# CS 254: Design And Analysis of Algorithms


Submitted By :-

Naman Jain ( 170001031 )

Mohit Chauhan ( 170001029 )

Under the guidance of

Dr. Kapil Ahuja

# Topic :

# *LZW Compression Algorithm*

# Contents:

- Introduction
- Algorithm Overview
- LZW Compression:
    - Algorithm
    - Example
- LZW Decompression:
    - Algorithm
    - Example
- Analysis
- Implementation
    - LZW_Array
    - LZW_BST
    - LZW_Hash

# Contents:

# Introduction

- **Lempel–Ziv–Welch** (**LZW**) is a universal lossless data compression algorithm created by Abraham Lempel, Jacob Zi, and Terry Welch.

- It is the algorithm of the widely used Unix file compression utility compress and is used in the GIF image format.

- LZW compression is one of the Adaptive Dictionary techniques.
  The dictionary is created while the data are being encoded. So encoding can be done on the fly.

- The computational and space complexity of LZW data compression algorithm is purely depends on the effective implementation of data structure.

- LZW is very effective on the files containing lots of repetitive data especially for text and monochrome images.

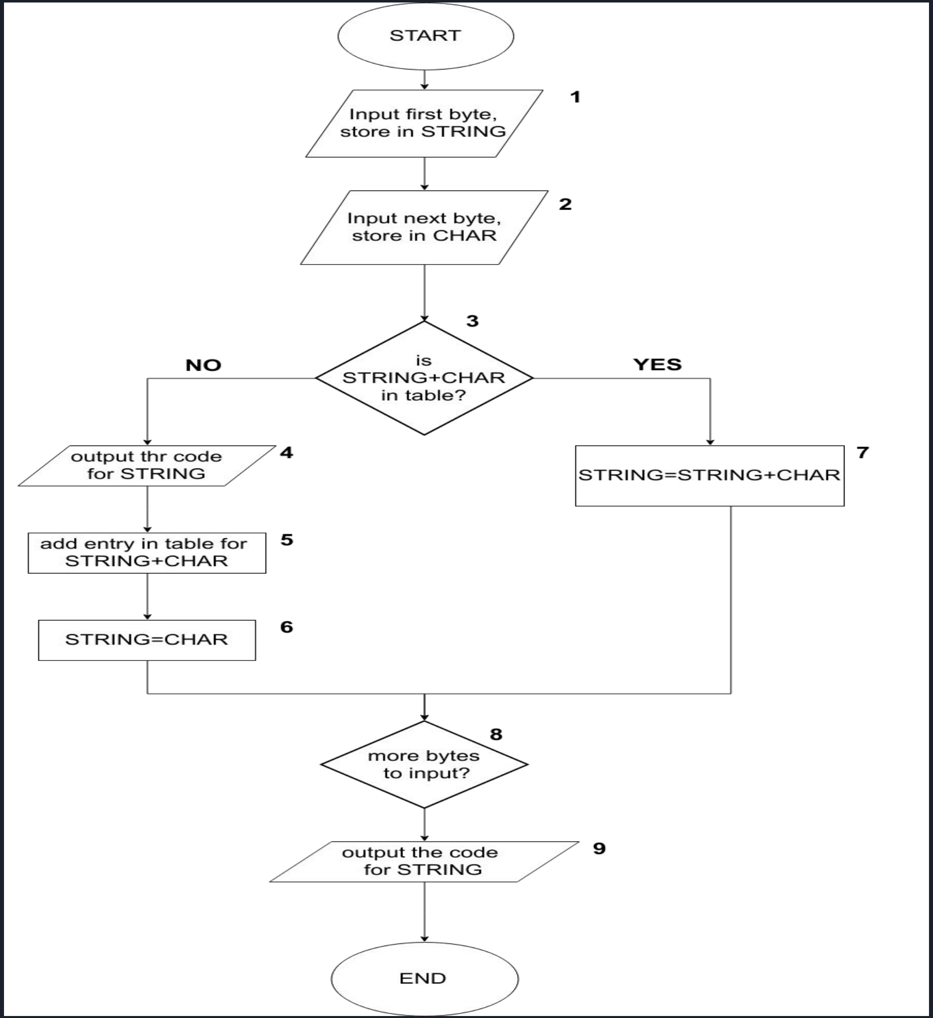
# Algorithm Overview:

- LZW is a ***GREEDY*** algorithm - It tries to find the longest possible string that it has a code for then output that string.
- The LZW algorithm uses dictionary while decoding and encoding.
- LZW compression uses a code table (common choice is to provide 4096 entries in the table). In this case, the LZW encoded data consists of 12 bit codes, each referring to one of the entries in the code table.
- Encoding : During Encoding phase, the algorithm identifies repetitive patterns in text and converts them to integer codes (12-bit usually).
- Decoding : The Algorithm takes integer codes and convert them back to corresponding string with the help of dictionary.

# LZW Compression:

A high level view of the encoding algorithm is shown here:

1. Initialize the dictionary to contain all strings of length one.
2. Find the longest string W in the dictionary that matches the current input.
3. Update dictionary with new code corresponding to string W + next character.
4. Output the code for string W.
5. Continue until end of file is reached.

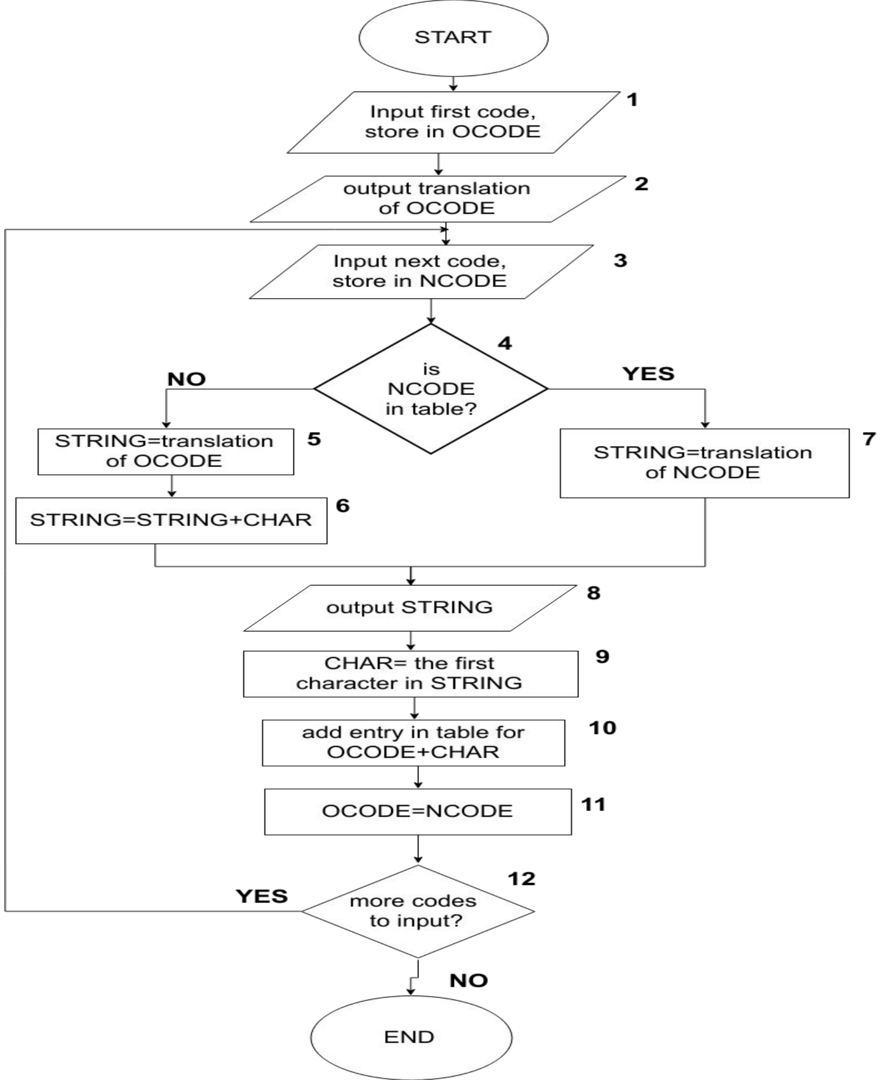

START
1
Input first byte, store in STRING
2
Input next byte, store in CHAR
3
is STRING+CHAR in table?
NO
YES
4
output thr code for STRING
5
add entry in table for STRING+CHAR
6
STRING=CHAR
7
STRING=STRING+CHAR
8
more bytes to input?
9
output the code for STRING
END

| Input String = /WED/WE/WEE/WEB/WET | | | |
|---|---|---|---|
| Character Input | Code Output | New code value | New String |
| /W | / | 256 | /W |
| E | W | 257 | WE |
| D | E | 258 | ED |
| / | D | 259 | D/ |
| WE | 256 | 260 | /WE |
| / | E | 261 | E/ |
| WEE | 260 | 262 | /WEE |
| /W | 261 | 263 | E/W |
| EB | 257 | 264 | WEB |
| / | B | 265 | B/ |
| WET | 260 | 266 | /WET |
| EOF | T | | |

**The Compression Process:**

# LZW Decompression:

1. The decoding algorithm works by reading a value from the encoded input and outputting the corresponding string from the initialized dictionary. To rebuild the dictionary in the same way as it was built during encoding, it also obtains the next value from the input and adds to the dictionary the concatenation of the current string and the first character of the string obtained by decoding the next input value, or the first character of the string just output if the next value can not be decoded.
2. The decoder then proceeds to the next input value and repeats the process until there is no more input, at which point the final input value is decoded without any more additions to the dictionary.
3. In this way, the decoder builds a dictionary that is identical to that used by the encoder, and uses it to decode subsequent input values. Thus, the full dictionary does not need to be sent with the encoded data.


START
Input first code, store in OCODE
1
output translation of OCODE
2
Input next code, store in NCODE
3
is NCODE in table?
4
NO
YES
STRING=translation of OCODE
5
STRING=STRING+CHAR
6
STRING=translation of NCODE
7
output STRING
8
CHAR= the first character in STRING
9
add entry in table for OCODE+CHAR
10
OCODE=NCODE
11
more codes to input?
12
YES
NO
END

| Input Codes: / W E D 256 E 260 261 257 B 260 T | | | | |
|---|---|---|---|---|
| Input/ NEW_CODE | OLD_CODE | STRING/ Output | CHARACTER | New table entry |
| / | / | / | | |
| W | / | W | W | 256 = /W |
| E | W | E | E | 257 = WE |
| D | E | D | D | 258 = ED |
| 256 | D | /W | / | 259 = D/ |
| E | 256 | E | E | 260 = /WE |
| 260 | E | /WE | / | 261 = E/ |
| 261 | 260 | E/ | E | 262 = /WEE |
| 257 | 261 | WE | W | 263 = E/W |
| B | 257 | B | B | 264 = WEB |
| 260 | B | /WE | / | 265 = B/ |
| T | 260 | T | T | 266 = /WET |

# Analysis of LZW :

- The computational and space complexity of LZW data compression algorithm purely depends on the effective implementation of data structure.

- The data structure implementation must give better performance on the following operation:
    - insertion of new pattern to the dictionary,
    - searching of a given pattern
    - returning the matching code word if it is present in the dictionary as a phrase.

We have analysed a number of data structures commonly employed in the LZW dictionary based compression and decompression.

# LZW Implementation:

1) *Array Implementation of Dictionary*

Code File : LZW_ARRAY.cpp

Compression ->

$$= |X| * \left(\frac{1}{N}\left(\frac{1}{1} + \frac{1+2}{2} + \frac{1+2+3}{3} + \cdots + \frac{1+2+3+\cdots+N}{N}\right)\right)$$

$$= |X| * \left(\frac{1}{N}\sum_{i=1}^{N}\left(\frac{i+1}{2}\right)\right)$$

$$= |X| * \frac{1}{2N}\left(\sum_{i=1}^{N} i + 1\right)$$

$$= |X| * \frac{1}{2}\left(\frac{N(N+1)}{2} + 1\right)$$

$$= |X| * \frac{1}{2}\left(\frac{N+1}{2} + 1\right)$$

$$= |X| * \frac{1}{2}\left(\frac{N+3}{2}\right)$$

$$= |X| * \left(\frac{N+3}{4}\right)$$

# LZW Implementation:

1) *Array Implementation of Dictionary*

Code File : LZW_ARRAY.cpp

Decompression ->

$$
\begin{aligned}
&|C| * \left(\frac{1}{n}\sum_{1=1}^{n} 1\right) \\
&= |C| * \left(\frac{1}{n} n\right) \\
&= |C| * \left(\frac{n}{n}\right) \\
&= |C| * (1) \\
&= O(|C|)
\end{aligned}
$$

# 2) *BST Implementation of Dictionary*

Code File : LZW_BST.cpp

$$= \frac{1}{N}\left(O\log(n_1) + O\log(n_2) + \cdots + O\log(n_{N-1}) + O\log(n_N)\right)$$

$$= O\frac{1}{N}\left((\log(n_1) + \log(n_2) + \cdots + \log(n_{N-1}) + \log(n_N))\right)$$

$$= O\left((\log(n_1 * n_2 * \ldots * n_{N-1} * n_N))^{\frac{1}{N}}\right)$$

$$= O\left(\left(\log\prod_{i=1}^{N} n_i\right)^{\frac{1}{N}}\right)$$

$$= O\left(log\left((N!)^{\frac{1}{N}}\right)\right)$$

The number of iteration is base on the length of X

$N = |X|$

Where to BST operation takes place so

$N$

So the comparison required to compress the sequence X is

$$= O\left(X * \left(log\left((N!)^{\frac{1}{N}}\right)\right)\right)$$

# 3) Hash Table Implementation of Dictionary

Code File : LZW_HASH.cpp

$$\frac{(1+\alpha_1)+(1+\alpha_1)+(1+\alpha_3)+\cdots+(1+\alpha_{n-1})+(1+\alpha_n)}{n}$$

$$=\frac{n+(\alpha_1+\alpha_1+\alpha_1+\cdots+\alpha_{n-1}+\alpha_n)}{n}$$

$$=\frac{n+\sum_{i=1}^{n}\alpha_i}{n}$$

$$=O\left(\frac{\sum_{i=1}^{n}\alpha_i}{n}\right)$$

$$=1+\frac{1}{n}\sum_{i=1}^{n}\alpha_i$$

$$=1+AM\{\alpha_i\}$$

Where $i = 1,2,\ldots,n$

The length of the X is n =|X| then

$O\ |X| * (1 + AM\{\alpha_i\})$

# Compression Ratio:

| File/Size | Input Size(bytes) | Compressed Size(bytes) | Compression Ratio |
|---|---|---|---|
| alice.txt | 148.5 | 68.4 | 2.17 |
| asyoulik.txt | 122.2 | 61.3 | 1.99 |
| lcet.txt | 416.8 | 180.6 | 2.3 |
| plrabn.txt | 470.6 | 200.5 | 2.34 |

NOTE: Compression Ratio does not change with data structure used.

# Compression Time:

| Type/File | alice.txt | asyoulik.txt | lcet.txt | plrabn.txt |
|---|---|---|---|---|
| Linear Array | 13.4612 seconds | 11.077 seconds | 59.9853 seconds | 71.4244 seconds |
| BST | 0.168326 seconds | 0.142711 seconds | 0.395633 seconds | 0.442393 seconds |
| HASH | 0.081763 seconds | 0.075617 seconds | 0.168414 seconds | 0.170973 seconds |

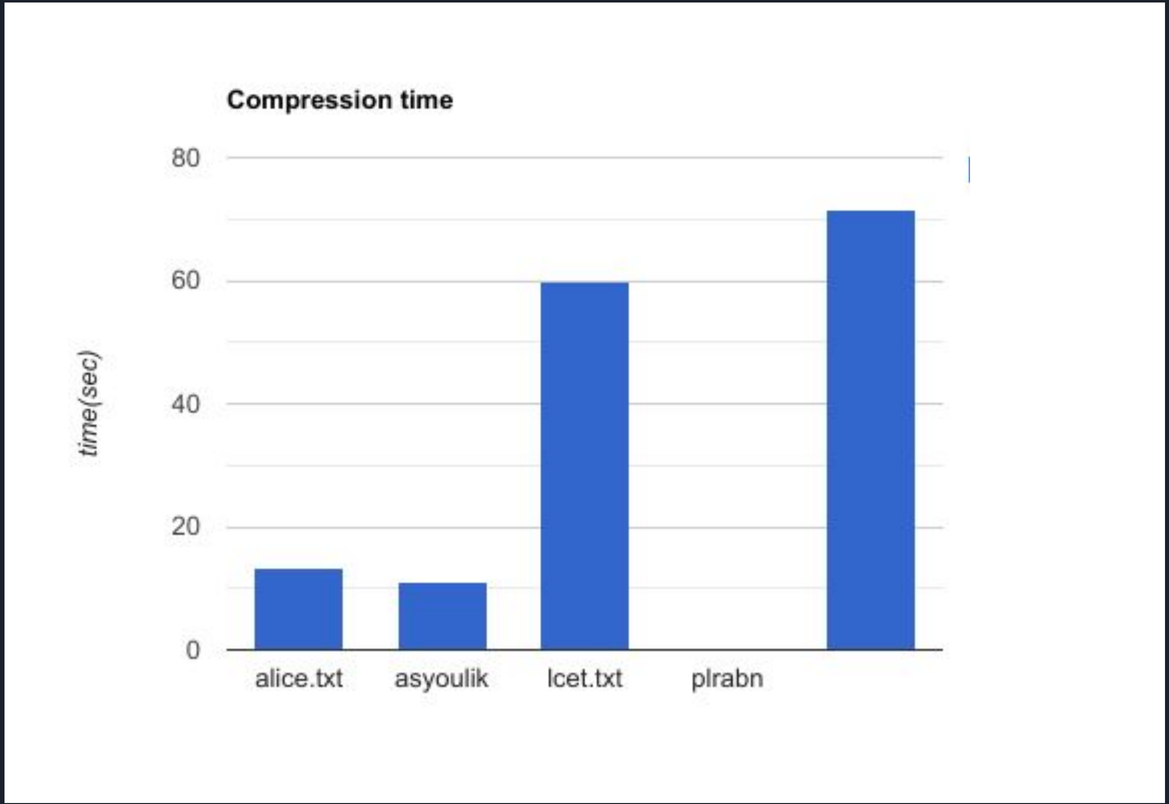
Compression

# Decompression Time:

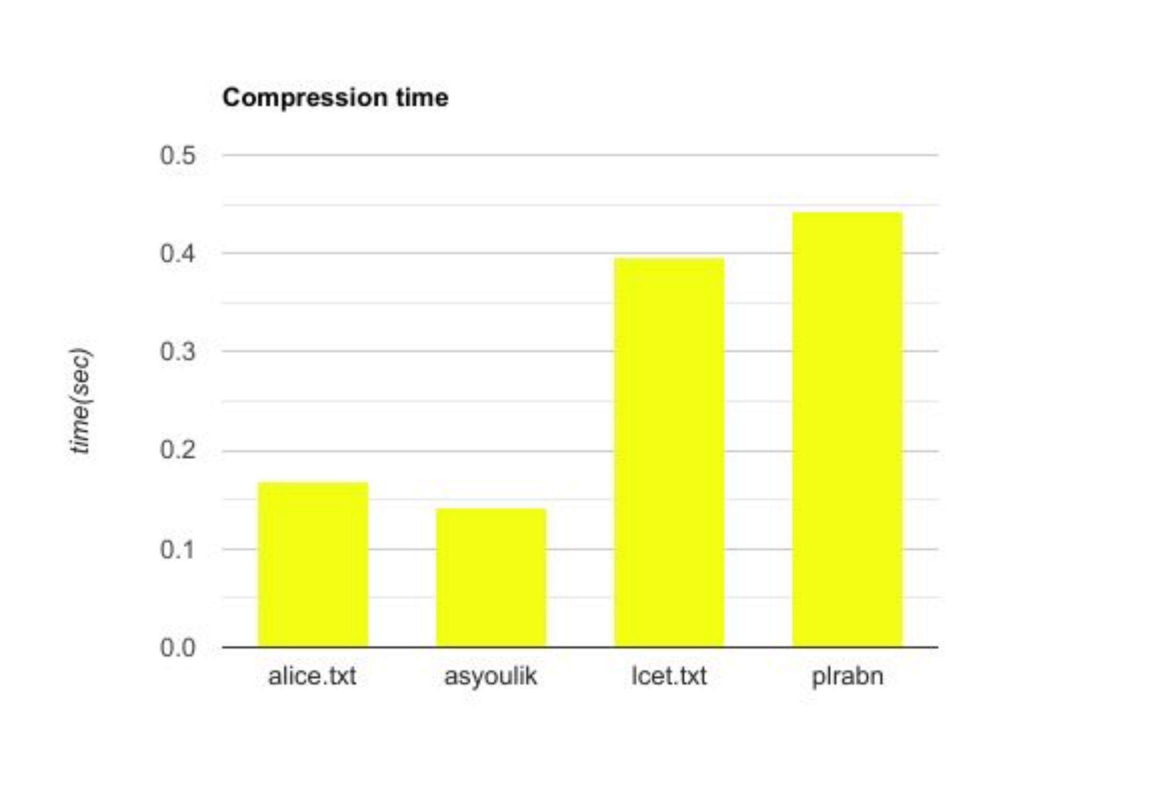
| Type/File | alice.txt | asyoulik.txt | lcet.txt | plrabn.txt |
|---|---|---|---|---|
| Linear Array | 0.003477 seconds | 0.00292 seconds | 0.00855 seconds | 0.015495 seconds |
| BST | 0.076178 seconds | 0.068592 seconds | 0.177581 seconds | 0.20527 seconds |
| HASH | 0.02537 seconds | 0.022566 seconds | 0.061815 seconds | 0.066865 seconds |

Decompression

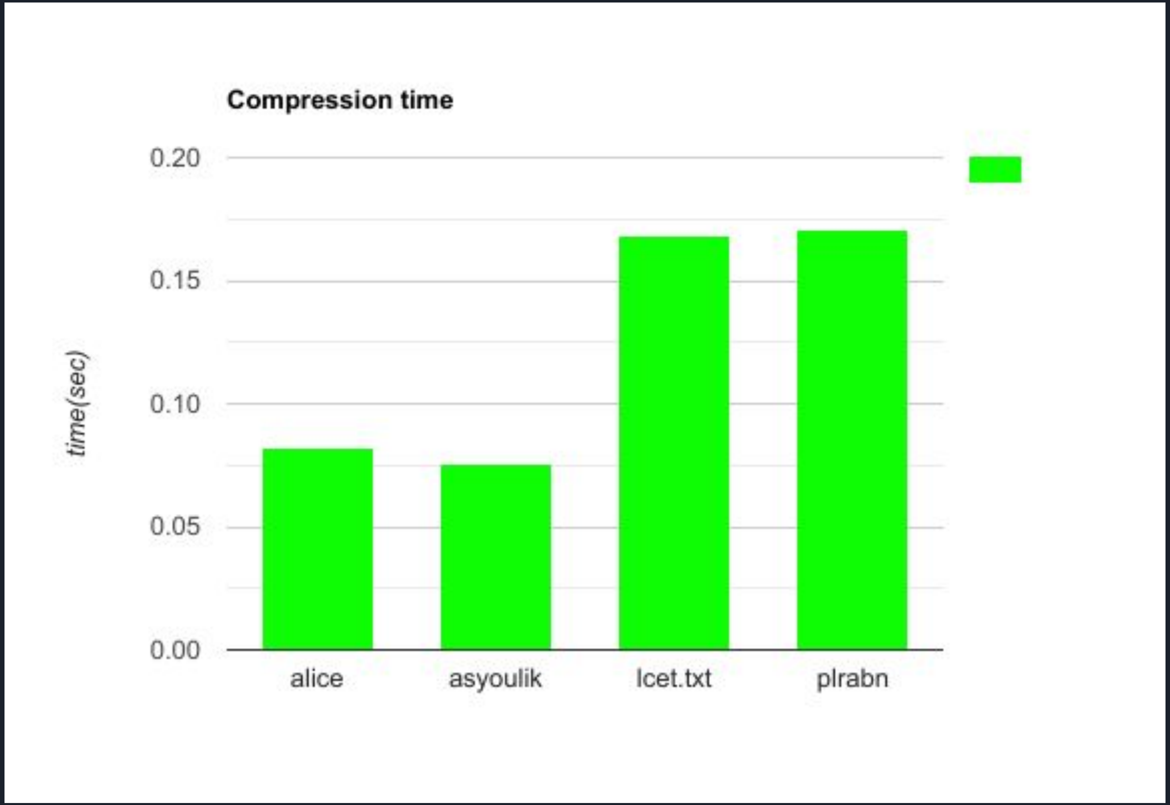
# Time Comparison for  Array Implementation:



# Comparison for BST:



# Comparison for Hash:



# Improving  LZW Algorithm :

*Adaptive Loading of dictionary :*

- Standard LZW Algorithm inserts only one string  s = prev + ch, when string prev + ch is the first string not present in dictionary during iteration.

- In Adaptive Loading , if w1 and w2 are two consecutive strings inserted to dictionary then encoder also inserts strings w1 + Ki , where Ki denotes i-th prefix of w2 , for i varying from 1 to maxlen. Larger the value of maxlen larger is the number of strings added to dictionary during each unsuccessful search.