LZW Data Compression

Another approach to lossless compression, especially for text, takes advantage of information that recurs in the text, though not necessarily in consecutive locations. Consider, for example, a famous quotation from John F. Kennedy's inaugural address:

> Ask not what your country can do for you—ask what you can do for your country.

Except for the word *not*, each word in the quotation appears twice. Suppose we made a table of the words:

index	word
1	ask
2	not
3	what
4	your
5	country
6	can
7	do
8	for
9	you

Then we could encode the quotation (ignoring capitalization and punctuation) by

1 2 3 4 5 6 7 8 9 1 3 9 6 7 8 4 5

Because this quotation consists of few words, and a byte can hold integers ranging from 0 to 255, we can store each index in a single byte. Thus, we can store this quotation in only 17 bytes, one byte per word, plus whatever space we need to store the table. At one character per byte, the original quotation, without punctuation but with spaces between words, requires 77 bytes.

Of course, the space to store the table matters, for otherwise we could just number every possible word and compress a file by storing only indices of words. For some words, this scheme expands, rather than compresses. Why? Let's be ambitious and assume that there are fewer than 2^{32} words, so that we can store each index in a 32-bit word. We would represent each word by four bytes, and so this scheme loses for words that are three letters or shorter, which require only one byte per letter, uncompressed.

The real obstacle to numbering every possible word, however, is that real text includes "words" that are not words, or rather, not words in the English language. For an extreme example, consider the opening quatrain of Lewis Carroll's "Jabberwocky":

> 'Twas brillig, and the slithy toves
> Did gyre and gimble in the wabe:
> All mimsy were the borogoves,
> And the mome raths outgrabe.

Consider also computer programs, which often use variable names that are not English words. Add in capitalization, punctuation, and *really* long place names,[1] and you can see that if we try to compress text by numbering every possible word, we're going to have to use *a lot* of indices. Certainly more than 2^{32} and, because any combination of characters *could* appear in text, in reality an unbounded amount.

All is not lost, however, for we can still take advantage of recurring information. We just have to not be so hung up on recurring *words*. Any recurring sequence of characters could help. Several compression schemes rely on recurring character sequences. The one we'll examine is known as ***LZW***,[2] and it's the basis for many compression programs used in practice.

LZW makes a single pass over its input for compression and for decompression. In both, it builds a dictionary of character sequences that it has seen, and it uses indices into this dictionary to represent character sequences. Think of the dictionary as an array of character strings. We can index into this array, so that we can speak of its ith entry. Toward the beginning of the input, the sequences tend to be short, and representing the sequences by indices could result in expansion, rather than compression. But as LZW progresses through its input, the sequences in the dictionary become longer, and representing them by an index can save quite a bit of space. For example, I ran the text of *Moby Dick*

---

[1] Such as Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch, a Welsh village.

[2] As you probably guessed, the name honors its inventors. Terry Welch created LZW by modifying the LZ78 compression scheme, which was proposed by Abraham Lempel and Jacob Ziv.

through an LZW compressor, and it produced in its output an index representing the 10-character sequence ␣from␣the␣ 20 times. (Each ␣ indicates one space character.) It also output an index representing the eight-character sequence ␣of␣the␣ 33 times.

Both the compressor and decompressor seed the dictionary with a one-character sequence for each character in the character set. Using the full ASCII character set, the dictionary starts with 256 single-character sequences; the ith entry in the dictionary holds the character whose ASCII code is i.

Before going into a general description of how the compressor works, let's look at a couple of situations it handles. The compressor builds up strings, inserting them into the dictionary and producing as output indices into the dictionary. Let's suppose that the compressor starts building a string with the character `T`, which it has read from its input. Because the dictionary has every single-character sequence, the compressor finds `T` in the dictionary. Whenever the compressor finds the string that it's building in the dictionary, it takes the next character from the input and appends that character to the string it's building up. So now let's suppose that the next input character is `A`. The compressor appends `A` to the string it's building, getting `TA`. Let's suppose that `TA` is also in the dictionary. The compressor then reads the next input character, let's say `G`. It appends `G` to the string it's building, resulting in `TAG`, and this time let's suppose that `TAG` is *not* in the dictionary. The compressor does three things: (1) it outputs the dictionary index of the string `TA`; (2) it inserts the string `TAG` into the dictionary; and (3) it starts building a new string, initially containing just the character (`G`) that caused the string `TAG` to not be in the dictionary.

Here is how the compressor works in general. It produces a sequence of indices into the dictionary. Concatenating the strings at these indices gives the original text. The compressor builds up strings in the dictionary one character at a time, so that whenever it inserts a string into the dictionary, that string is the same as some string already in the dictionary but extended by one character. The compressor manages a string s of consecutive characters from the input, maintaining the invariant that the dictionary always contains s in some entry. Even if s is a single character, it appears in the dictionary, because the dictionary is seeded with a single-character sequence for each character in the character set. Initially, s is just the first character of the input. Upon reading a new character c, the compressor checks to see whether the string $s\,c$, formed by appending c to the end of s, is currently in the dictionary. If it is, then it appends c to the end of s and calls the result s; in other words, it sets s to $s\,c$. The compressor is building a longer string that it will eventually insert into the dictionary. Otherwise, s is in the dictionary but $s\,c$ is not. In this case, the compressor outputs the index of s in the dictionary, inserts $s\,c$ into the next available dictionary entry, and sets s to just the input character c. By inserting $s\,c$ into the dictionary, the compressor has added a string that extends s by one character, and by then setting s to c, it restarts the process of building a string to look up in the dictionary. Because c is a single-character string in the dictionary, the compressor maintains the invariant that s appears somewhere in the dictionary. Once the input is exhausted, the compressor outputs the index of whatever string s remains.

The procedure LZW-COMPRESSOR appears on the next page. Let's run through an example, compressing the text `TATAGATCTTAATATA`. (The sequence `TAG` that we saw on the previous page will come up.) The following table shows what happens upon each iteration of the loop in step 3. The values shown for the string s are at the start of the iteration.

Iteration	s	c	Output	New dictionary string
1	T	A	84 (T)	256: TA
2	A	T	65 (A)	257: AT
3	T	A		
4	TA	G	256 (TA)	258: TAG
5	G	A	71 (G)	259: GA
6	A	T		
7	AT	C	257 (AT)	260: ATC
8	C	T	67 (C)	261: CT
9	T	T	84 (T)	262: TT
10	T	A		
11	TA	A	256 (TA)	263: TAA
12	A	T		
13	AT	A	257 (AT)	264: ATA
14	A	T		
15	AT	A		
step 4	ATA		264 (ATA)	

After step 1, the dictionary has one-character strings for each of the 256 ASCII characters in entries 0 through 255. Step 2 sets the string s to hold just the first input character, `T`. In the first iteration of the main loop of step 3, c is the next input character, `A`. The concatenation $s\,c$ is the string `TA`, which is not yet in the dictionary, and so step 3C runs. Be-

Procedure LZW-COMPRESSOR(*text*)

Input: text: A sequence of characters in the ASCII character set.

Output: A sequence of indices into a dictionary.

1. For each character c in the ASCII character set:
   A. Insert c into the dictionary at the index equal to c's numeric code in ASCII.
2. Set s to the first character from *text*.
3. While *text* is not exhausted, do the following:
   A. Take the next character from *text*, and assign it to c.
   B. If $s\,c$ is in the dictionary, then set s to $s\,c$.
   C. Otherwise ($s\,c$ is not yet in the dictionary), do the following:
      i. Output the index of s in the dictionary.
      ii. Insert $s\,c$ into the next available entry in the dictionary.
      iii. Set s to the single-character string c.
4. Output the index of s in the dictionary.

cause the string s holds just `T`, and the ASCII code of `T` is 84, step 3Ci outputs the index 84. Step 3Cii inserts the string `TA` into the next available entry in the dictionary, which is at index 256, and step 3Ciii restarts building s, setting it to just the character `A`. In the second iteration of the loop of step 3, c is the next input character, `T`. The string $s\,c =$ `AT` is not in the dictionary, and so step 3C outputs the index 65 (the ASCII code for `A`), inserts the string `AT` into entry 257, and sets s to hold `T`.

We see the benefit of the dictionary upon the next two iterations of the loop of step 3. In the third iteration, c becomes the next input character, `A`. Now the string $s\,c =$ `TA` is present in the dictionary, and so the procedure doesn't output anything. Instead, step 3B appends the input character onto the end of s, setting s to `TA`. In the fourth iteration, c becomes `G`. The string $s\,c =$ `TAG` is not in the dictionary, and so step 3Ci outputs the dictionary index 256 of s. One output number gives not one, but two characters: `TA`.

Not every dictionary index is output by the time LZW-COMPRESSOR finishes, and some indices may be output more than once. If you concatenate all the characters in parentheses in the output column, you get the original text, `TATAGATCTTAATATA`.

This example is a little too small to show the real benefit of LZW compression. The input occupies 16 bytes, and the output consists of 10 dictionary indices. Each index requires more than one byte. Even if we use two bytes per index in the output, it occupies 20 bytes. If each index occupies four bytes, a common size for integer values, the output takes 40 bytes.

Longer texts tend to yield better results. LZW compression reduces the size of *Moby Dick* from 1,193,826 bytes to 919,012 bytes. Here, the dictionary contains 230,007 entries, and so indices have to be at least four bytes.[3] The output consists of 229,753 indices, or 919,012 bytes. That's not as compressed as the result of Huffman coding (673,579 bytes), but we'll see some ideas a little later to improve the compression.

LZW compression helps only if we can decompress. Fortunately, the dictionary does not have be stored with the compressed information. (If it did, unless the original text contained a huge amount of recurring strings, the output of LZW compression plus the dictionary would constitute an expansion, not a compression.) As mentioned earlier, LZW decompression rebuilds the dictionary directly from the compressed information.

Here is how LZW decompression works. Like the compressor, the decompressor seeds the dictionary with the 256 single-character sequences corresponding to the ASCII character set. It reads a sequence of indices into the dictionary as its input, and it mirrors what the compressor did to build the dictionary. Whenever it produces output, it's from a string that it has added to the dictionary.

Most of the time, the next dictionary index in the input is for an entry already in the dictionary (we'll soon see what happens the rest of the time), and so the LZW decompressor finds the string at that index in the dictionary and outputs it. But how can it build the dictionary? Let's think for a moment about how the compressor operates. When it outputs an index within step 3C, it has found that, although the string s is in the dictionary, the string $s\,c$ is not. It outputs the index of s in the dictionary, inserts $s\,c$ into the dictionary, and starts building a new string

---

[3] I'm assuming that we represent integers using the standard computer representations of integers, which occupy one, two, four, or eight bytes. In theory, we could represent indices up to 230,007 using just three bytes, so that the output would take 689,259 bytes.

to store, starting with c. The decompressor has to match this behavior. For each index it takes from its input, it outputs the string s at that index in the dictionary. But it also knows that at the time the compressor output the index for s, the compressor did not have the string $s\,c$ in the dictionary, where c is the character immediately following s. The decompressor knows that the compressor inserted the string $s\,c$ into the dictionary, so that's what the decompressor needs to do—eventually. It cannot insert $s\,c$ yet, because it hasn't seen the character c. That's coming as the first character of the next string that the decompressor will output. But the decompressor doesn't have that next string just yet. Therefore, the decompressor needs to keep track of two consecutive strings that it outputs. If the decompressor outputs strings X and Y, in that order, then it concatenates the first character of Y onto X and then inserts the resulting string into the dictionary.

Let's look at an example, referring to the table on page 4, which shows how the compressor operates on `TATAGATCTTAATATA`. In iteration 11, the compressor outputs the index 256 for the string `TA`, and it inserts the string `TAA` into the dictionary. That's because, at that time, the compressor already had $s =$ `TA` in the dictionary but not $s\,c =$ `TAA`. That last `A` begins the next string output by the compressor, `AT` (index 257), in iteration 13. Therefore, when the decompressor sees indices 256 and 257, it should output `TA`, and it also should remember this string so that when it outputs `AT`, it can concatenate the `A` from `AT` with `TA` and insert the resulting string, `TAA`, into the dictionary.

On rare occasions, the next dictionary index in the decompressor's input is for an entry not yet in the dictionary. This situation arises so infrequently that when decompressing *Moby Dick*, it occurred for only 15 of the 229,753 indices. It happens when the index output by the compressor is for the string most recently inserted into the dictionary. This situation occurs only when the string at this index starts and ends with the same character. Why? Recall that the compressor outputs the index for a string s only when it finds s in the dictionary but $s\,c$ is not, and then it inserts $s\,c$ into the dictionary, say at index i, and begins a new string s starting with c. If the next index output by the compressor is going to be i, then the string at index i in the dictionary must start with c, but we just saw that this string is $s\,c$. So if the next dictionary index in the decompressor's input is for an entry not yet in the dictionary, the decompressor can output the string it had most recently output, concatenated with the first character of this string, and insert this new string into the dictionary.

Because these situations are so rare, an example is a bit contrived. The string TATATAT causes it to occur. The compressor does the following: outputs index 84 (T) and inserts TA at index 256; outputs index 65 (A) and inserts AT at index 257; outputs index 256 (TA) and inserts TAT at index 258; and finally outputs index 258 (TAT—the string just inserted). The decompressor, upon reading in index 258, takes the string it had most recently output, TA, concatenates the first character of this string, T, outputs the resulting string TAT, and inserts this string into the dictionary.

Although this rare situation occurs only when the string starts and ends with the same character, this situation does not occur every time the string starts and ends with the same character. For example, when compressing *Moby Dick*, the string whose index was output had the same starting and ending character 11,376 times (a shade under 5% of the time) without being the string most recently inserted into the dictionary.

The procedure LZW-DECOMPRESSOR, on the next page, makes all of these actions precise. The following table shows what happens in each iteration of the loop in step 4 when given as input the indices in the output column in the table on page 4. The strings indexed in the dictionary by *previous* and *current* are output in consecutive iterations, and the values shown for *previous* and *current* in each iteration are after step 4B.

Iteration	*previous*	*current*	Output (s)	New dictionary string
Steps 2, 3		84	T	
1	84	65	A	256: TA
2	65	256	TA	257: AT
3	256	71	G	258: TAG
4	71	257	AT	259: GA
5	257	67	C	260: ATC
6	67	84	T	261: CT
7	84	256	TA	262: TT
8	256	257	AT	263: TAA
9	257	264	ATA	264: ATA

Except for the last iteration, the input index is already in the dictionary, so that step 4D runs only in the last iteration. Notice that the dictionary built by LZW-DECOMPRESSOR matches the one built by LZW-COMPRESSOR.

Procedure LZW-DECOMPRESSOR(*indices*)

Input: indices: a sequence of indices into a dictionary, created by LZW-COMPRESSOR.

Output: The text that LZW-COMPRESSOR took as input.

1. For each character c in the ASCII character set:
   A. Insert c into the dictionary at the index equal to c's numeric code in ASCII.
2. Set *current* to the first index in *indices*.
3. Output the string in the dictionary at index *current*.
4. While *indices* is not exhausted, do the following:
   A. Set *previous* to *current*.
   B. Take the next number from *indices* and assign it to *current*.
   C. If the dictionary contains an entry indexed by *current*, then do the following:
      i. Set s to be the string in the dictionary entry indexed by *current*.
      ii. Output the string s.
      iii. Insert, into the next available entry in the dictionary, the string at the dictionary entry indexed by *previous*, concatenated with the first character of s.
   D. Otherwise (the dictionary does not yet contain an entry indexed by *current*), do the following:
      i. Set s to be the string at the dictionary entry indexed by *previous*, concatenated with the first character of this dictionary entry.
      ii. Output the string s.
      iii. Insert, into the next available entry in the dictionary, the string s.

I haven't addressed how to look up information in the dictionary in the LZW-COMPRESSOR and LZW-DECOMPRESSOR procedures. The latter is easy: just keep track of the last dictionary index used, and if the index in *current* is less than or equal to the last-used index, then the string is in the dictionary. The LZW-COMPRESSOR procedure has a more difficult task: given a string, determine whether it's in the dictionary and, if it is, at what index. Of course, we could just perform a linear search on the dictionary, but if the dictionary contains n items, each linear search takes $O(n)$ time. We can do better by using either one of a couple of data structures. I won't go into the details here, however. One is called a ***trie***, and it's like the binary tree we built for Huffman coding, except that each node can have many children, not just two, and each edge is labeled with an ASCII character. The other data structure is a ***hash table***, and it provides a simple way to find strings in the directory that is fast on average.