[top](#)

Overview

If you were to take a look at almost any data file on a computer, character by character, you would notice that there are many recurring patterns. LZW is a data compression method that takes advantage of this repetition. The original version of the method was created by Lempel and Ziv in 1978 (LZ78) and was further refined by Welch in 1984, hence the LZW acronym. Like any adaptive/dynamic compression method, the idea is to (1) start with an initial model, (2) read data piece by piece, (3) and update the model and encode the data as you go along. LZW is a "dictionary"-based compression algorithm. This means that instead of tabulating character counts and building trees (as for Huffman encoding), LZW encodes data by referencing a dictionary. Thus, to encode a substring, only a single code number, corresponding to that substring's index in the dictionary, needs to be written to the output file. Although LZW is often explained in the context of compressing text files, it can be used on any type of file. However, it generally performs best on files with repeated substrings, such as text files.

[top](#)

Compression

LZW starts out with a dictionary of 256 characters (in the case of 8 bits) and uses those as the "standard" character set. It then reads data 8 bits at a time (e.g., 't', 'r', etc.) and encodes the data as the number that represents its index in the dictionary. Everytime it comes across a new substring (say, "tr"), it adds it to the dictionary; everytime it comes across a substring it has already seen, it just reads in a new character and concatenates it with the current string to get a new substring. The next time LZW revisits a substring, it will be encoded using a single number. Usually a maximum number of entries (say, 4096) is defined for the dictionary, so that the process doesn't run away with memory. Thus, the codes which are taking place of the substrings in this example are 12 bits long (2^12 = 4096). It is necessary for the codes to be longer in bits than the characters (12 vs. 8 bits), but since many frequently occuring substrings will be replaced by a single code, in the long haul, compression is achieved.

Here's what it might look like in pseudocode:

```
string s;
char ch;
...
```

s = empty string;
while (there is still data to be read)
{
   ch = read a character;
   if (dictionary contains s+ch)

```
    {
    s = s+ch;
    }
    else
    {
    encode s to output file;
    add s+ch to dictionary;
    s = ch;
    }
}
encode s to output file;
```

Now, let's suppose our input stream we wish to compress is "banana_bandana", and that we are only using the initial dictionary:

```
Index   Entry
  0       a
  1       b
  2       d
  3       n
  4       _ (space)
```

The encoding steps would proceed like this:

Input	Current String	Seen this Before?	Encoded Output	New Dictionary Entry/Index
b	b	yes	nothing	none
ba	ba	no	1	ba / 5
ban	an	no	1,0	an / 6
bana	na	no	1,0,3	na / 7
banan	an	yes	no change	none
banana	ana	no	1,0,3,6	ana / 8
banana_	a_	no	1,0,3,6,0	a_ / 9
banana_b	_b	no	1,0,3,6,0,4	_b / 10
banana_ba	ba	yes	no change	none
banana_ban	ban	no	1,0,3,6,0,4,5	ban / 11
banana_band	nd	no	1,0,3,6,0,4,5,3	nd / 12
banana_banda	da	no	1,0,3,6,0,4,5,3,2	da / 13
banana_bandan	an	yes	no change	none
banana_bandana	ana	yes	1,0,3,6,0,4,5,3,2,8	none

Notice that after the last character,"a", is read, the final
substring, "ana",
must be output.

---

top

Uncompression

The uncompression process for LZW is also straightforward. In addition, it has an advantage over static compression methods because no dictionary or other overhead information is necessary for the decoding algorithm--a dictionary identical to the one created during compression is reconstructed during the process. **Both encoding and decoding programs must start with the same initial dictionary,** in this case, all 256 ASCII characters.

Here's how it works. The LZW decoder first reads in an index (integer), looks up the index in the dictionary, and outputs the substring associated with the index. The first character of this substring is concatenated to the current working string. This new concatenation is added to the dictionary (resimulating how the substrings were added during compression). The decoded string then becomes the current working string (the current index, ie. the substring, is remembered), and the process repeats.

Again, here's what it might look like:

```
string entry;
char ch;
int prevcode, currcode;
...
```

```
prevcode = read in a code;
decode/output prevcode;
while (there is still data to read)
{
   currcode = read in a code;
   entry = translation of currcode from dictionary;
   output entry;
   ch = first char of entry;
   add ((translation of prevcode)+ch) to dictionary;
   prevcode = currcode;
}
```

There is an exception where the algorithm fails, and that is when the code calls for an index which has not yet been entered (eg. calling for an index 31 when index 31 is currently being processed and therefore not in the dictionary yet). An example from Sayood will help illustrate this point. Suppose you had the string *abababab*..... and an initial dictionary of just *a* & *b* with indexes 0 & 1, respectively. The encoding process begins:

Input	Current String	Seen this Before?	Encoded Output	New Dictionary Entry/Index
a	a	yes	nothing	none
ab	ab	no	0	ab / 2
aba	ba	no	0,1	ba / 3
abab	ab	yes	no change	none
ababa	aba	no	0,1,2	aba / 4
ababab	ab	yes	no change	none
abababa	aba	yes	no change	none
abababab	abab	no	0,1,2,4	abab / 5
...	...	...	...	...

So, the encoded output starts out *0,1,2,4,...* . When we start
trying to decode, a problem arises (in the table below, keep in mind that the *Current String* is just the substring that was
decoded/translated in the last pass of the loop. Also, the *New Dictionary Entry* is created by concatenating the *Current String* with the first character of the new *Dictionary Translation*):

Encoded Input	Dictionary Translation	Decoded Output	Current String	New Dictionary Entry / Index
0	0 = a	a	none	none
0,1	1 = b	ab	a	ab / 2
0,1,2	2 = ab	abab	b	ba / 3
0,1,2,4	4 = ???	abab???	ab	???

As you can see, the decoder comes across an index of 4 while the entry that belongs there is currently being processed. To understand why this happens, take a look at the encoding table. Immediately after "aba" (with an index of 4) is entered into the dictionary, the next substring that is encoded is an "aba" (ie. the very next code written to the encoded output file is a 4). Thus, the only case in which this special case can occur is if the substring begins and ends with the same character ("aba" is of the form <char><string><char>). So, to deal with this exception, you simply take the substring you have so far, "ab", and concatenate its first character to itself, "ab"+"a" = "aba", instead of following the procedure as normal. Therefore the pseudocode provided above must be altered a bit in order to handle all cases.