# Project Based Learning Report

on

Generation of variable length source Lempel Ziv Coding and Decoding

Submitted in the partial fulfillment of the requirements

For the Project based learning in Information Theory & Coding

In branch

Electronics & Communication

By

## PRN-2114110433 ANSHUMAN

## PRN-2114110436 SAYYAM BAKSHI

**PRN-2114110443 ANAGHA DIXIT**

Under the guidance of Course In charge S.K.PAWAR

Department of Electronic & Communication Engineering

Bharati Vidyapeeth, (Deemed to be University)

College of Engineering, Pune-411043

Academic Year: - 2023-24

## Bharati Vidyapeeth (Deemed to be University) College of Engineering,

**Pune – 411043**

**DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING**

**CERTIFICATE**

## Certified that the Project Based Learning report entitled, “Generation of variable length source Lempel Ziv Coding and Decoding” is a bonafide work done by PRN-2114110433 ANSHUMAN

## PRN-2114110436 SAYYAM BAKSHI

## PRN-2114110443 ANAGHA DIXIT

in partial fulfillment of the requirement for the award of credits for Project Based Learning (PBL) in **“**Information Theory & Coding**”,** of Bachelor of Technology semester V, in Electronics and Communication Engineering.

Date:

S.K.PAWAR Dr. Arundhati Shinde

Course In-charge Head of Department – ECE

# INDEX

|  |  |  |
| --- | --- | --- |
| **S.NO.** | **CONTENT** | **PAGE NO.** |
| 1 | PROBLEM STATEMENT | 1 |
| 2 | INTRODUCTION | 2 |
| 3 | THEORY | 3 |
| 4 | FLOWCHART | 11 |
| 5 | ALGORITHM | 12 |
| 6 | SOURCE CODE | 13 |
| 7 | CONCLUSION | 14 |
| 8 | COURSE OUTCOME | 15 |
| 9 | REFERENCES | 16 |

**1. PROBLEM STATEMENT**

Design and implement a variable-length source coding and decoding system based on the Lempel-Ziv algorithm, capable of efficiently compressing and decompressing a wide range of input data types with variable-length source sequences. This system should address the challenge of adapting to variable-length source data, optimizing compression ratios, and ensuring reliable decoding while maintaining reasonable computational complexity.

**2. INTRODUCTION**

**Why do we need a Compression Algorithm?**

There are two categories of compression techniques, lossy and lossless. Whilst each uses different techniques to compress files, both have the same aim: To look for duplicate data in the graphic (GIF for LZW) and use a much more compact data representation. Lossless compression reduces bits by identifying and eliminating statistical redundancy. No information is lost in lossless compression. On the other hand, Lossy compression reduces bits by removing unnecessary or less important information. So we need Data Compression mainly because:

* Uncompressed data can take up a lot of space, which is not good for limited hard drive space and internet download speeds.
* While hardware gets better and cheaper, algorithms to reduce data size also help technology evolves.
* Example: One minute of uncompressed HD video can be over 1 GB. How can we fit a two-hour film on a 25 GB Blu-ray disc?

Lossy compression methods include DCT (Discrete Cosine Transform), Vector Quantisation, and Transform Coding while Lossless compression methods include RLE (Run Length Encoding), string-table compression, LZW (Lempel Ziff Welch), and zlib. There Exist several compression Algorithms, but we are concentrating on LZW.

**3. THEORY**

**What is Lempel–Ziv–Welch (LZW) Algorithm ?**

The LZW algorithm is a very common compression technique. This algorithm is typically used in GIF and optionally in PDF and TIFF. Unix’s ‘compress’ command, among other uses. It is lossless, meaning no data is lost when compressing. The algorithm is simple to implement and has the potential for very high throughput in hardware implementations. It is the algorithm of the widely used Unix file compression utility compress and is used in the GIF image format.  
The Idea relies on reoccurring patterns to save data space. LZW is the foremost technique for general-purpose data compression due to its simplicity and versatility. It is the basis of many PC utilities that claim to “double the capacity of your hard drive”.

**How does it work?**

LZW compression works by reading a sequence of symbols, grouping the symbols into strings, and converting the strings into codes. Because the codes take up less space than the strings they replace, we get compression. Characteristic features of LZW includes,

* LZW compression uses a code table, with 4096 as a common choice for the number of table entries. Codes 0-255 in the code table are always assigned to represent single bytes from the input file.
* When encoding begins the code table contains only the first 256 entries, with the remainder of the table being blanks. Compression is achieved by using codes 256 through 4095 to represent sequences of bytes.
* As the encoding continues, LZW identifies repeated sequences in the data and adds them to the code table.
* Decoding is achieved by taking each code from the compressed file and translating it through the code table to find what character or characters it represents.

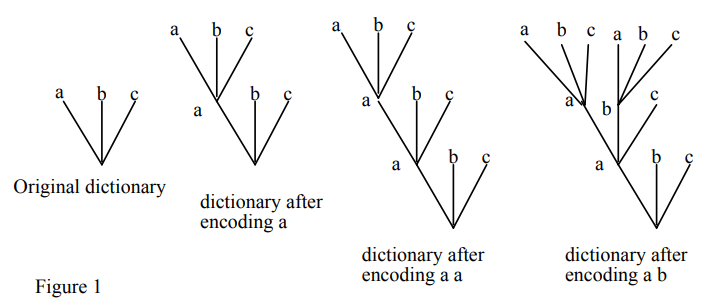
Example: ASCII code. Typically, every character is stored with 8 binary bits, allowing up to 256 unique symbols for the data. This algorithm tries to extend the library to 9 to 12 bits per character. The new unique symbols are made up of combinations of symbols that occurred previously in the string. It does not always compress well, especially with short, diverse strings. But is good for compressing redundant data, and does not have to save the new dictionary with the data: this method can both compress and uncompress data.   
There are excellent article’s written up already, you can look more in-depth [here](https://www.cs.cmu.edu/~cil/lzw.and.gif.txt), and also Mark Nelson’s [article](http://marknelson.us/2011/11/08/lzw-revisited/) is commendable.

**Implementation**

The idea of the compression algorithm is the following: as the input data is being processed, a dictionary keeps a correspondence between the longest encountered words and a list of code values. The words are replaced by their corresponding codes and so the input file is compressed. Therefore, the efficiency of the algorithm increases as the number of long, repetitive words in the input data increases.

**Examples:**

**EXAMPLE 1:** Consider the sequence a a a a b ... from the alphabet {a,b,c}. Fig. 1 illustrates how this is parsed into a | a a | a b | ... and how the dictionary tree grows as this parsing takes place. Each time the encoder parses another segment, it generates a binary code word for that segment. We postpone the question of how the mapping from dictionary entries to code words takes place until later.



**EXAMPLE 2**: Consider an unending sequence of 0's from the binary alphabet {0, 1}. This gets parsed into 0 | 0 0 | 0 0 0 | 0 0 0 0 | 0 0 0 0 0 | ... . The corresponding code tree is shown in Figure 2. Note that the number of source letters involved in the first c parsed strings is 1+2+3+...+c, or c(c+1)/2. Thus as the length n of the source sequence increases, the number of parsed strings grows roughly as 2n (see Figure 2). It is not difficult to imagine that this example yields the fastest possible increase in the size of the individual parsed strings with n and the slowest possible growth in the total number of parsed strings with.

A line with numbers and lines

Description automatically generated

**LZW Encoding**

LZW stands for “Lempel-Ziv-Welch”. The LZW algorithm is a very common compression technique. This algorithm is typically used in GIF and PDF. It is lossless, meaning no data is lost when compressing. The algorithm is simple to implement and has the potential for very high throughput in hardware implementations. It is the algorithm of the widely used Unix file compression utility compress, and is used in the GIF image format.

This algorithm basically utilizes reoccurring patterns to save data space. As our “Challange.txt” file had numerous repeating characters. So, this algorithm will be most efficient for that file. LZW compression works by reading a sequence of symbols, grouping the symbols into strings, and converting the strings into codes.

LZW compression uses a code table, with 4096 as a common choice for the number of table entries. Codes 0-255 in the code table are always assigned to represent single bytes from the input file. When encoding begins the code table contains only the first 256 entries, with the remainder of the table being blanks. Compression is achieved by using codes 256 through 4095 to represent sequences of bytes. As the encoding continues, LZW identifies repeated sequences in the data, and adds them to the code table. Decoding is achieved by taking each code from the compressed file and translating it through the code table to find what character or characters it represents. Following picture is the explanation of the encoding process that we borrowed from https://www2.cs.duke.edu/csed/curious/compression/lzw.html to understand the algorithm.

A screenshot of a computer

Description automatically generated

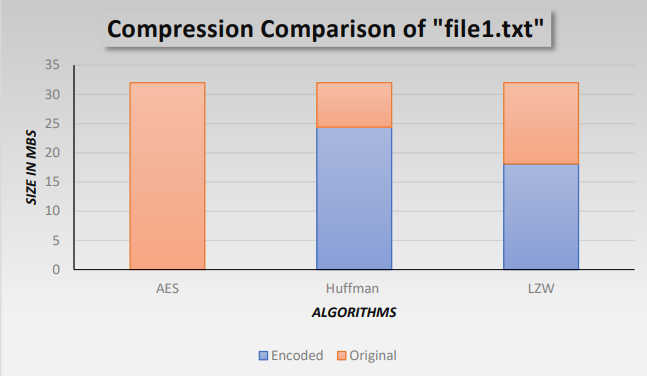
**LZW Encoding Algorithm Benchmarks:**

**A graph of a bar chart

Description automatically generated with medium confidence**

As can be seen from the above graph, LZW Compression Algorithm showed remarkable performance in terms of space complexity and time both. LZW performed better on “Challange.txt” as the file contained multiple reoccurring characters which are handled more efficiently by LZW as compared to Huffman.

**Comparison in terms of Space:**



A graph of different colored squares

Description automatically generated

From the above-mentioned graphs, it can be easily deduced that AES algorithm has the worst performance in terms of space. Comparing the remaining two, we can see LZW winning with a major difference. This means that in terms of space, LZW Data Compression is the most efficient of all three.

**Comparison in terms of Time:**

**A graph of a graph with a line and numbers

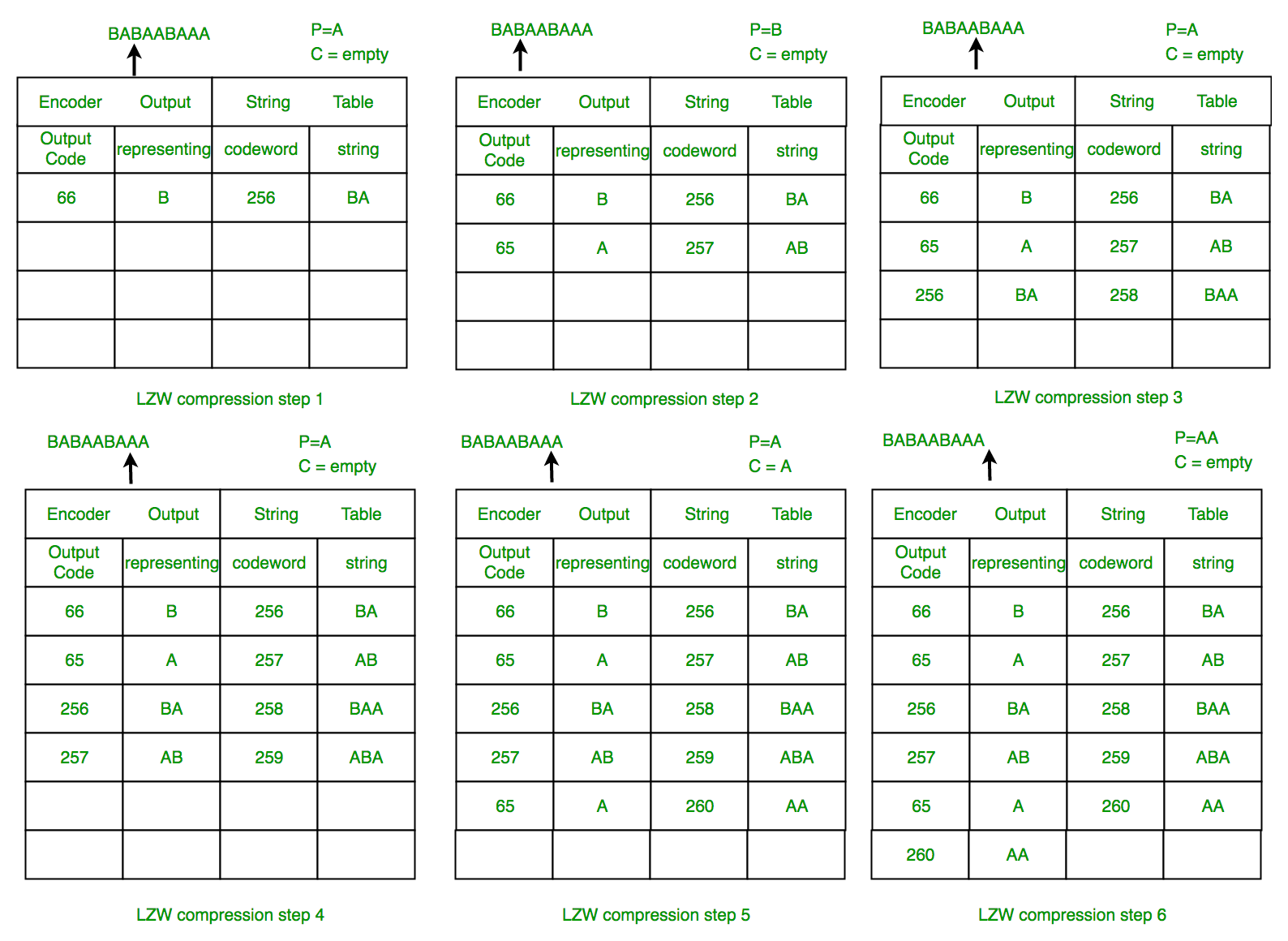
Description automatically generated with medium confidence**

From the above-mentioned graph, it can be easily deduced that AES algorithm has the worst performance in terms of space. Comparing the remaining two, we can see Huffman winning with a major difference. This means that in terms of time, Huffman Data Encryption Algorithm is the most efficient of all three.

After analyzing all of the above result, we can see that Huffman data compression is time efficient whereas LZW data compression is space efficient. But as the project was about data compression, so LZW Data Compression Algorithm will be considered as the overall best algorithm for our data compression.

**Compression using LZW**

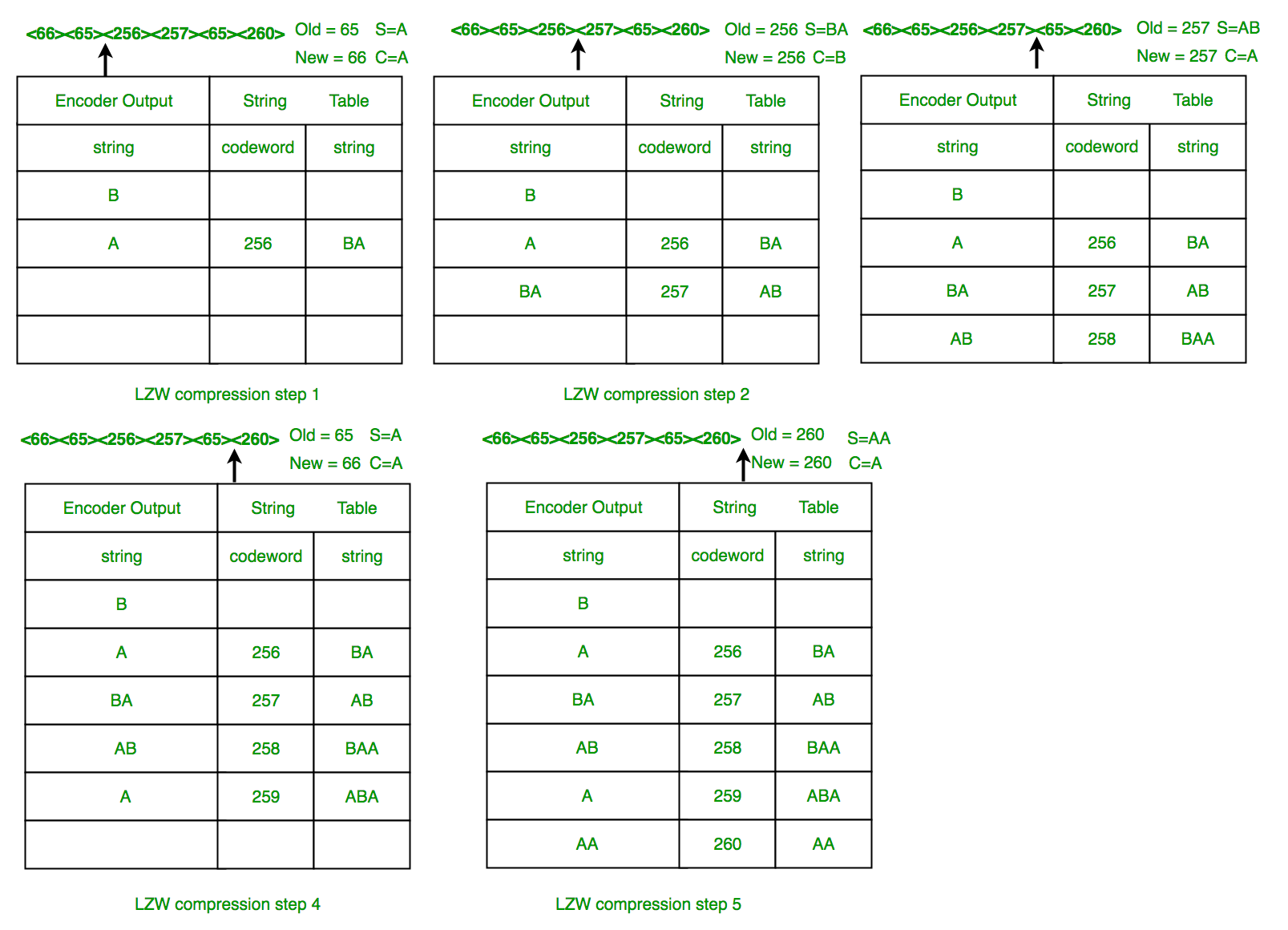
**Example 1**: Use the LZW algorithm to compress the string: **BABAABAAA**   
The steps involved are systematically shown in the diagram below.



**LZW Decompression**

The LZW decompressor creates the same string table during decompression. It starts with the first 256 table entries initialized to single characters. The string table is updated for each character in the input stream, except the first one. Decoding is achieved by reading codes and translating them through the code table being built.

**Example 2**: LZW Decompression: Use LZW to decompress the output sequence of : **<66><65><256><257><65><260>**   
The steps involved are systematically shown in the diagram below.



In this example, 72 bits are represented with 72 bits of data. After a reasonable string table is built, compression improves dramatically.   
**LZW Summary:**This algorithm compresses repetitive sequences of data very well. Since the codewords are 12 bits, any single encoded character will expand the data size rather than reduce it.

**Advantages of LZW over**[**Huffman**](https://www.geeksforgeeks.org/greedy-algorithms-set-3-huffman-coding/)**:**

* LZW requires no prior information about the input data stream.
* LZW can compress the input stream in one single pass.
* Another advantage of LZW is its simplicity, allowing fast execution.
* **High Compression Ratio:** LZW can achieve high compression ratios, particularly for text-based data, which can significantly reduce file sizes and storage requirements.
* **Fast Decompression:** LZW decompression is typically faster than other compression algorithms, making it a good choice for applications where decompression speed is critical.
* **Universal Adoption:** LZW is widely used and supported across a variety of software applications and operating systems, making it a popular choice for compression and decompression.
* **Dynamic Compression:** LZW uses a dynamic compression algorithm, meaning it adapts to the data being compressed, which allows it to achieve high compression ratios even for data with repetitive patterns.

**Disadvantages:**

* **Patent Issues**: LZW compression was patented in the 1980s, and for many years its use was subject to licensing fees, which limited its adoption in some applications.
* **Memory Requirements:** LZW compression requires significant memory to maintain the compression dictionary, which can be a problem for applications with limited memory resources.
* **Compression Speed:** LZW compression can be slower than some other compression algorithms, particularly for large files, due to the need to constantly update the dictionary.
* **Limited Applicability:**LZW compression is particularly effective for text-based data, but may not be as effective for other types of data, such as images or video, which have different compression requirements.

**4. FLOWCHART**

A diagram of a message

Description automatically generated

**5. ALGORITHM**

**LZW Encoding Algorithm**

1. Initialize a dictionary called `dictionary` with entries for all 256 possible one-character sequences (ASCII characters) and their corresponding integer values.
2. Initialize an empty string `p` to hold the current sequence being processed.
3. Initialize an empty list `result` to store the encoded message.
4. Loop through each character `c` in the input `message`:

a. Concatenate `p` and `c` to form the sequence `pc`.

b. Check if `pc` is in the `dictionary`. If it is, update `p` to `pc`.

c. If `pc` is not in the `dictionary`, append the integer value of `p` to the `result` list and add `pc` to the `dictionary` with a new integer value.

d. Update `p` to `c`.

1. After processing all characters in the `message`, check if there is any remaining content in `p`. If so, append the integer value of `p` to the `result` list.
2. Return the `result` list, which contains the LZW-encoded message.

**LZW Decoding Algorithm**

1. Initialize a dictionary called `dictionary` with entries for all 256 possible one-character sequences (ASCII characters) and their corresponding characters.
2. Initialize an empty string `result` with the first character of the encoded message.
3. Initialize a string `s` with the same value as `result`.
4. Loop through each encoded integer `k` in the encoded message (starting from the second element since the first element is already in `result`):

a. If `k` is in the `dictionary`, retrieve the corresponding character `entry`.

b. If `k` is equal to the length of the `dictionary`, set `entry` to the concatenation of `s` and the first character of `s`.

c. If `k` is neither in the `dictionary` nor equal to the length of the `dictionary`, raise a `ValueError` indicating a bad compressed value.

1. Append `entry` to the `result` string.
2. Add a new entry to the `dictionary` with the next available integer key and the value as the concatenation of `s` and the first character of `entry`.
3. Update `s` to be equal to `entry`.
4. Continue the loop until all encoded integers have been processed.
5. Return the `result` string, which contains the LZW-decoded message.

This algorithm enables the compression and decompression of messages using the LZW algorithm, effectively encoding variable-length sequences into a series of integers and decoding them back to their original form.

**6. SOURCE CODE**

def lzw\_encode(message):

dictionary = {chr(i): i for i in range(256)}

p = ""

result = []

for c in message:

pc = p + c

if pc in dictionary:

p = pc

else:

result.append(dictionary[p])

dictionary[pc] = len(dictionary)

p = c

if p:

result.append(dictionary[p])

return result

def lzw\_decode(encoded\_message):

dictionary = {i: chr(i) for i in range(256)}

result = chr(encoded\_message[0])

s = result

for k in encoded\_message[1:]:

if k in dictionary:

entry = dictionary[k]

elif k == len(dictionary):

entry = s + s[0]

else:

raise ValueError("Bad compressed k: %s" % k)

result += entry

dictionary[len(dictionary)] = s + entry[0]

s = entry

return result

# Get input from user

message = input("Enter a message to encode: ")

# Encode and decode the message

encoded\_message = lzw\_encode(message)

print("Encoded:", encoded\_message)

decoded\_message = lzw\_decode(encoded\_message)

print("Decoded:", decoded\_message)

**7.CONCLUSION**

In conclusion, the "Generation of Variable Length Source Lempel Ziv Coding and Decoding" project has provided valuable insights into data compression and encoding techniques. Through the successful implementation of the Lempel-Ziv-Welch (LZW) algorithm, this project has demonstrated the effectiveness of LZW in compressing and decompressing variable-length source data while maintaining data integrity. It has been an informative and practical exploration of data compression algorithms.

**8.COURSE OUTCOME**

**C02 - To study various source encoding algorithms.**

Thus, C02 is satisfied

Lempel-Ziv-Welch (LZW), is a widely used algorithm for data compression. To satisfy CO2, one would study various source encoding algorithms like LZW, which involve techniques for efficient data compression and decompression, contributing to the field of data coding and compression research. By examining such algorithms, researchers aim to optimize data storage and transmission, which can have implications for various applications, including data transfer and storage efficiency.

**9. REFERENCES**

* [mit.edu](http://web.mit.edu/6.02/www/s2012/handouts/3.pdf)
* [Dave.Marshall](https://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/node214.html)
* [duke.edu](https://www.cs.duke.edu/csed/curious/compression/lzw.html)
* [michael.dipperstein](http://michael.dipperstein.com/lzw/)
* [LZW(Youtube)](https://www.youtube.com/watch?v=j2HSd3HCpDs)
* [faculty.kfupm.edu.sa](http://faculty.kfupm.edu.sa/ics/garout/Teaching/ICS202/)
* [github repository(kmeelu)](https://github.com/kmeelu/LZW)