct any errors.

As a test, lets use the data for a 3x5 gif image that consist of black and white pixels. Though the Wikipedia page uses different values than the actual output from paint (likely due to different versions of paint), they produce the same results. We use the version that our paint provided.

It uses a global table where 0 and 40 are black and 255 is white.

For paint,

There are 256 values in the dictionary so 257

00 01 EC 1B 28 70 A0 C1 83 01 01

As binary separated by bytes

0000\_0000 0000\_0001 1110\_1100 0001\_1011 0010\_1000 0111\_0000

1010\_**0000 1100\_0001 1000\_0011** **0000\_0001 0000\_0001**

//After reversing the data around to easily read.

01 01 83 C1 A0 70 28 1B EC 01 00

As binary separated by bytes

0000\_0001 0000\_0001 1000\_0011 1100\_0001 1010\_0000 0111\_0000 0010\_1000 0001\_1011 1110\_1100 0000\_0001 0000\_0000

0000\_000

1\_0000\_0001 - 257

1\_0000\_0111 - 263

1\_0000\_0110 - 262

1\_0000\_0011 - 259

1\_0000\_0010 - 258

1\_0000\_0011 - 259

0\_1111\_1011 - 251

0\_0000\_0000 - 0

1\_0000\_0000 - 256

Note: For the Wikipedia version, 251 is where the color White is stored and 0 is where Black is stored. There are 256 colors in the Wikipedia gif, but the last 4 colors are just black repeated. It does not use a global table and instead uses a local table.

Last thing worth noting, for compression and decompression, you may run into indices that aren’t in the table yet. They are always the next index in the dictionary and follow the rules listed up top.