HCMUS
Viet Nam National University
Ho Chi Minh City
University of Science

fit@hcmus

Lempel–Ziv–Welch (LZW) Compression

CSC10004: Data Structures and Algorithms

Supervisor:
Nguyen Thanh Tinh

Group 7

Full Name	Student ID
Thai Gia Huy	23120008
Ho Thuy Tram	23120421

Ho Chi Minh City, 8th December 2024

Contents

1 Preface

This report is prepared for the Data Structures and Algorithms project by group 7 under the supervision of Mr. Nguyen Thanh Tinh. The work has been divided among team members, with their individual contributions outlined in the following table 1.1:

Table 1.1: Work assignment and contribution evaluation

Team Member	Assigned Tasks		Completion Rate (%)
Thai Gia Huy	Report	1	100
		2.2	100
		3.3	100
		4.2	100
		5.3	100
		6	100
		8	100
	Slide	3	100
		4	100
		5	100
Ho Thuy Tram	Report	2.1	100
		3.1	100
		3.2	100
		4.1	100
		5.1	100
		5.2	100
		7	100
	Slide	1	100
		2	100

The report provides a comprehensive exploration of the LZW (Lempel-Ziv-Welch) compression algorithm, one of the most influential and widely used data compression techniques. The primary purpose of this report is to examine the theoretical foundations, practical applications, and performance characteristics of LZW, making it accessible to readers with an interest in data compression technologies.

The report begins with an introduction to the historical development of LZW and its significance in various real-world scenarios. It then delves into the algorithm's mechanics, outlining both the compression and decompression processes with illustrative examples to enhance understanding. Following this, the report analyzes the algorithm's performance, focusing on efficiency and how compression ratios influence time complexity. To place LZW in a broader context, the report provides an overview of related algorithms, such as LZ77 and LZ78, emphasizing their unique features and how they compare with LZW.

The report aims to offer a balanced view, covering both theoretical insights and practical implications. While every effort has been made to ensure the content is comprehensive and accurate, certain simplifications were adopted to maintain clarity and accessibility. Feedback and constructive suggestions for improvement are welcome as they will help refine this work further.

2 Introduction

2.1 Historical Background of LZW Compression

Lempel-Ziv-Welch (LZW) is a widely known lossless, dictionary-based compression algorithm that builds upon the pioneering work of Abraham Lempel and Jacob Ziv on LZ78, published in 1978 in *IEEE Transactions on Information Theory*[1]. Terry Welch further refined the algorithm to improve its efficiency and adaptiveness, leading to his 1984 article, "A Technique for High-Performance Data Compression"[2], published in *Computer*.

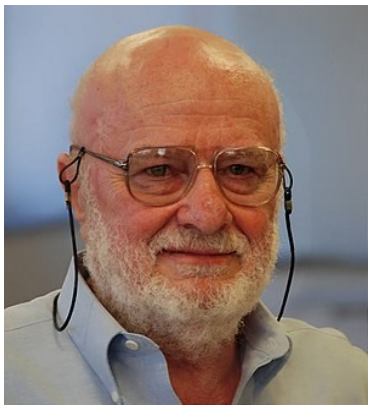

Figure 2.1: Abraham Lempel

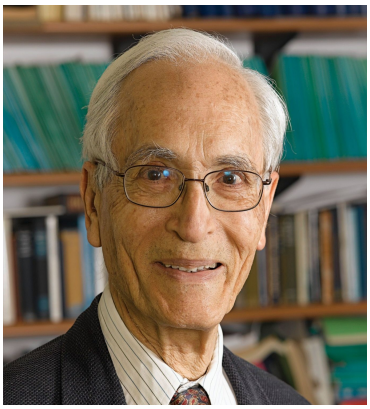

Figure 2.2: Jacob Ziv

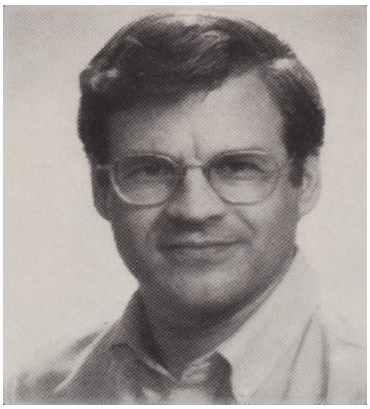

Figure 2.3: Terry Welch

The development of LZW arose from the need for efficient data compression methods during a time of rapidly growing digital information. Its design aimed to address the limitations of earlier compression methods by introducing a dynamic dictionary-based approach that did not require pre-analysis of the data. This adaptability marked a significant evolution in lossless compression techniques.

Unlike its predecessor LZ78, LZW uses a simplified mechanism to build and maintain its dictionary. This innovation made the algorithm computationally efficient and highly adaptable to various input data types, while preserving the core properties of the Lempel-Ziv methods. LZW's simplicity in logic allowed it to achieve high compression ratios without the complexity of earlier methods, making it a cornerstone in the history of data compression.

2.2 Applications of LZW Compression

The LZW compression algorithm is a cornerstone of modern data compression, applied extensively due to its efficiency and reliability. One of its most recognizable applications is in the *GIF (Graphics Interchange Format)*. GIFs utilize LZW compression to minimize file sizes while preserving image quality, enabling quick load times and low bandwidth usage, which are crucial for web graphics and animations.

In the domain of *text compression*, LZW is highly effective for files with repetitive patterns, such as structured data in XML or JSON formats. By replacing repeating strings with shorter codes, the algorithm significantly reduces file sizes while retaining the ability to reconstruct the original content without any loss. This makes LZW indispensable in archiving and transferring text-based data.

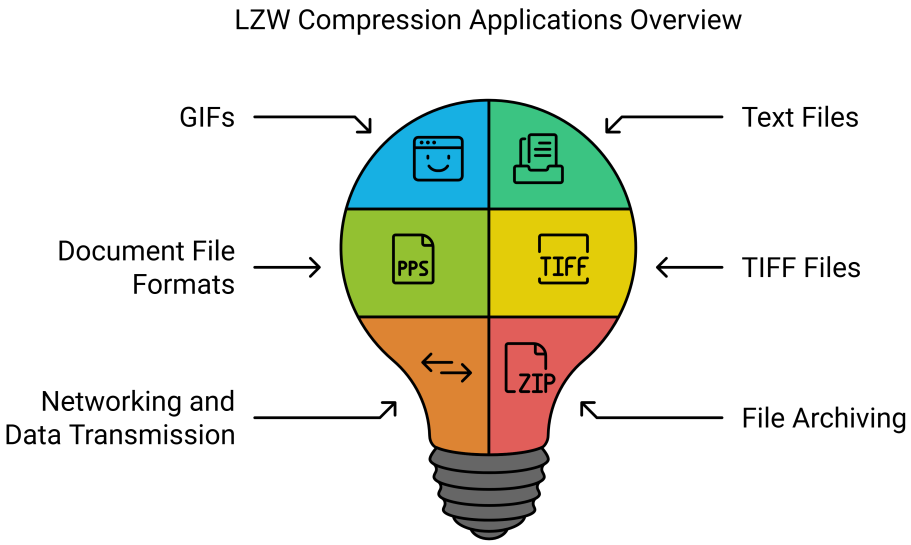

Figure 2.4: Applications of LZW Compression

LZW also plays a pivotal role in *document file formats* like PostScript and PDF. These formats benefit from LZW's ability to optimize file sizes without compromising the integrity of the content, ensuring rapid rendering and efficient storage. This application is particularly valuable in professional publishing and document sharing.

In addition to GIFs, LZW is used in certain *TIFF (Tagged Image File Format)* files. This is particularly relevant in medical imaging and archival photography, where maintaining high precision is essential. The lossless compression provided by LZW ensures that the quality and integrity of these critical files remain intact.

LZW's impact extends to *networking and data transmission*. By compressing data before sending it, the algorithm reduces bandwidth requirements and accelerates transfer speeds, making it integral to systems that rely on high-speed communication and efficient resource usage.

Finally, the influence of LZW can be seen in *file archiving and software packaging tools*. Formats like ZIP have drawn inspiration from LZW's principles to enable efficient storage and transfer of large datasets and software installations. This reliability and effectiveness in handling large volumes of data have made LZW a foundational component in these technologies.

These examples demonstrate LZW's versatility and its ability to address diverse needs, from compressing multimedia files to enhancing network efficiency. The algorithm's continued relevance is a testament to its robustness and adaptability in modern computing.

3 Understanding the LZW Compression Algorithm

3.1 Key Concepts and Mechanisms

Lempel-Ziv Welch coding makes use of dictionaries to store substrings of characters that have occured before in the text, thus "memorizing" them. When a repeating substring is recognized, it uses the indices of the place in the dictionary where the required substring is stored and compresses the text doing so. This method thus relies heavily on repetition.

A further property is that it makes use of the "greedy" parsing algorithm, where the text is looped over exactly once. During this parsing, the longest recognized substring is saved to the result and the combination of the current substring and the next occurring character is added to the dictionary.

Since the dynamic dictionary is created by both compressor and decompressor with small dictionary of all possible single-character strings like ASCII characters, there is no dictionary transmission which reduces transmission overhead.

Although this algorithm can achieve great compression, it does not attempt to optimally select strings by making use of probability estimation. Therefore, its effectiveness is less than optimal, but creates great usability by the simplicity of the algorithm.

3.2 Compression Process with Examples

Algorithm 1 LZW Compression Process

```
1: Initialize an empty dictionary.
2: Populate the emptied dictionary with all possible one-length characters from the
   extended ASCII set (values 0 to 255).
3: Initialize an empty string P.
4: while not at the end of the character stream do
5:     C = next character.
6:     if P + C exists in the dictionary then
7:         P = P + C (Extend by adding C to the string P).
8:     else
9:         Output the code for P.
10:        Append the string P + C to the dictionary.
11:        P = C (Replace the string P with the current character).
12:    end if
13: end while
```

The compression phase of the LZW algorithm involves scanning through the input data, identifying repeating sequences, and replacing them with shorter codes that refer to these sequences. Initially, the dictionary contains all possible individual symbols that appear in the input data. As the algorithm processes the input, it builds the dictionary dynamically by adding longer sequences of characters encountered during the scan.

The process starts by taking an input string and checking if it already exists in the dictionary. If the sequence is found, it is replaced with the corresponding code. If the

sequence is not found, it is added to the dictionary with a new code, and the algorithm continues with the next symbol. This approach works efficiently when dealing with data that contains a lot of repetitive sequences, as a single code can replace each repeating pattern.

Example: LZW compression process for the string "**ABCBCCAB**".

Initialize: Dictionary with all possible single-length characters from the extended ASCII table. Each character serves as the key, and the corresponding value is initialized to 0, for instance: dictionary["0"] = 176, dictionary["A"] = 193.

Table 3.1: Trace of the LZW Algorithm step by step: Compression process

Step	Encoder Output		Dictionary		Explaination
	Output	Represents	Key	Value	
1					P = "", C = "A", assign P+C = "A" to P
2	193	"A"	"AB"	256	P = "A", C = "B", assign C = "B" to P
3	194	"B"	"BC"	257	P = "B", C = "C", assign C = "C" to P
4	195	"C"	"CB"	258	P = "C", C = "B", assign C = "B" to P
5					P = "B", C = "C", assign P+C = "BC" to P
6	257	"BC"	"BCC"	259	P = "BC", C = "C", assign C = "C" to P
7	195	"C"	"CA"	260	P = "C", C = "A", assign C = "A" to P
8					P = "A", C = "B", assign P+C = "AB" to P
9	256	"AB"			P = "AB", C = ""
					end of stream

Encoder: The compressor output is the binary form of each column encoder output for characters representation, such as here: **193 194 195 257 195 256**.

3.3 Decompression Process with Examples

Algorithm 2 LZW Decompression Process

```
Initialize the dictionary with all possible single-length characters.
OLD = first input code.
Output translation of OLD (decoded sequence).
while not at the end of the encoded sequence do
    NEW = next input code.
    if NEW exists in the dictionary then
        S = translation of NEW.
    else
        S = translation of OLD + C.
    end if
    Output S.
    C = first character of S.
    Append translation of OLD + C to the dictionary.
    OLD = NEW.
end while
```

The decompression phase of the LZW algorithm is designed to reverse the compression process, reconstructing the original data from the compressed codes. The decompression algorithm begins with the same initial dictionary used during the compression phase, containing all individual symbols. It then reads the compressed data, consisting of a series of codes referencing sequences in the dictionary.

As the decompressor processes each code, it retrieves the corresponding sequence from the dictionary and adds it to the output. If the code refers to an existing sequence, the sequence is output directly. If the code refers to a new sequence, the decompressor will add the sequence to the dictionary, ensuring it can be used in subsequent decompression steps.

Example: LZW decompression process for the encoded input 193 194 195 257 195 256.

Initialize: Dictionary with all possible single-length characters, mapping ASCII character codes, for instance: dictionary[176] = "0", dictionary[193] = "A".

Table 3.2: Trace of the LZW Algorithm step by step: Decompression process

Step	Decoder Output	Dictionary		Explaination
		Key	Value	
1	"A"			OLD = 193
2	"B"	256	"AB"	NEW = 194, S = "B", C = "B", OLD = 194
3	"C"	257	"BC"	NEW = 195, S = "C", C = "C", OLD = 195
4	"BC"	258	"CB"	NEW = 257, S = "BC", C = "B", OLD = 257
5	"C"	259	"BCC"	NEW = 195, S = "C", C = "C", OLD = 195
6	"AB"	260	"CA"	NEW = 256, S = "AB", C = "A", OLD = 256
				end of stream

Decocoder: The decompressor output of encoded input is the string reconstructed from binary encoders, such as here: "**ABCBCCAB**".

4 Performance Analysis of LZW Compression

4.1 Compression Ratios

The effectiveness of compression is expressed as a ratio relating to the number of bits needed to express the message before and after compression. The compression ratio used here will be the uncompressed bit count divided by the compressed bit count. The resulting value, usually greater than one, indicates the factor of increased data density achieved by compression. For example, compression that serves to eliminate half the bits of a particular message is presented as fulfilling a 2.0 compression ratio, indicating that two-to-one compression has been achieved.

The compression ratios presented in the table below are derived from the original article "A Technique for High-Performance Data Compression"[2] by Terry Welch. These results, obtained through software simulation, are provided in Table 4.1 for various data types.

Table 4.1: Compression results for a variety of data types.

Data Type	**Compression Ratio**
English Text	1.8
Cobol Files	2 to 6
Floating Point Arrays	1.0
Formatted Scientific Data	2.1
System Log Data	2.6
Program Source Code	2.3
Object Code	1.5

English text. Text samples for compression were obtained from ASCII word processing files in a technical environment. Results were reasonably consistent for simple text, at a compression ratio of 1.8. Surprisingly, long individual documents did not compress better than groups of short documents, indicating that other factors (such as formatting or structural information) might be contributing more significantly to redundancy, rather than the content itself.

Cobol Files. A significant number of large Cobol files from several types of applications were compressed, producing widely variable results. Compression depends on record format, homogeneity across data records, and the extent of integer usage. These were eight-bit ASCII files, so the integer data would compress very well. A side experiment showed that one-third to two-thirds of the space in some of these files appeared as strings of repeated identical characters, indicating a high fraction of blank space (fixed-width record format).

Floating Point Arrays. Arrays of floating point numbers look pretty much like white noise and so they compress rather poorly. The fraction part is a nearly random bit pattern since there are no redundancy savings to offset the overhead.

Formatted Scientific Data. Most data used by Fortran programs tended to compress about 50 percent. This data included input data, primarily integers. It also included print files, which were ASCII-coded.

System Log Data. Information describing past system activity, such as job start and

stop times, is mostly formatted integers and is therefore reasonably compressible. This log data is used for recovery and constitutes perhaps 10 percent of the data stored on backup/recovery tapes. It tends to be in a tightly packed, fixed-length format, so the compression achieved is due to null fields and repetition in the data values.

Program Source Code. Source code can be compressed by a factor of better than two. It can be compressed better than text because words are frequently repeated and blank spaces are introduced by the source code format. Highly structured programming yields source code with greater compressibility than the average 2.3 factor cited here.

Object Code. Object code consists of arbitrary bit patterns and does not compress well. Uneven usage of opcodes and incomplete utilization of displacement fields would account for most of the compression achieved.

4.2 Time Complexity

The performance of LZW compression is influenced by the type of input data being processed. Different data types often exhibit varying compression ratios, as the efficiency of the algorithm depends on the frequency and patterns of repeating sequences within the input. For example, highly repetitive data tends to achieve better compression ratios, thereby reducing the computational workload per unit of compressed output. Conversely, inputs with little to no redundancy may lead to lower compression ratios and potentially increase the time required for dictionary operations. This relationship between data characteristics and compression efficiency underscores the need to evaluate LZW's performance under diverse data scenarios.

When applied to English text, the complexity depends on the structure of the input. We will explore both the Best Case and Worst Case scenarios, supported by mathematical reasoning. Additionally, as the dictionary grows, operations like searching and updating may become more computationally expensive, which can further affect performance in scenarios with large or non-repetitive datasets.

4.2.1 Best Case

The best-case scenario for LZW compression occurs when the input data exhibit highly repetitive patterns, allowing the algorithm to use its dictionary efficiently. For example, an input like *"aaaaaaa"* or *"abababab"* is ideal. In this case:

- After the initial iterations, most substrings encountered in the input are already present in the dictionary. The algorithm quickly matches these patterns with existing dictionary entries, minimizing the need for new insertions. Each search operation is $\mathcal{O}(1)$ on average when using a hash table for the dictionary.
- Since the input is repetitive, new entries are rarely added to the dictionary. As a result, the dictionary remains small, reducing memory usage and processing overhead. This efficient usage of the dictionary further enhances performance.
- The compression process involves scanning through the input and performing a constant-time dictionary lookup for each character. The overall complexity is $\mathcal{O}(L)$, where L is the length of the input. The decompression process follows a similar

pattern, as most patterns are reused, and the dictionary's small size ensures fast lookups. Thus, the complexity is $\mathcal{O}(L_c)$, where L_c is the compressed data length.

The chart below displays the results of several test cases, each containing a string with repetitive patterns of varying lengths:

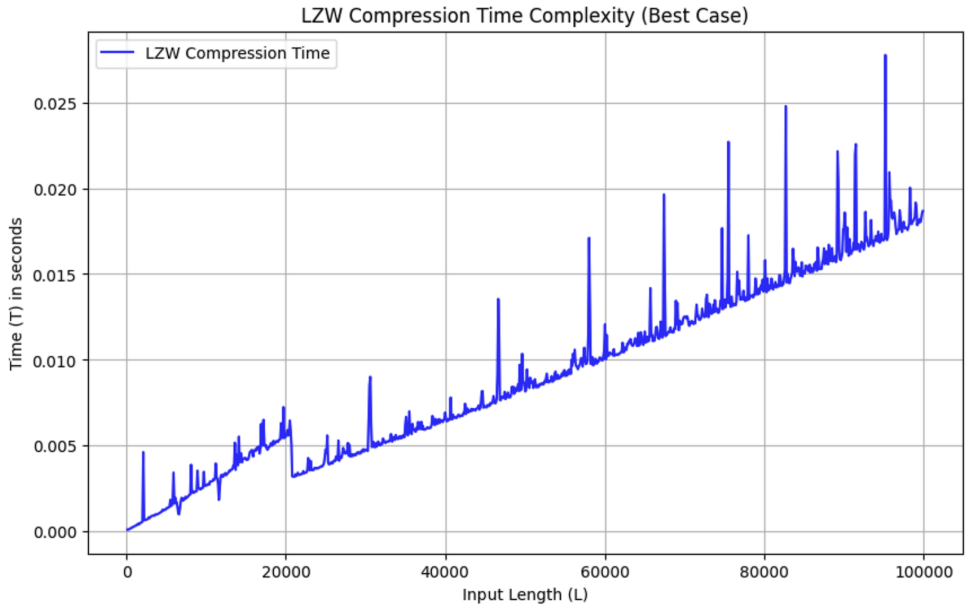

Figure 4.1: LZW Compression Runtime in Best Case

Based on this, we can prove that $T = \mathcal{O}(L)$ using the following chart:

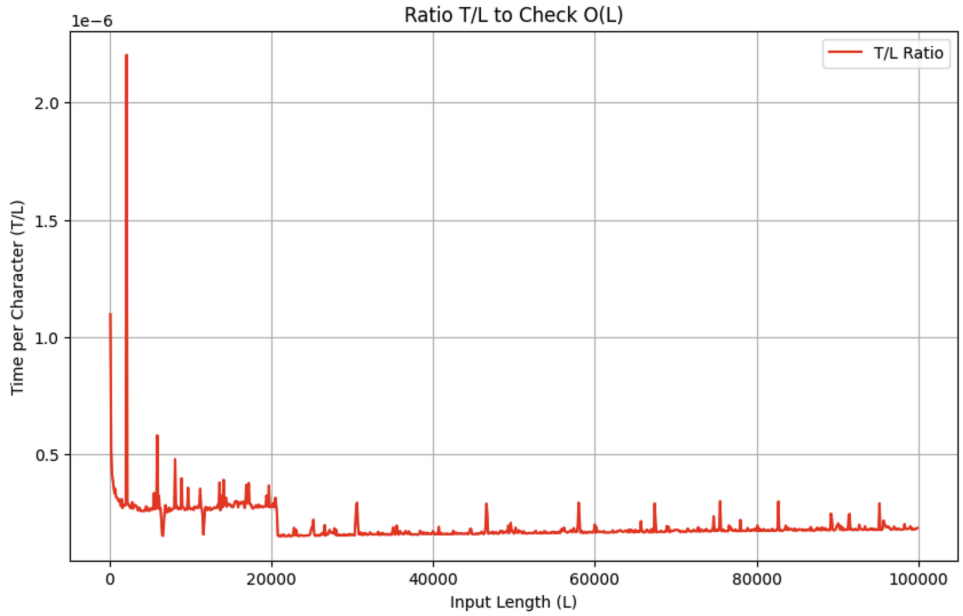

Figure 4.2: Best Case: $T = \mathcal{O}(L)$

4.2.2 Worst Case

The worst-case scenario for LZW compression arises when the input data lacks any repetition or structure, such as unique or random characters. For example, an input like "abcdefg..." forces the algorithm to handle every substring as a new entity. In this case:

- Every substring encountered in the input is unique and not found in the dictionary. This means that the algorithm frequently needs to perform dictionary insertions. Although lookups are $\mathcal{O}(1)$ on average, the frequent insertions can cause the dictionary size to grow rapidly.
- With each new substring, the dictionary expands, potentially reaching its maximum size M quickly. If the dictionary size is not capped, this continuous growth consumes more memory and may degrade performance. In cases where a dictionary cap is implemented, frequent resets or replacements add additional overhead.
- Compression and decompression in the worst case are heavily affected by the rapid growth of the dictionary. Each character or code requires a dictionary lookup, and most substrings demand insertions. While the average-case lookup complexity remains $\mathcal{O}(1)$ with hash tables, poor hash distributions or collisions can degrade lookup performance to $\mathcal{O}(M)$, where M is the dictionary size. Consequently, the compression complexity can reach $\mathcal{O}(L \cdot M)$, and the decompression complexity can grow to $\mathcal{O}(L_c \cdot M)$, where L, L_c are the input and compressed lengths, respectively.

The following chart presents the results of multiple test cases, each containing a random string (barely repetitive) of different lengths:

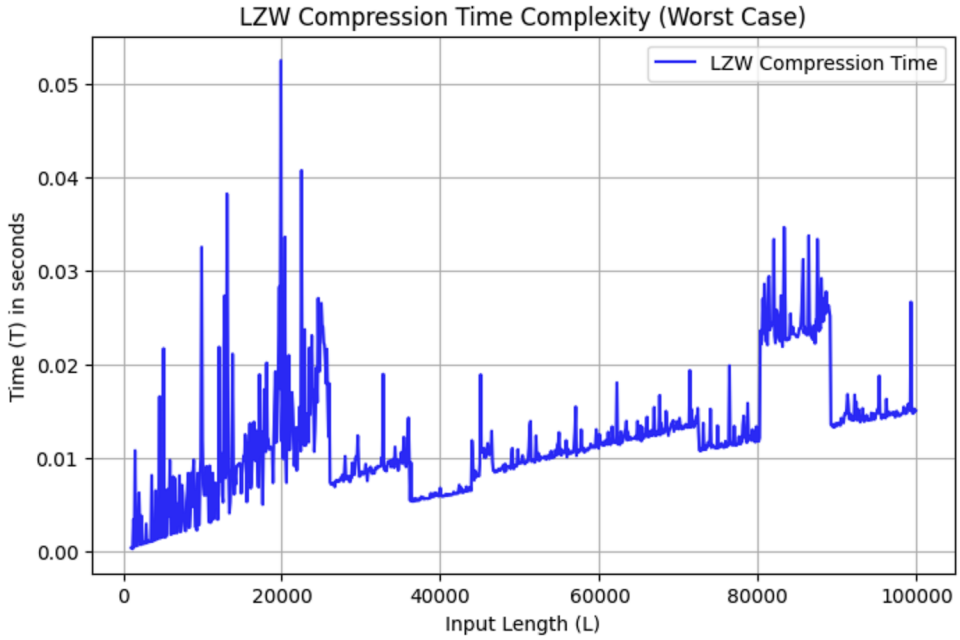

Figure 4.3: LZW Compression Runtime in Worst Case

From this, we can demonstrate that $T = \mathcal{O}(L \cdot M)$ through the following chart:

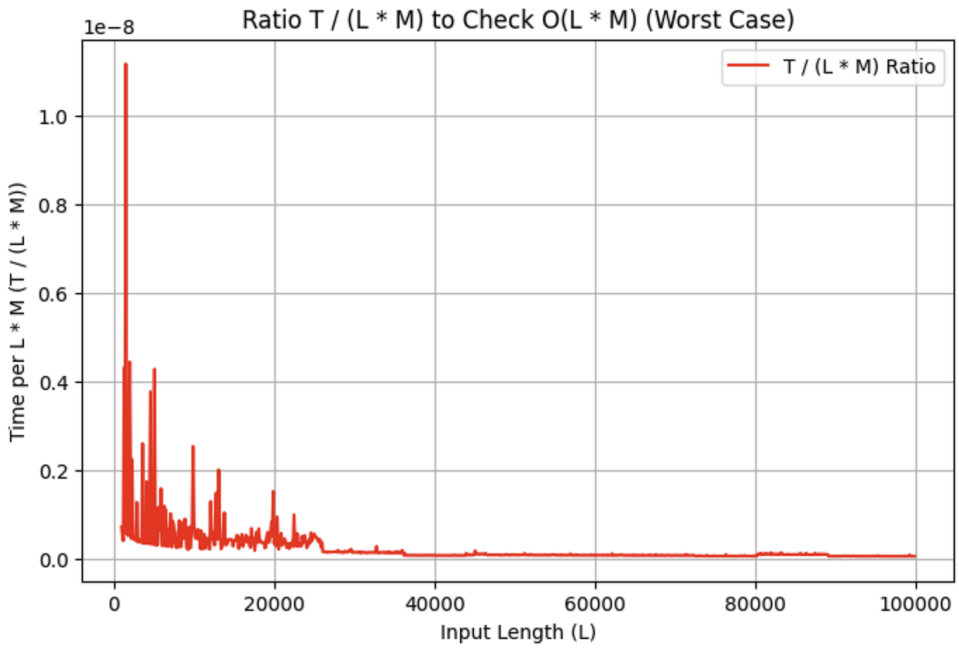

Figure 4.4: Worst Case: $T = \mathcal{O}(L \cdot M)$

5 Overview of LZW-Related Algorithms

LZ77 and LZ78 are the two most common lossless data compression algorithms, which Abraham Lempel and Jacob Ziv published in 1977[3] and 1978 [1]. They are sometimes referred to as LZ1 and LZ2 respectively. These two theoretical dictionary algorithms form the basis for many variations including LZW, LZSS, and others. Additionally, they are also the basis of several ubiquitous compression schemes, including GIF and the DEFLATE algorithm used in PNG.

5.1 LZ77

LZ77[4] operates on the concept of a sliding window, which tracks a portion of previously processed data. The algorithm identifies the longest match between a substring in the look-ahead buffer and a substring within the sliding window. After locating a match, the window slides forward based on the length of the match.

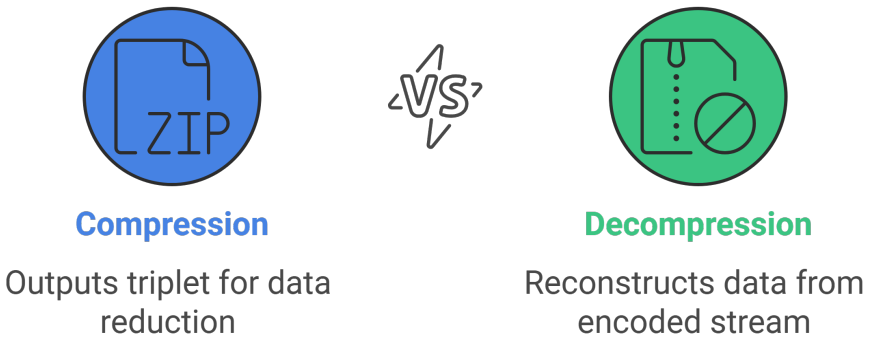

Figure 5.1: LZ77 Compression and Decompression

During compression, the algorithm outputs a triplet: a pointer to the starting position of the match (relative to the start of the sliding window), the length of the match, and the first unmatched character in the look-ahead buffer. If no match is found, the algorithm outputs a null pointer and the unmatched character.

For decompression, the process starts by reading the encoded stream sequentially. When a null pointer is encountered, the algorithm appends the corresponding character to the output. For non-null pointers, the algorithm retrieves the matched substring using the pointer and length and then appends the retrieved substring to the output.

5.2 LZ78

LZ78-based schemes work by entering phrases into a dictionary and then, when a repeat occurrence of that particular phrase is found, outputting the dictionary index instead of the phrase.
At each step, LZ78[5] outputs a pair $(i,\ a)$, where i is the index of the phrase in the dictionary and a is the next symbol following the phrase. The dictionary is represented like the trie with numbered nodes. If we go from the root to a certain node, we will get a phrase from the input text.

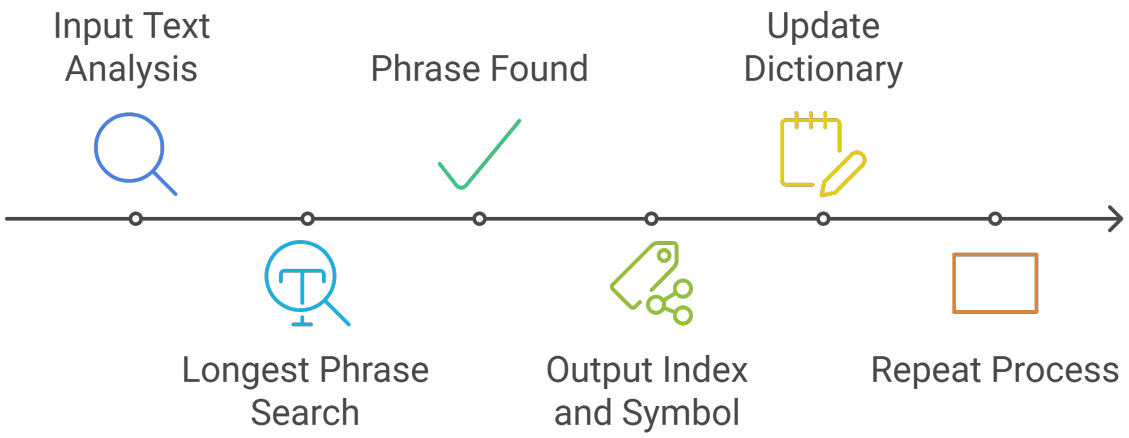

Figure 5.2: LZ78 Compression Process

In each step, we look for the longest phrase in the dictionary, that would correspond to the unprocessed part of the input text. The index of this phrase together with the symbol, which follows the found part in the input text, is then sent to the output. The old phrase extended by the new symbol is then put into the dictionary. This new phrase is numbered by the smallest possible number.

5.3 Comparison between LZ77, LZ78, and LZW

LZ77, LZ78, and LZW are foundational compression algorithms that differ in their approach to handling redundancy in data.

LZ77 operates using a sliding window, where it searches for the longest match between the look-ahead buffer (unprocessed data) and the sliding window (recently processed data). It outputs a triplet *(offset, length, next symbol)*, with the offset indicating the starting position of the match, the length specifying the number of matched characters, and the next symbol representing the first unmatched character. This makes LZ77 effective for compressing data with localized repetition, but its performance diminishes when dealing with long-range patterns or globally repetitive data.

In contrast, LZ78 constructs an explicit and growing dictionary to store phrases dynamically as it processes the input. At each step, it outputs a pair *(index, next symbol)*, where the index refers to the dictionary entry of the longest matching phrase, and the next symbol is the character that follows the match. Unlike LZ77, LZ78 does not limit itself to a sliding window, enabling it to capture patterns that recur over larger ranges in the input. However, the need to transmit the raw next symbol alongside the dictionary index can reduce compression efficiency for highly repetitive data.

LZW, an enhancement of LZ78, pre-initializes the dictionary with all single-character symbols, eliminating the need to transmit raw symbols in the output. This allows LZW to rely entirely on dictionary indices for encoding, which makes it more efficient than LZ78 for repetitive data. As the compression progresses, LZW extends the dictionary with

new phrases formed from previously matched sequences. While this approach improves compression ratios for data with significant redundancy, it also introduces complexity in managing and synchronizing the dictionary between the encoder and decoder.

In terms of practical applications, LZ77 forms the basis of the widely used DEFLATE algorithm, which powers ZIP and PNG formats, with variants like LZSS optimizing its performance. LZ78 serves primarily as a theoretical foundation, influencing the development of algorithms like LZW, which gained widespread adoption in formats such as GIF. Each algorithm has its strengths: LZ77 excels with localized patterns, LZ78 handles long-range repetitions effectively, and LZW achieves high compression efficiency by using only dictionary indices. These differences make them suitable for various types of data and applications.

Table 5.1: Comparison of LZ77, LZ78, and LZW.

Feature	**LZ77**	**LZ78**	**LZW**
Dictionary Type	Sliding window	Incremental, explicit trie	Pre-initialized dictionary
Output	(offset, length, symbol)	(index, symbol)	(index)
Strengths	Local repetition, flexibility	Long-range patterns	High compression ratio
Weaknesses	Limited range	Includes raw symbols	Complex initialization
Applications	DEFLATE, ZIP, PNG	Basis for LZW	GIF, legacy systems

6 Conclusion

The Lempel-Ziv-Welch (LZW) algorithm, developed in 1984 by Abraham Lempel, Jakob Ziv, and Terry Welch, is a pivotal contribution to the field of lossless data compression. Renowned for its ability to efficiently reduce file sizes without losing information, LZW has become indispensable in various domains, including its prominent use in the GIF image format and the Unix compression utility compress.

As a dictionary-based compression algorithm, LZW optimizes storage and transmission by identifying repetitive patterns in the data and replacing them with shorter codes. This approach enables it to handle text, images, and other data types effectively, making it versatile for both general-purpose and specialized applications.

LZW's simplicity, efficiency, and adaptability have cemented its place as a foundational algorithm in computer science. Despite advancements in compression technology, LZW remains relevant, especially in legacy systems and formats where computational efficiency and lossless output are critical. Yet, its limitations highlight the need for selecting appropriate algorithms tailored to specific use cases.

In conclusion, the LZW algorithm is a testament to the power of innovative thinking in computer science, showcasing how efficient data representation can drive both technological and practical advancements in data storage and transmission.

7 References

[1] Jacob Ziv and Abraham Lempel, "Compression of individual sequences via variable-rate coding", *IEEE Trans. Inf. Theory*, 24(5):530–536, 1978. Available: https://api.semanticscholar.org/CorpusID:9267632

[2] Terry A. Welch, "A technique for high-performance data compression", *Computer*,17(06):8–19, 1984. Available: https://courses.cs.duke.edu/spring03/cps296.5/papers/welch_1984_technique_for.pdf

[3] Jacob Ziv and Abraham Lempel, "A universal algorithm for sequential data compression", *IEEE Transactions on Information Theory*, 23:337–343, 1977.

[4] From Wikipedia, the free encyclopedia, "LZ77 and LZ78". Available: https://en.wikipedia.org/wiki/LZ77_and_LZ78

[5] From stringology.org, Czech Technical University in Prague, "Data Compression - LZ - 78". Available: https://www.stringology.org/DataCompression/lz78/index_en.html

8 Appendices

Table 8.1: Evaluation of Completion Rates for General Project Requirements

General Requirements	**Completion Rate (%)**	**Remarks**
Group and member information (Student ID, full name, etc.)	100	Information provided completely and accurately.
Work assignment table, which includes information on each task assigned to team members, along with the completion rate of each member compared to the assigned tasks.	100	All tasks and contributions documented in detail.
Self-evaluation of the completion rate of the project requirements.	100	Honest and thorough self-assessment included.
Detailed description of each algorithm. Illustrative images and diagrams are encouraged.	100	Algorithms are well-explained with clear tables and examples.
Description of the test cases and experiment results along with your insights.	100	Test cases and results thoroughly analyzed with meaningful insights provided.
The report needs to be well-formatted and exported to PDF. If there are figures cut-off by the page break, etc., points will be deducted.	100	Formatting is clean and all figures are properly aligned without any cut-off.
References	100	Complete and correctly formatted reference list included.

Table 8.2: Evaluation of Completion Rates for Project LZW Compression Requirements

Requirements	**Completion Rate (%)**	**Remarks**
Introduce its background: history and applications.	100	Background and applications of the algorithm are thoroughly explored, providing clear historical context and modern use cases.
Trace the algorithm, step by step, for both compression and decompression phases, using simple examples.	100	Step-by-step tracing includes clear examples for both phases, enhancing understanding.
Analyze the algorithm's time complexity in the best case and worst case.	100	Comprehensive analysis provided with detailed calculations for both scenarios.
Present a comprehensive overview of LZW-related algorithms, such as the LZ77 and LZ78. For each algorithm, just give the overall idea while ignoring the details.	100	LZW-related algorithms are briefly but effectively summarized, focusing on key concepts.