7.2 Adaptive Dictionary

Most adaptive-dictionary-based techniques have their roots in two landmark papers by Jacob Ziv and Abraham Lempel in 1977 and 1978. These papers provide two different approaches to adaptively building dictionaries, and each approach has given rise to a number of variations. The approaches based on the 1977 paper are said to belong to the LZ77 family (also known as LZ1), while the approaches based on the 1978 paper are said to belong to the LZ78, or LZ2, family.

7.2.1 LZ77

In the LZ77 approach, the dictionary is simply a portion of the previously encoded sequence. The encoder examines the input sequence through a sliding window as shown in Figure 5.1. The window consists of two parts, a *search buffer* that contains a portion of the recently encoded sequence, and a *look-ahead buffer*that contains the next portion of the sequence to be encoded. In Figure 5.1, the search buffer contains eight symbols, while the look-ahead buffer contains sevensymbols. In practice, the sizes of the buffers are significantly larger; however, forthe purpose of explanation, we will keep the buffer sizes small.

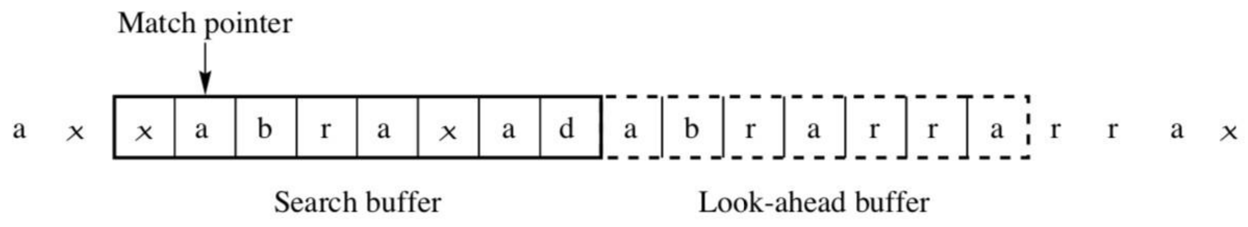


FIGURE 5. 1 Encoding using the LZ77 approach.

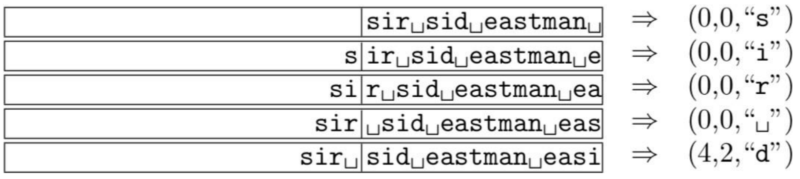
To encode the sequence in the look-ahead buffer, the encoder moves a search pointer back through the search buffer until it encounters a match to the first symbol in the look-ahead buffer. The distance of the pointer from the look-ahead buffer is called the *offset*. The encoder then examines the symbols following the symbol at the pointer location to see if they match consecutive symbols in the look-ahead buffer. The number of consecutive symbols in the search buffer that match consecutive symbols in the look-ahead buffer, starting with the first symbol, is called the *length of the match*. The encoder searches the search buffer for the longest match. Once the longest match has been found, the encoder encodes it with a triple $<o, l, c>$, where o is the *offset*, l is the *length of the match*, and c is the codeword corresponding to the symbol in the look-ahead buffer that follows the match. For example, in Figure 5.1 the pointer is pointing to the beginning of the longest match. The offset o in this case is 7, the length of the match l is 4, and the symbol in the look-ahead buffer following the match is *r*.

The reason for sending the third element in the triple is to take care of the situation where no match for the symbol in the look-ahead buffer can be found in the search buffer. In this case, the offset and match-length values are set to 0, and the third element of the triple is the code for the symbol itself.

If the size of the search buffer is S, the size of the window (search and look-ahead buffers) is W , and the size of the source alphabet is A, then the number of bits needed to code the triple using fixed-length codes is $[\log_2 S]+[\log_2 W]+[\log_2 A]$. Notice that the second term is $[\log_2 W]$, not $[\log_2 S]$. The reason for this is that the length of the match can actually exceed the length of the search buffer. We will see how this happens in Example 5.4.1. In the following example, we will look at three different possibilities that may be encountered during the coding process:

1. There is no match for the next character to be encoded in the window.
2. There is a match.
3. The matched string extends inside the look-ahead buffer.

Explanation Example:



	sir␣sid␣eastman␣	⇒	(0,0,"s")
s	ir␣sid␣eastman␣e	⇒	(0,0,"i")
si	r␣sid␣eastman␣ea	⇒	(0,0,"r")
sir	␣sid␣eastman␣eas	⇒	(0,0,"␣")
sir␣	sid␣eastman␣easi	⇒	(4,2,"d")

Example: Suppose the sequence to be encoded is … *cabracadabrarrarrad*…

Suppose the length of the window is 13, the size of the look-ahead buffer is six, and the current condition is as follows:

cabraca	*dabrar*

with *dabrar* in the look-ahead buffer. We look back in the already encoded portion of the window to find a match for d. As we can see, there is no match, so we transmit the triple <0,0,C(d)>. The first two elements of the triple show that there is no match to d in the search buffer, while C(d) is the code for the character d. For now, let's continue with the encoding process. As we have encoded a single character, we move the window by one character. Now the contents of the buffer are

abracad	*abrarr*

with *abrarr* in the look-ahead buffer. Looking back from the current location, we find a match to a at an offset of two. The length of this match is one. Looking further back, we have another match for a at an offset of four; again the length of the match is one. Looking back even further in the window, we have a third match for a at an offset of seven. However, this time the length of the match is four (see Figure 5.2). So we encode the string *abra* with the triple <7, 4, C(r)>, and move the window forward by five characters. The window now contains the following characters:

adabrar	*rarrad*

Now the look-ahead buffer contains the string *rarrad*. Looking back in the window, we find a match for *r* at an offset of one and a match length of one, and a second match at an offset of three with a match length of what at first appears to be three. It turns out we can use a match length of five instead of three.

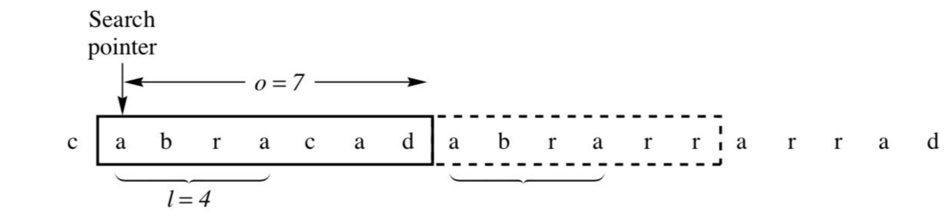


FIGURE 5. 2 **The encoding process.**

Why this is so will become clearer when we decode the sequence. To see how the decoding works, let us assume that we have decoded the sequence *cabraca* and we receive the triples <0, 0, C(d)>, <7, 4, C(r)>, and <3, 5, C(d)>. The first triple is easy to decode; there was no match within the previously decoded string, and the next symbol is d. The decoded string is now *cabracad*. The first element of the next triple tells the decoder to move the copy pointer back seven characters and copy four characters from that point. The decoding process works as shown in Figure 5.3.

Finally, let’s see how the triple <3, 5, C(d)> gets decoded. We move back three characters and start copying. The first three characters we copy are *rar*. The copy pointer moves once again, as shown in Figure 5.4, to copy the recently copied character *r*. Similarly, we copy the next character *a*. Even though we started copying only three characters back, we end up decoding five characters. Notice that the match only has to start in the *search buffer*; it can extend into the look-ahead buffer. In fact, if the last character in the look-ahead buffer had been *r* instead of *d*, followed by several more repetitions of *rar*, the entire sequence of repeated *rar’s* could have been encoded with a single triple.

As we can see, the LZ77 scheme is a very simple adaptive scheme that requires no prior knowledge of the source and seems to require no assumptions about the characteristics of the source. At first it seems that this method does not make any assumptions about the input data. Specifically, it does not pay attention to any symbol frequencies. A little thinking, however, shows that because of the nature of the sliding window, the LZ77 method always compares the look-ahead buffer to the recently-input text in the search buffer and never to text that was input long ago (and has therefore been flushed out of the search buffer). Thus, the method implicitly assumes that patterns in the input data occur close together. Data that satisfies this assumption will compress well.

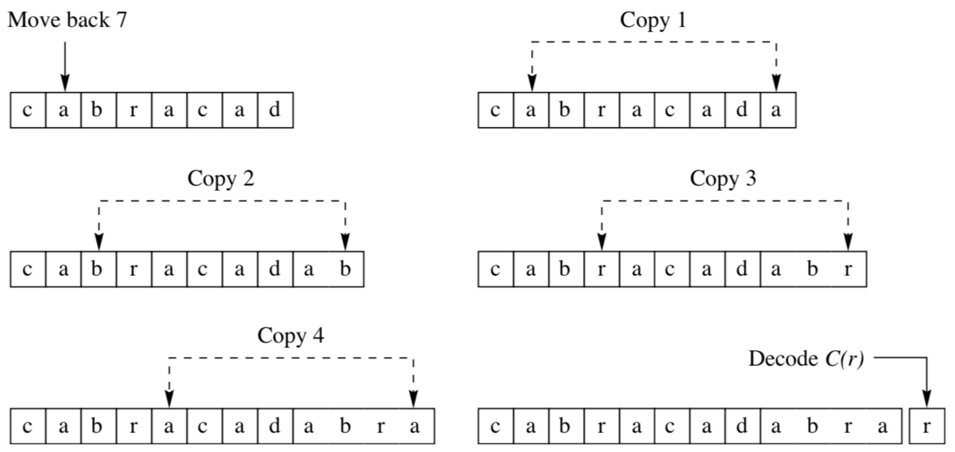


FIGURE 5.3 **Decoding of the triple ⟨7, 4, *C*(*r*)⟩.**

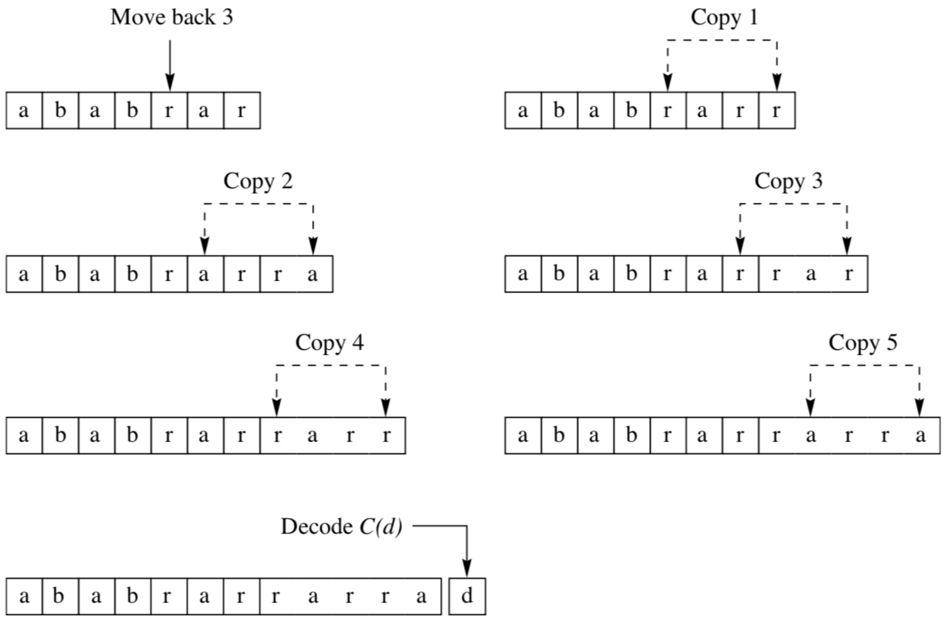


FIGURE 5.4 **Decoding the triple ⟨3, 5, *C*(*d*)⟩.**

The basic LZ77 method was improved in several ways by researchers and programmers during the 1980s and 1990s. One way to improve it is to use variable-size “offset” and “length” fields in the tokens. Another way is to increase the sizes of both buffers. Increasing the size of the search buffer makes it possible to find better matches, but the trade-off is an increased search time. A large search buffer therefore requires a more sophisticated data structure that allows for fast search. A third improvement has to do with sliding the window. The simplest approach is to move all the text in the window to the left after each match. A faster method is to replace the linear window with a circular queue, where sliding the window is done by resetting two pointers.

7.2.2 LZ78

The LZ77 approach implicitly assumes that like patterns will occur close together. It makes use of this structure by using the recent past of the sequence as the dictionary for encoding. However, this means that any pattern that recurs over a period longer than that covered by the coder window will not be captured. The worst-case situation would be where the sequence to be encoded was periodic with a period longer than the search buffer. Consider Figure 5.5.

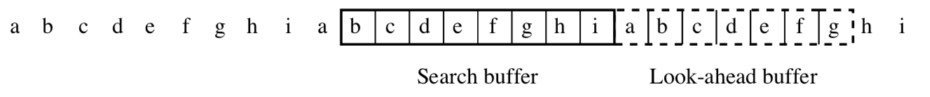


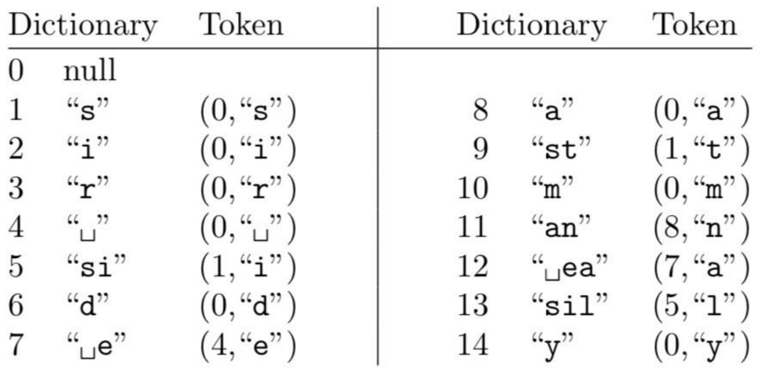
FIGURE 5.5 **The Achilles' heel of LZ77.**

This is a periodic sequence with a period of nine. If the search buffer had been just one symbol longer, this sequence could have been significantly compressed. As it stands, none of the new symbols will have a match in the search buffer and will have to be represented by separate codewords. As this involves sending along overhead (a triple for the LZ77 algorithm), the net result will be an expansion rather than a compression.

Although this is an extreme situation, there are less drastic circumstances in which the finite view of the past would be a drawback. The LZ78 algorithm solves this problem by dropping the reliance on the search buffer and keeping an explicit dictionary. This dictionary has to be built at both the encoder and decoder, and care must be taken that the dictionaries are built in an identical manner. The inputs are coded as a double <*i*, *c*>, with *i* being an index corresponding to the dictionary entry that was the longest match to the input, and *c* being the code for the character in the input following the matched portion of the input. As in the case of LZ77, the index value of 0 is used in the case of no match. This double then becomes the newest entry in the dictionary. Thus, each new entry into the dictionary is one new symbol concatenated with an existing dictionary entry. To see how the LZ78 algorithm works, consider the following example.

Explanation Example:

“sir␣sid␣eastman␣easily␣teases␣sea␣sick␣seals”.



Dictionary		Token	Dictionary		Token
0	null				
1	“s”	(0,“s”)	8	“a”	(0,“a”)
2	“i”	(0,“i”)	9	“st”	(1,“t”)
3	“r”	(0,“r”)	10	“m”	(0,“m”)
4	“␣”	(0,“␣”)	11	“an”	(8,“n”)
5	“si”	(1,“i”)	12	“␣ea”	(7,“a”)
6	“d”	(0,“d”)	13	“sil”	(5,“l”)
7	“␣e”	(4,“e”)	14	“y”	(0,“y”)

Table 3.10: First 14 Encoding Steps in LZ78.

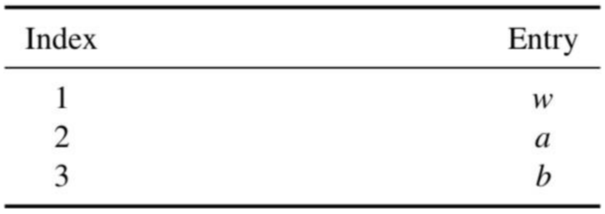
Example: Let us encode the following sequence using the LZ78 approach:

wabbaƀwabbaƀwabbaƀwabbaƀwooƀwooƀwoo

where *b/* stands for space. Initially, the dictionary is empty, so the first few symbols encountered are encoded with the index value set to 0. The first three encoder outputs are <0, C(w)>, <0, C(a)>, <0, C(b)>, and the dictionary looks like Table 5.4.

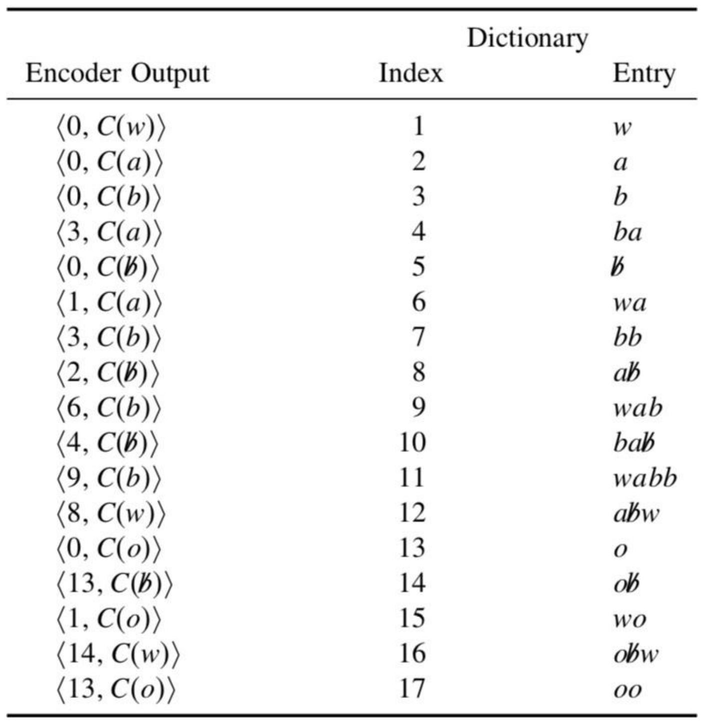
The fourth symbol is a *b*, which is the third entry in the dictionary. If we append the next symbol, we would get the pattern *ba*, which is not in the dictionary, so we encode these two symbols as <3, C(a)>, and add the pattern *ba* as the fourth entry in the dictionary. Continuing in this fashion, the encoder output and the dictionary develop as in Table 5.5. Notice that the entries in the dictionary generally keep getting longer, and if this particular sentence was repeated often, as it is in the song, after a while the entire sentence would be an entry in the dictionary.

TABLE 5.4 The initial dictionary.



Index	Entry
1	*w*
2	*a*
3	*b*

TABLE 5.5 Development of dictionary.



	Dictionary	
Encoder Output	Index	Entry
$\langle 0, C(w)\rangle$	1	*w*
$\langle 0, C(a)\rangle$	2	*a*
$\langle 0, C(b)\rangle$	3	*b*
$\langle 3, C(a)\rangle$	4	*ba*
$\langle 0, C(b\!\!\!/)\rangle$	5	$b\!\!\!/$
$\langle 1, C(a)\rangle$	6	*wa*
$\langle 3, C(b)\rangle$	7	*bb*
$\langle 2, C(b\!\!\!/)\rangle$	8	$ab\!\!\!/$
$\langle 6, C(b)\rangle$	9	*wab*
$\langle 4, C(b\!\!\!/)\rangle$	10	$bab\!\!\!/$
$\langle 9, C(b)\rangle$	11	*wabb*
$\langle 8, C(w)\rangle$	12	$ab\!\!\!/w$
$\langle 0, C(o)\rangle$	13	*o*
$\langle 13, C(b\!\!\!/)\rangle$	14	$ob\!\!\!/$
$\langle 1, C(o)\rangle$	15	*wo*
$\langle 14, C(w)\rangle$	16	$ob\!\!\!/w$
$\langle 13, C(o)\rangle$	17	*oo*

While the LZ78 algorithm has the ability to capture patterns and hold them indefinitely, it also has a rather serious drawback. As seen from the example, the dictionary keeps growing without bound. In a practical situation, we would have to stop the growth of the dictionary at some stage, and then either prune it back or treat the encoding as a fixed dictionary scheme.

7.2.3 LZW

There are a number of ways the LZ78 algorithm can be modified, and as is the case with the LZ77 algorithm, anything that can be modified probably has been. The most well-known modification, one that initially sparked much of the interest in the LZ algorithms, is a modification by Terry Welch known as LZW. Welch proposed a technique for removing the necessity of encoding the second element of the pair <i, c>. That is, the encoder would only send the index to the dictionary.

Before encoding/decoding process is done, firstly the encoder will initialize a dictionary with the basic symbols (characters) of the message, it’s usually 0-255 index of ASCII characters. So, the default value of the dictionary will be 256 ASCII characters, which means single character will be always encoded by an index from 0-255, and the string from 256 – 2^n (two power of n where n is the bit length being used) . So, if we use 10 bits, it means that there are 0-1024 possible index in dictionary to encode the characters or strings from a message. The length of the dictionary can be arbitrary, but Terry Welch suggested that it shouldn’t more than 12 bit, so the maximum index would be 4095. The problem is, what happens when the dictionary gets too large or full? The solution is to clear entries 256-4095 and start building the dictionary again where this approach must also be used by the decoder. Another solution is to let the dictionary full and only use the existing entries to encode the next symbols from the message.

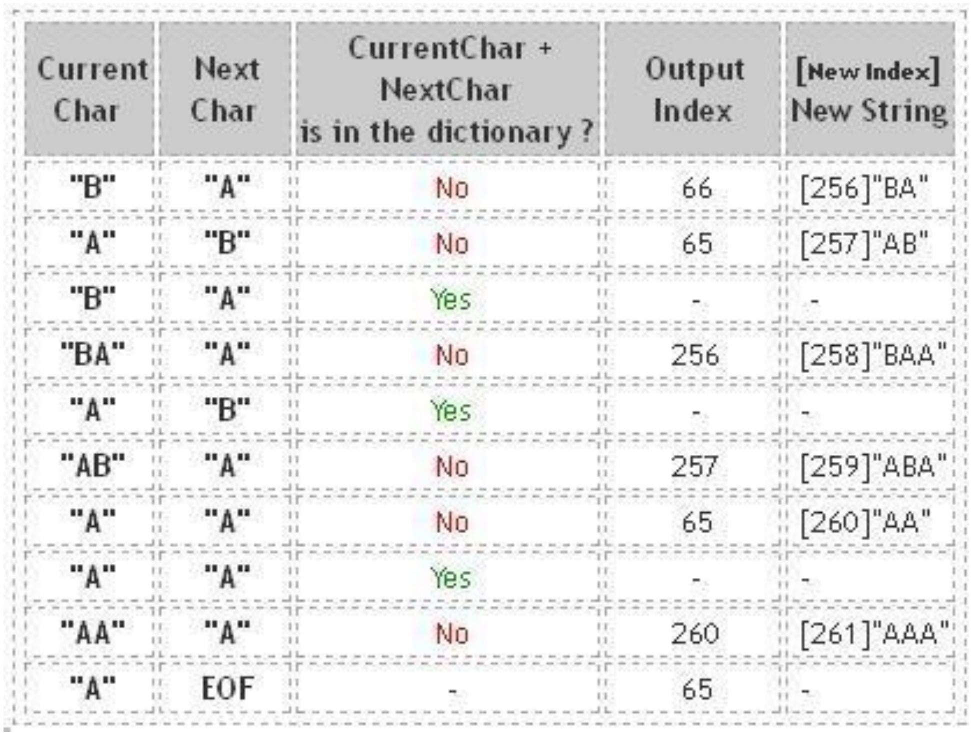
LZW Encoding Algorithm-Pseudo Code

```
Dictionary is initialized with the ASCII characters;
CurrentChar = The first character from the message;

while NOT EOF {
      NextChar = Read a character from the message;
      ConcatChars = CurrentChar + NextChar; // new string

      if ConcatChars exists in the dictionary {
            CurrentChar = ConcatChars;
      } else {
            Output = Take the CurrentChar's index from the dictionary;
            Fill the dictionary with the ConcatChars string;
            CurrentChar = NextChar; // update the current character
      }
}

EncodedMessage = Output;
```

Example: LZW Encoding Algorithm

Use the LZW algorithm to compress the string: BABAABAAAA

First of all, encoder will read the first character of the message, that's "B". Since "B" always found in the dictionary, the encoder simply continues to read the next character ("A") and concatenated them to be string "BA". Since string "BA" doesn't exist in the dictionary, the decoder then saved it into the dictionary and give an index to the output (B's index in the dictionary) as our first output. The complete process can be seen in the below table. So, the final result from the encoding is the codes from the "output index" column: "66, 65, 256, 257, 65, 260, 65". Does this show the good compression? The answer is "yes". Before compressing, the message required 10 x 8 = 80 bits, if the codes are converted into 9 bits, the result will be 7 x 9 = 63 bits. Now, we can save 17 bits.



Current Char	Next Char	CurrentChar + NextChar is in the dictionary ?	Output Index	[New Index] New String
"B"	"A"	No	66	[256]"BA"
"A"	"B"	No	65	[257]"AB"
"B"	"A"	Yes	-	-
"BA"	"A"	No	256	[258]"BAA"
"A"	"B"	Yes	-	-
"AB"	"A"	No	257	[259]"ABA"
"A"	"A"	No	65	[260]"AA"
"A"	"A"	Yes	-	-
"AA"	"A"	No	260	[261]"AAA"
"A"	EOF	-	65	-

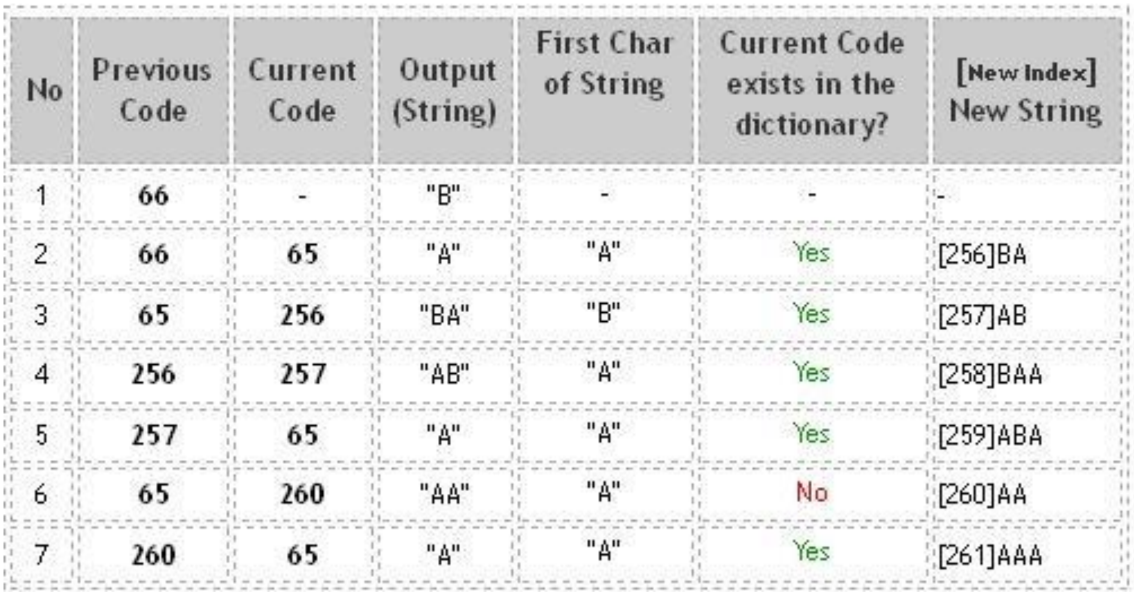
LZW Decoding Algorithm-Pseudo Code

```
Dictionary is initialized with the ASCII characters;
PreviousCode = The first codeword from the encoded message;
CurrentString = Convert the PreviousCode to the string;
CurrentChar = Convert the PreviousCode to the character;

while NOT EOF {
     CurrentCode = The next code from the PreviousCode;

     if the CurrentCode exists in the dictionary {
        CurrentString = Convert the CurrentCode to the string;
     } else {
        CurrentString = Convert the PreviousCode to the string and concatenate it with th
     }

     Output = all the CurrentString;
     CurrentChar = the first character of the CurrentString;
     Insert to dictionary the PreviousCode's string + CurrentChar;
     PreviousCode = CurrentCode;
}

DecodeMessage = Output;
```

Example: LZW Decoding Algorithm

let’s decode “66, 65, 256, 257, 65, 260, 65” to original string.

The decompression result as shown in the table below is the collection of characters/strings from the output column, that’s “BABAABAAAA”. In the first line of the table shows you that the decoder doesn’t add any symbol into the dictionary since in the first step of decoding, the decoder translated the first code directly, that’s ok since the first code from the encoded message will be always found in the dictionary, so we don’t need to check it.



No	Previous Code	Current Code	Output (String)	First Char of String	Current Code exists in the dictionary?	[New Index] New String
1	**66**	-	"B"	-	-	-
2	**66**	**65**	"A"	"A"	Yes	[256]BA
3	**65**	**256**	"BA"	"B"	Yes	[257]AB
4	**256**	**257**	"AB"	"A"	Yes	[258]BAA
5	**257**	**65**	"A"	"A"	Yes	[259]ABA
6	**65**	**260**	"AA"	"A"	No	[260]AA
7	**260**	**65**	"A"	"A"	Yes	[261]AAA