# The Lempel-Ziv-Welch Algorithm

Group 1 - Halliwell, Panova, Sudmann-Day

**1- Overview**

Data compression is an important topic in the field of computer science. In the era of Big Data, algorithms for data compression continue to maintain importance. However, writing efficient, lossless data compression algorithms is not an easy task. The Lempel-Ziv Algorithm was created by Abraham Lempel and Jacob Ziv in 1977 and 1978 as just such a lossless data compression algorithm. This algorithm was eventually modified by Terry Welch who made minor adjustments to improve efficiency, thereby becoming the Lempel-Ziv-Welch Algorithm (LZW). The resulting algorithm is widely used for large text files with repeating characters, along with GIF and TIFF image formats.

The LZW algorithm takes advantage of the repetition of multiple characters in a file, allowing storage and transmission of fewer bytes. The algorithm is built around the concept of a *dictionary* containing text segments that are constructed during the compression process. The beauty of the algorithm is that this dictionary can be discarded after compression has completed because the reader of the content can reconstruct the dictionary as a just-in-time during decompression. The algorithm reads through the text one character at a time, never having to revisit any part of the text; this classifies the algorithm as 'greedy' and contributes to its efficiency.

**2 - Compression**

The compression portion of LZW algorithm operates as follows:

- **Input:** A text string or text file.
- **Assumption:** All characters in the text belonging to the UTF-8 character set.
- **Steps:**

(1) Create a **dictionary**, populating it initially with all characters in the character set, along with their index: { …, [a,65], [b,66], [c,67], … }. The index starts at 1 and continues up to 256.

(2) Create an empty list of integers. This is the **output** list that will be returned from the function.

(3) Read each character in the input text. For each **character**:

(a) Create the **current "word"** that we are working with. This word is created by appending the current **character** to the **previous word** (the current word from the previous iteration).

(b) If the **current word** is found in the dictionary, set the **previous word** to be the **current word** and return to the start of the loop to get the next **character**.

(c) Lookup the **previous word** in the dictionary to get its index. This is appended to the **output list**.

(d) Append the **current word** to the **dictionary**.
*[Note: The previous two steps comprise the counterintuitive nature of this algorithm. There is effectively a delay; the **current word** does not appear unbroken in the output until it is seen for the second time, but it is necessary to put it in the dictionary even if it is never used again. This is because the decompression algorithm will follow exactly the same steps, and the indexes of both dictionaries need to be identical for decompression to be correct. This is a small cost to pay for not having to transmit the dictionary.]*

(e) Set the **previous word** to the current character. This **character** is the only part of the **current word** that has not yet been committed to the **output list**. By setting the **previous word** to this **character**, we ensure its processing in the next iteration.

(4) If the loop ends with any remaining **character** in the **previous word**, it will have been placed in the **dictionary** already, but not in the **output list**. Simply add it to the **output list**.

- **Output:** Only the output list containing indexes to the dictionary, *even though* we were able to discard the dictionary.

**3 - Decompression**

The decompression portion of the LZW algorithm operates as follows:

- **Input:** A list of indexes (the output from the compression algorithm).
- **Assumption:** All characters in the text belonging to the UTF-8 character set.
- **Steps:**

(1) Create a **dictionary**, populating it initially with all characters in the character set, along with their index: { …, [65,a], [66,b], [67,c], … }. This is equivalent to the first step in the compression algorithm, except that we now lookup text by index, rather than index by text.

(2) Read each **index** from the input:

(a) We have to start the loop handling a special case that usually does not occur; it occurs only in certain cases of repeating characters. This is perhaps best understood after having read the rest of the steps in the loop. If the **index** that has been read does not exist in the **dictionary**, add the **index** to the **dictionary** and associate the following text with that index: the first character of the **previous word** appending to the end of the same word.

(a) Set the **current "word"** to be the item in the **dictionary** that corresponds to the current **index**.

(b) Determine the **dictionary value** that we expect to find in, or add to, the **dictionary**. The **dictionary value** gets set to the **previous word** appended by the first character of the **current word**.

(c) If the **dictionary value** is not found in the **dictionary**, add it and generate a new index in the process.

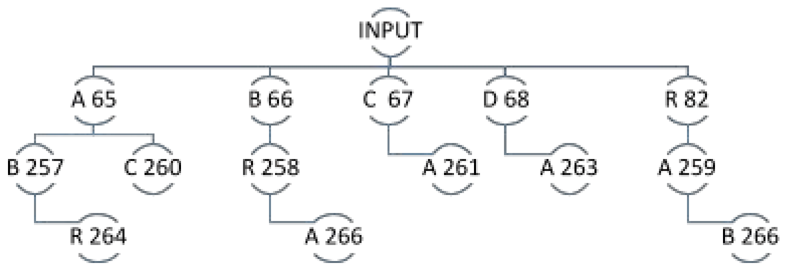
(d) Append the **current word** to the **output text**.

- **Output:** The text constructed by the loop above.

## 4 - Discussion

The LZW algorithm is an application of a suffix tree, a data structure that stores the suffixes of a given text. The suffix tree methodology is used to solve exact matching problems in linear time, such as finding the longest substring of a string or compressing a string.[1] There are different ways to construct a suffix tree, but the most famous and widely used was proposed in 1995 by Esco Ukkonen, a Finnish Computer-Science Professor.[2] His procedure was an "online" algorithm, meaning that the algorithm can "process its input piece-by-piece in a serial fashion, i.e., in the order that the input is fed to the algorithm, without having the entire input available from the start."[3] Because the dictionary in the LZW algorithm can be represented as a suffix tree, its completeness is easily proven. We start with an implicit suffix tree which contains the first suffix, in our case, the first letter. Then we continue step by step through the text adding the new inputs until it is complete. The original Ukkonen algorithm uses suffixes and works backwards, whereas LWZ adds new suffixes moving forward. This simple and greedy strategy reads off the longest recognized input each time. In order to illustrate that LZW is indeed complete, let's represent the dictionary for the text "ABRACADABRABRABRA" as a suffix tree:

[1] Gusfield,p.89
[2] Ukkonen, E. (1995)
[3] Karp, (1992)



**5 - Complexity**

First, we will look at the compression portion. The outer loop visits each character in the text one time only. At the end of the text, the algorithm is complete. This outer loop carries a complexity of O(n) where n is the number of characters in the text. Within that loop, a dictionary is constructed. Adding items to end of the dictionary is O(1), but determining if an item is already in the dictionary is O(log n). This gives us the following result:

> **O(n log n)** for compression
>
> [Note: we found some resources on line claiming that the dictionary has fixed length and therefore the complexity is O(n). We disagree.]

The overall complexity of LZW depends upon the implementation of this dictionary. Given the domain-specific nature of the content (which might not actually be text), this dictionary could be further optimized for its domain, file format, language, etc. Although the official complexity may remain at O(n log n), the practical improvements could be significant.

Second we'll look at the decompression portion. Determining the complexity of decompression follows very similar logic to that of compression. The outer loop visits each index in the input one time only carrying a complexity of O(n). We also, lookup elements in the dictionary and add missing ones. The difference, however, is that when we look up, we are using an integer index that is contiguous beginning with zero. This enables dictionary lookups to perform individually at O(1). This gives us the following result:

> **O(n)** for decompression

**6 - Completeness**

To help discuss the level of compression obtained with the LZW algorithm, let's define some notation:

> **n** = the length of the string which we have to compress
>
> **α** = the length of the initial dictionary
>
> **β** = the length of the dynamic part of the dictionary

**$U(\alpha, \beta)$** = the full length of the dictionary

**d** = the length of the active portion of the dictionary (many dictionary entries are not included in the final output and we consider these “inactive”)

Thus, **$\text{length}(d) \leq \text{length}(U(\alpha, \beta))$.**

We store each character as combinations of zeros and ones. In the case of the UTF-8 character set, we have an 8-bit system, meaning that to store each character we need 8 digits. Since the alphabet consist of **α** entries, the memory that each character occupies is

**$\log_2\alpha$**. (i.e. $\log_2 256=8$)

Therefore, the memory that the uncompressed string consumes is:

**$n\log_2\alpha$** (we have n symbols in the uncompressed string).

What about the compressed string? Let’s define

**$c=\text{ceiling}(\log_2(\beta/ \alpha))$** where ceiling is a function mapping a real number to the smallest subsequent integer**.**

Within the structure of the dictionary, we have consumed the initial eight bits in the initial population of all UTF-8 characters. Therefore we need to allocate additional bits to store additional indexes as the dictionary size increases.

New dictionary entries, beyond the initial set, will consume **$\log_2 c^*\alpha$** in the output. Therefore, the memory which the compressed string will occupy is **$d \log_2 2^c\alpha$.**

To prove that compression has occurred, the memory occupied by the compressed string must be less than the one occupied by the uncompressed string.

$$d \log_2 2^c\alpha < n \log_2\alpha$$

$$d(c + \log_2\alpha) < n\log_2\alpha$$

$$\frac{d}{n}(c+1) < 1$$

Assume the inequality is true, increasing **n** causes an increase in **d** but by a smaller fraction. A change in **n** also results in a change to **c** by a small amount. In order for **c** to change we need to have the new entries be as big as the initial alphabet. If we consider the change in **c** as insignificant when **n** approaches infinity, we can state that the inequality holds whenever **n** is bigger than **d**. In other words, LZW is asymptotically optimal.

## 7 - Our Observations

We performed a number of tests against our implementation, including special cases. As you can see from our results below, smaller strings yield less (and often negative) compression while larger strings, especially involving repeating content, produce significant savings. The “worst case” compression calculations used the formula (old-new)/old so that positive number is positive savings and assumed that all indexes would consume 16 bits. However, a lower-level implementation could be optimize this further based on the calculations described in the previous section.

| **Text** | **Text Length** | **Worst-Case Compression** |
|---|---|---|
| {empty string} | 0 | 0% |
| A | 1 | -100% |
| AA | 2 | -100% |
| AAA | 3 | -33% |
| AB | 2 | -100% |
| ABA | 3 | -100% |
| ABBA | 4 | -100% |
| ABBBBBBBBBBBBBBBBBA | 19 | 15% |
| AAAAAAAAAAAAAAAAAAA | 19 | 36% |
| ABCABCABCABCABCABC | 18 | 0% |
| The brown dog jumped... | 61 | -77% |
| {DNA} TGATGATGAA… | 990 | 35% |
| Linear programming, surprisingly, … | 1218 | -5% |
| {16 times the previous text} | 19488 | 53% |

## 8 - References

Gusfield, Dan. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge: Cambridge UP, 1997. Print.

Karp, Richard M. (1992). "On-line algorithms versus off-line algorithms: How much is it worth to know the future?" (PDF). *IFIP Congress (1)* **12**: 416–429. Retrieved 17 August 2015.

Ukkonen, Esko. "On-line Construction of Suffix Trees." *Algorithmica* (1995): n. pag. Web. 11 Dec. 2015.