# 20XTE9 – DATA COMPRESSION

ASSIGNMENT PRESENTATION

DONE BY


**21PT28 - SHERIN. J. A**

**21PT32 - THIRULOASHANA. A**

**21PT40 - S. NITISH KRISHNA**


ON

# IMPLEMENTATION OF LZSS ALGORITHM

Project report submitted in partial fulfilment of the requirements for the degree of

## FIVE YEAR INTEGRATED

## M.Sc. THEORETICAL COMPUTER SCIENCE

of Anna University

**MARCH - 2025**


## PSG COLLEGE OF TECHNOLOGY

(Autonomous Institution)
COIMBATORE – 641 004

# CONTENTS

# CHAPTER - 1

## 1. INTRODUCTION

Data compression is a fundamental technique in multimedia systems, aimed at reducing the size of data for efficient storage and transmission while preserving its integrity. Among various compression algorithms, the LZSS (Lempel-Ziv-Storer-Szymanski) algorithm stands out as an enhancement of the LZ77 algorithm, offering a balance between compression efficiency and computational simplicity. LZSS employs a sliding window approach, utilizing a search buffer and a lookahead buffer to identify repeated patterns in the data, encoding them as pointers when beneficial, or as literals otherwise.

In this project, we implement the LZSS algorithm using Python, integrated with a Streamlit-based graphical interface to demonstrate its functionality interactively. The algorithm compresses input text by finding the longest matching substring within a defined search buffer and encoding it as an offset-length pair, or outputting a literal character when no beneficial match is found. The compressed data is then decompressed to verify the lossless nature of the algorithm. This implementation highlights the practical utility of LZSS in real-world applications, such as file compression utilities and embedded systems, where both speed and efficiency are critical.

The report details the methodology, implementation, and analysis of the LZSS algorithm, supported by demo images of the executed code and compression statistics, showcasing its effectiveness in reducing data size while maintaining fidelity.

# CHAPTER - 2

# 2. LZSS ALGORITHM

## 2.1 PROPERTIES

- Sliding Window Mechanism: Uses a search buffer and lookahead buffer to identify repeated patterns.
- Lossless Compression: Ensures no data loss during compression and decompression.
- Adaptive Encoding: Switches between pointers and literals based on efficiency.
- Variable-Length Coding: Adjusts output based on match length and offset.

## 2.2 DETAILED METHODOLOGY

### 2.2.1 Input Preprocessing:

The process begins with accepting a text input string from the user. The search buffer size and lookahead buffer size are configurable parameters, allowing flexibility in compression performance tuning.

### 2.2.2 Compression Process:

- The algorithm initializes an empty search buffer and fills the lookahead buffer with the initial portion of the input text.
- It searches for the longest matching substring between the search buffer and lookahead buffer.
- If a match of length $\geq 1$ is found and deemed efficient, it outputs a pointer (flag=1, (offset, length)). Otherwise, it outputs a literal (flag=0, character).
- The search buffer is updated with the processed text, constrained by the search buffer size, and the lookahead buffer shifts forward, refilling from the input as needed.
- This process repeats until the entire input is processed.

### 2.2.3 Decompression Process:

- The compressed data, consisting of flag-value pairs, is processed sequentially.
- For literals (flag=0), the character is appended to the output and buffer.
- For pointers (flag=1), the algorithm retrieves the substring from the buffer using the offset and length, appending it to both the buffer and output.
- The buffer is maintained within the search size limit, ensuring accurate reconstruction.

### 2.2.4 Output Generation:

The compressed output is a list of tuples representing literals or pointers, while decompression reconstructs the original text, verifying the algorithm's lossless property.

## 2.3 ADVANTAGES

- Efficient for repetitive data
- Simple implementation
- Lossless compression
- Fast decompression

## 2.4 DISADVANTAGES

- Limited by search buffer size
- Less effective for non-repetitive data
- Overhead from flag bits

## 2.5 APPLICATIONS

- File compression (e.g., ZIP, RAR)
- Embedded systems
- Network data transmission
- Text and binary data storage

# CHAPTER - 3

## 3. PYTHON IMPLEMENTATION

```
%%writefile lzss_streamlit.py


import streamlit as st
from tabulate import tabulate
import pandas as pd

def lzss_compress(text, search_size=7, lookahead_size=5):
    search_buffer = ""
    lookahead_buffer = text[:lookahead_size]
    pos = lookahead_size
    output = []
    steps = []
    i = 0

    while lookahead_buffer:
        # Find the longest match in the search buffer
        longest_match_len = 0
        longest_match_offset = 0

        for j in range(len(search_buffer)):
            match_len = 0
            while (match_len < len(lookahead_buffer) and
                   j + match_len < len(search_buffer) and
                   search_buffer[j + match_len] ==
lookahead_buffer[match_len]):
                match_len += 1

            if match_len >= 1 and match_len > longest_match_len:
                longest_match_len = match_len
```

```
                longest_match_offset = len(search_buffer) - j  # 
Offset from end

        # Decide output based on match
        if longest_match_len >= 1:  # Only use pointers when they 
save space
            output.append((1, (longest_match_offset, 
longest_match_len)))
            matched_text = lookahead_buffer[:longest_match_len]
        else:
            output.append((0, lookahead_buffer[0]))
            matched_text = lookahead_buffer[0]
            longest_match_len = 1

        # Record step
        steps.append([i, search_buffer, lookahead_buffer, 
                      f"({output[-1][0]}, {repr(output[-1][1]) if 
output[-1][0] == 0 else output[-1][1]})"])

        # Update buffers
        search_buffer += matched_text
        if len(search_buffer) > search_size:
            search_buffer = search_buffer[-search_size:]

        # Move forward in the lookahead buffer
        lookahead_buffer = lookahead_buffer[longest_match_len:]

        # Refill lookahead buffer
        refill_length = min(longest_match_len, len(text) - pos)
        if refill_length > 0:
            lookahead_buffer += text[pos:pos + refill_length]
            pos += refill_length

        i += 1

    return output, steps
```

```python
# LZSS Decompression Function
def lzss_decompress(compressed, search_size=7):
    buffer = ""
    output = ""
    steps = []

    for i, (flag, value) in enumerate(compressed):
        if flag == 0:  # Literal
            buffer += value
            output += value
            if len(buffer) > search_size:
                buffer = buffer[-search_size:]
        else:  # Match
            offset, length = value
            # Handle the case where we need to copy from what we're currently generating
            decoded = ""
            for j in range(length):
                if j < offset:
                    char = buffer[len(buffer) - offset + j]
                else:
                    char = decoded[j - offset]
                decoded += char

            buffer += decoded
            output += decoded
            if len(buffer) > search_size:
                buffer = buffer[-search_size:]

        steps.append([i, buffer, f"({flag}, {repr(value) if flag == 0 else value})", output])

    return output, steps

# Calculate compression statistics
```

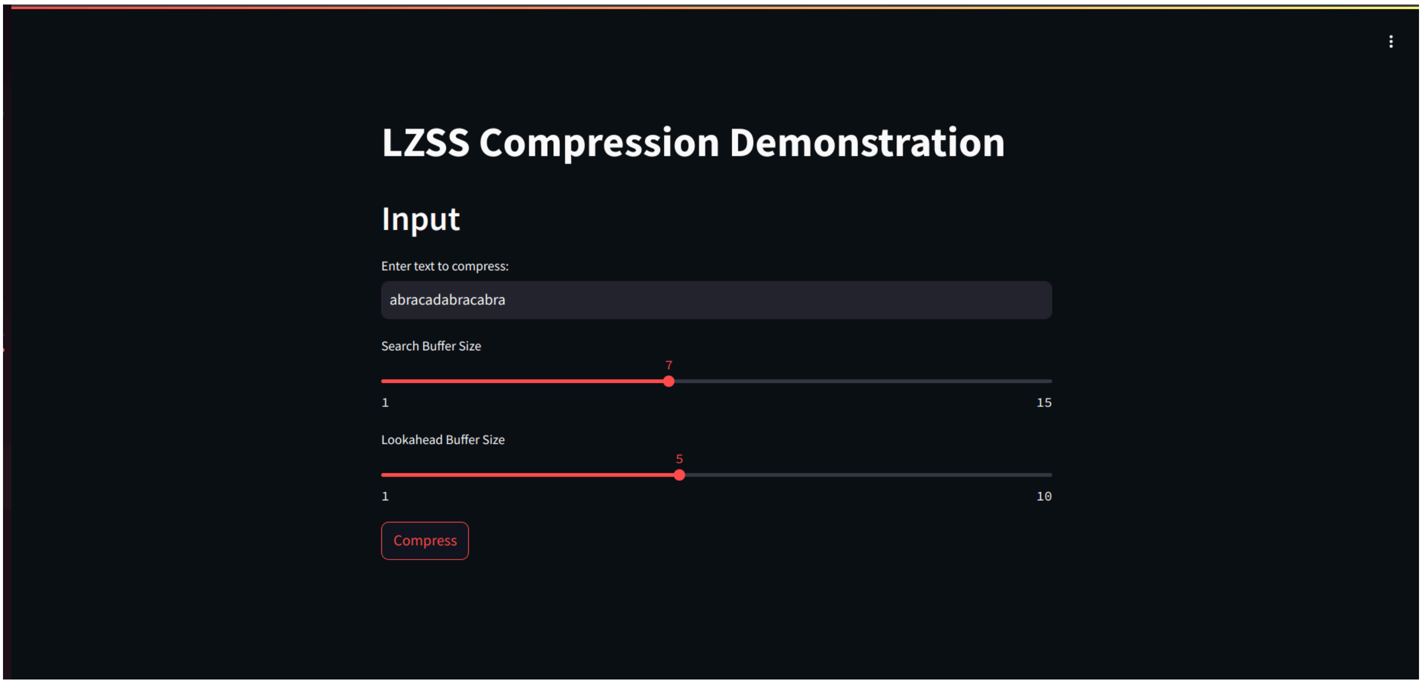
```python
def calculate_compression_stats(original_text, compressed_data):
    # Calculate original size (1 byte per character)
    original_size = len(original_text)

    # Calculate compressed size
    compressed_size = 0
    for flag, value in compressed_data:
        if flag == 0:  # Literal: 1 bit flag + 8 bits for character
            compressed_size += 1 + 8
        else:  # Pointer: 1 bit flag + bits for offset + bits for
length
            compressed_size += 1 + 3 + 3

    # Convert bits to bytes (round up to nearest byte)
    compressed_size_bytes = (compressed_size + 7) // 8

    # Calculate compression ratio
    compression_ratio = original_size / compressed_size_bytes if
compressed_size_bytes > 0 else 0

    # Calculate space savings percentage
    space_savings = (1 - (compressed_size_bytes / original_size)) *
100 if original_size > 0 else 0

    return {
        "original_size_bytes": original_size,
        "compressed_size_bits": compressed_size,
        "compressed_size_bytes": compressed_size_bytes,
        "compression_ratio": compression_ratio,
        "space_savings_percentage": space_savings
    }

# Streamlit App
def main():
    st.title("LZSS Compression Demonstration")
```

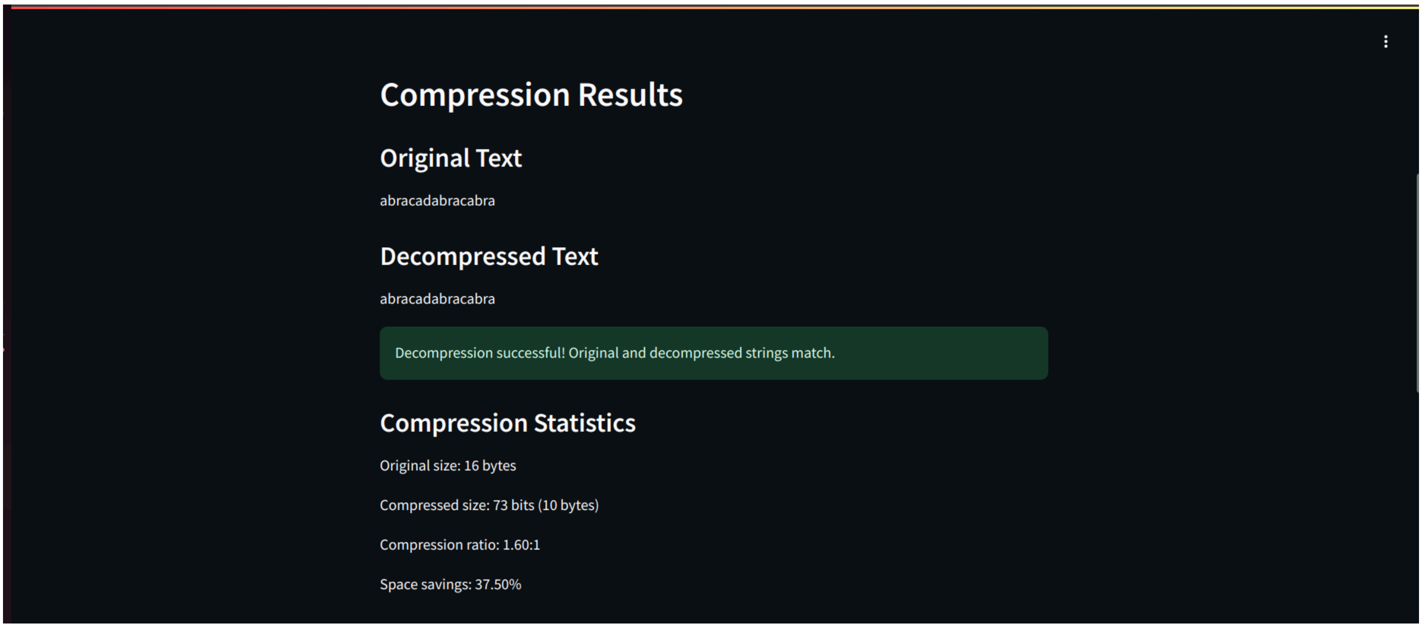
```
    # Input section
    st.header("Input")
    input_text = st.text_input("Enter text to compress:",
"abracadabracabra")
    search_size = st.slider("Search Buffer Size", min_value=1,
max_value=15, value=7)
    lookahead_size = st.slider("Lookahead Buffer Size",
min_value=1, max_value=10, value=5)

    # Compression button
    if st.button("Compress"):
        # Perform compression
        compressed, compression_steps = lzss_compress(input_text,
search_size, lookahead_size)

        # Decompress to verify
        decompressed, decompression_steps =
lzss_decompress(compressed)

        # Calculate statistics
        stats = calculate_compression_stats(input_text, compressed)

        # Display results
        st.header("Compression Results")

        # Original Text
        st.subheader("Original Text")
        st.text(input_text)

        # Decompressed Text
        st.subheader("Decompressed Text")
        st.text(decompressed)

        # Verification
        if input_text == decompressed:
            st.success("Decompression successful! Original and
```

```
decompressed strings match.")
        else:
            st.error("Error: Decompression failed. Strings do not
match.")

        # Compression Statistics
        st.subheader("Compression Statistics")
        st.write(f"Original size: {stats['original_size_bytes']}
bytes")
        st.write(f"Compressed size: {stats['compressed_size_bits']}
bits ({stats['compressed_size_bytes']} bytes)")
        st.write(f"Compression ratio:
{stats['compression_ratio']:.2f}:1")
        st.write(f"Space savings:
{stats['space_savings_percentage']:.2f}%")

        # Compression Steps (Detailed View)
        st.subheader("Compression Steps")
        compression_steps_df = pd.DataFrame(compression_steps,
                                            columns=["Step",
"Search Buffer", "Lookahead Buffer", "Output"])
        st.dataframe(compression_steps_df)

        # Decompression Steps (Detailed View)
        st.subheader("Decompression Steps")
        decompression_steps_df = pd.DataFrame(decompression_steps,
                                              columns=["Step",
"Buffer", "Input", "Output"])
        st.dataframe(decompression_steps_df)




# Run the Streamlit app
if __name__ == "__main__":
    main()
```

# CHAPTER - 4

# 4. OUTPUT - DEMONSTRATION



## Compression Steps

| | Step | Search Buffer | Lookahead Buffer | Output |
|---|---|---|---|---|
| 0 | 0 | | abrac | (0, 'a') |
| 1 | 1 | a | braca | (0, 'b') |
| 2 | 2 | ab | racad | (0, 'r') |
| 3 | 3 | abr | acada | (1, (3, 1)) |
| 4 | 4 | abra | cadab | (0, 'c') |
| 5 | 5 | abrac | adabr | (1, (5, 1)) |
| 6 | 6 | abraca | dabra | (0, 'd') |
| 7 | 7 | abracad | abrac | (1, (7, 5)) |
| 8 | 8 | adabrac | abra | (1, (5, 4)) |

## Decompression Steps

| | Step | Buffer | Input | Output |
|---|---|---|---|---|
| 0 | 0 | a | (0, 'a') | a |
| 1 | 1 | ab | (0, 'b') | ab |
| 2 | 2 | abr | (0, 'r') | abr |
| 3 | 3 | abra | (1, (3, 1)) | abra |
| 4 | 4 | abrac | (0, 'c') | abrac |
| 5 | 5 | abraca | (1, (5, 1)) | abraca |
| 6 | 6 | abracad | (0, 'd') | abracad |
| 7 | 7 | adabrac | (1, (7, 5)) | abracadabrac |
| 8 | 8 | racabra | (1, (5, 4)) | abracadabracabra |

# 5. CONCLUSION

The LZSS algorithm proves to be an effective lossless compression technique, particularly for data with repetitive patterns. Its sliding window approach, combined with adaptive encoding, provides a practical solution for reducing data size without sacrificing integrity. The Python implementation with a Streamlit interface demonstrates its functionality interactively, offering insights into the compression and decompression processes. While limited by buffer size and less effective for non-repetitive data, LZSS remains valuable in applications like file compression and embedded systems, balancing simplicity and efficiency.

# 6. REFERENCES

Salomon, D. (2007). *Data Compression: The Complete Reference*. Springer.

Khalid Sayood *Introduction to Data Compression*

Streamlit Documentation. (2024). Retrieved from https://docs.streamlit.io