ITCT Lecture 8.3: Lempel-Ziv Coding
— Adaptive Dictionary Compression Algorithm

1. LZ77：Sliding Window Lempel-Ziv Algorithm [gzip, pkzip]

Encode a string by finding the longest match anywhere within a window of past symbols and represents the string by a pointer to location of the match within the window and the length of the match.

J. A. Storer and T. G. Szymanski, "Data Compression Via Textual Substitution," J. ACM, 29(4), pp.928-951, 1982

Assume we have a string $x_1, x_2, \cdots,$ to be compressed from a finite alphabet. A parsing S of a string $x_1 x_2, \cdots, x_n$ is a division of the string into phrases, separated by commas.

Let W be the length of the window.
Then the algorithm can be described as follows：

Assume that we have compressed the string until time $i-1$.
To find the next phrase, find the largest k such that for some j, $i-1-W \leq j \leq i-1$, the string of length k starting at j is equal to the string (of length k) starting at x_i (i.e., $x_{j+l} = x_{i+l}$ for all $0 \leq l < k$).

The next phrase is then of length k (i.e., $x_i, x_{i+1}, \cdots, x_{i+k-1}$) and is represented by the pair (P, L), where P is the location of the beginning of the match and L is the length of the match.

If a match is not found in the window, the next character is sent uncompressed.

To distinguish between these two cases, a flag bit is needed, and hence the phrases are of two types：(F, P, L) of (F, C), where C represents an uncompressed character.

Note that the target of a (pointer, length) paint could extend beyond the window, so that it overlaps with the new phrase.

In theory, this match could be arbitrarily long; in practice, though, the maximum phrase length is restricted to be less than some parameter.

For example, if $W = 4$ and the string is ABBABBABBBAABABA and the initial window is empty, the string will be parsed as follows：

A, B, B, ABBABB, BA, A, BA, BA,

which is represented by the sequence of "pointers"：

(0,A), (0,B), (1,1,1), (1,3,6), (1,4,2), (1,1,1), (1,3,2), (1,2,2),

where the flag bit is 0 if there is no match and 1 if there is a match, and the location of the match is measured backward from the end of the window.

(0,A);(0,B) | (1,1,1)

B ···

Output	4	3	2	1	Input
(1,1,1)			A	B	B A B A ···
(1,3,6)		A	B	B	A B B A B B
(1,3,2) ···	B	A	B	B	B A A B A B A
(1,1,1)	B	B	B	A	A B A B A
(1,3,2)	B	B	A	A	B A B A
(1,2,2)	A	A	B	A	B A

{← longest match →}

2. LZ-78 Tree-structured Lempel-Ziv Algorithms [GIF; compress on Unix]

This algorithm parsed a string into phrases, where each phrase is the shortest phrase not seen so far.

This algorithm can be viewed as building a dictionary in the form of a tree, where the nodes correspond to phrases seen so far.

This algorithm is simple to implement and has become popular as one of the early standard algorithms for file compression on computers because of its speed and efficiency. It is also used for data compression in high-speed modems.

For the string： ABBABBABBBAABABAA $\cdots$,
we parse it as

$$\text{A, B, BA, BB, AB, BBA, ABA, BAA}, \cdots$$

After every comma, we look along the input sequence until we come to the shortest string that has not been marked off before.

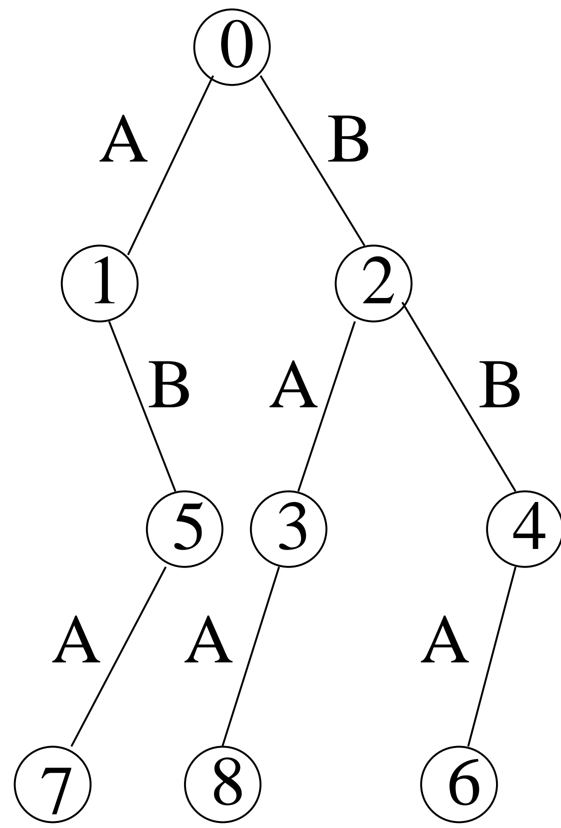
Since this is the shortest string, all its prefixes must have occurred earlier. (Thus, we can build up a tree of these phrases.)

In particular, the string consisting of all but the last bit of this string must have occurred earlier. We code this phrase by giving the location of the prefix and the value of the last symbol.

Thus, the string above would be represented as：

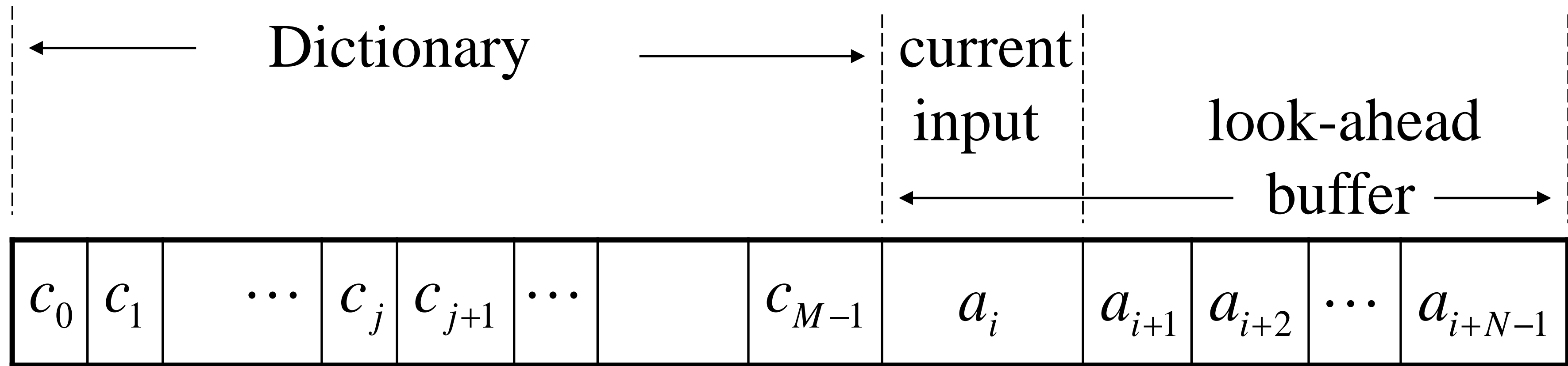
(0,A), (0,B), (2,A), (2,B), (1,B), (4,A), (5,A), (3,A), ⋯

1	A	⟶	0A
2	B	⟶	0B
3	BA	⟶	2A
4	BB	⟶	2B
5	AB	⟶	1B
6	BBA	⟶	4A
7	ABA	⟶	5A
8	BAA	⟶	3A



Sending an uncompressed character in each phrase results in a loss of efficiency. It is possible to get around this by considering the extension character (the last character of the current phrase) as part of the next phrase.

T. A. Welch, A technique for high-performance data compression, computer, 17(1)：pp.8-19, 1984. → LZW-algorithm.



(1) Find the longest match between the strings stored in the dictionary and the string, started at a_i, in the look-ahead buffer.

Assume the starting address of the matched string in the dictionary is J and the longest match length is K (i.e., $A_I = \{a_i, a_{i+1}, \cdots, a_{i+K-1}\}$)

(2) Send (J, K, a_{i+k}) to the decoder

update the dictionary by pushing A_I (the longest matched string) $+ a_{i+k}$ into the Dictionary.

(3) fill the look-ahead buffer, starting at a_{i+k}, from the following input string.

(4) repeat step 1 until the end of the input.