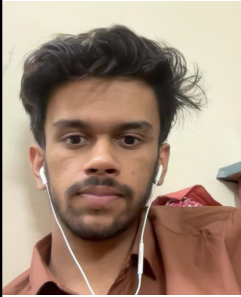




## PROJECT AND TEAM INFORMATION

### Project Title

*Efficient LZW Compression/Decompression Using Trie Dictionary*

### Student / Team Information

<b>Team ID:</b>	<u><a href="#">DAA-IV-T251</a></u>
<b>Team member 1 (Team Lead)</b> <i>Sumit Kumar</i> <i>Student ID: 23021201</i> <i>Email id: <a href="mailto:sumitkumarbittuair@gmail.com">sumitkumarbittuair@gmail.com</a></i>	
<b>Team member 2</b> <i>Aviral Maheshwari</i> <i>Student ID: 230213637</i> <i>Email id: <a href="mailto:maheshwariaviral05@gmail.com">maheshwariaviral05@gmail.com</a></i>	
<b>Team member 3</b> <i>Anubhav Vashistha</i> <i>Student ID: 230211649</i> <i>Email id: <a href="mailto:anubhavvashistha2005@gmail.com">anubhavvashistha2005@gmail.com</a></i>	

## PROPOSAL DESCRIPTION

### Motivation

Data compression is essential for reducing storage requirements and improving transmission efficiency. The **Lempel-Ziv-Welch (LZW)** algorithm is widely used in formats like GIF, TIFF, and ZIP. However, traditional LZW implementations use hash tables for dictionary management, which can be inefficient for large datasets due to high memory usage and lookup times.

This project aims to optimize LZW compression by using a **Trie (Prefix Tree)** for dictionary storage. A trie improves efficiency by:

- **Faster string matching** ( $O(k)$  lookup time, where  $k$  is the string length).
- **Reduced memory overhead** compared to hash tables when storing similar prefixes.
- **Better scalability** for large text and repetitive data.

### State of the Art / Current solution

Current LZW implementations typically use:

- **Hash Tables:** Fast but memory-intensive, with  $O(1)$  average lookup but  $O(n)$  worst-case collisions.
- **Arrays:** Simple but slow for dynamic dictionary growth.
- **Binary Search Trees (BSTs):** Balanced variants (AVL, Red-Black) offer  $O(\log n)$  lookups but are complex.

### Project Goals and Milestones

Goals:

- Implement **LZW compression** using a **Trie dictionary** for faster string matching.
- Develop **LZW decompression** with dynamic dictionary reconstruction.
- Benchmark against traditional hash-based LZW for **speed and compression ratio**.

Milestones:

- Implement basic Trie structure for dictionary storage.
- Integrate Trie into LZW compression.
- Implement decompression with dynamic dictionary rebuilding.
- Optimize memory usage and benchmark performance.
- Final testing, documentation, and report.

## Project Approach

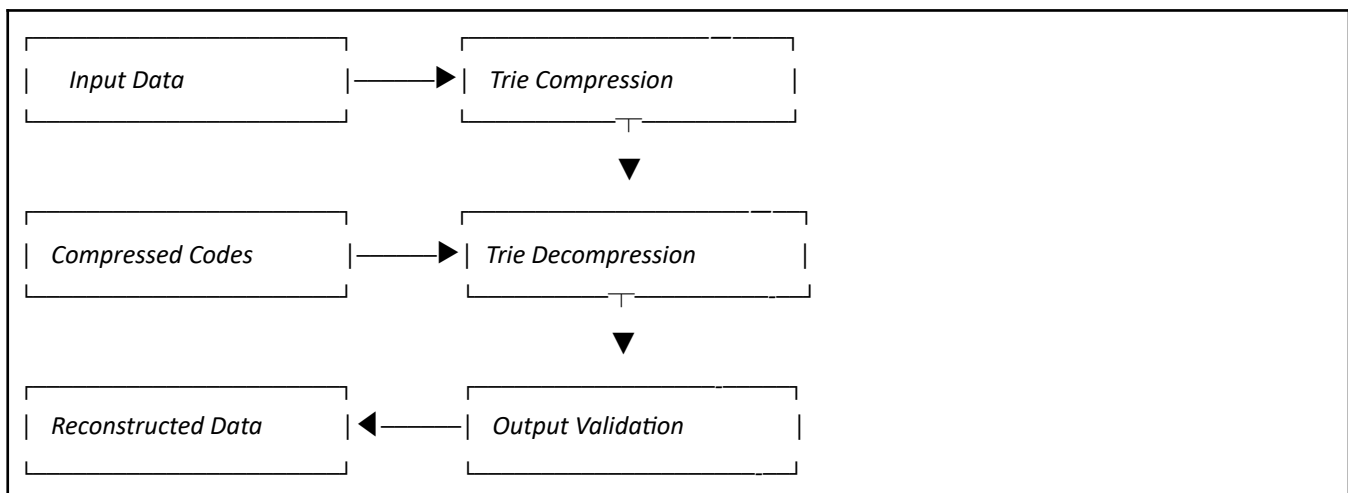
### Solution Design:

- **Trie-Based Dictionary:** Each node stores a character and a unique code.
- **Compression:**
  - Traverse input, building strings in the trie.
  - Emit codes when no further matches exist.
- **Decompression:**
  - Rebuild dictionary dynamically from compressed codes.
  - Handle edge cases (e.g., "cScSc" pattern).

### Technologies:

- **Language:** C++ (for performance-critical operations).
- **Data Structures:** `unordered_map` (for Trie children), `vector` (for dictionary).
- **Benchmarking:** Compare against `std::map` and hash-based LZW.

## System Architecture (High Level Diagram)



## Project Outcome / Deliverables

- *Working C++ implementation of Trie-based LZW.*
- *Performance benchmarks vs. hash-table LZW.*
- **Documentation** on optimization techniques.
- **Report** comparing compression ratios and speed.

## Assumptions

- *Input data is primarily **text-based** (optimal for LZW).*
- *Dictionary size is 16-bit (65,536 entries).*
- *No hardware acceleration is used (pure software implementation).*

## References

- *GNU **gzip** and **zlib** implementations for benchmarking.*
- Ziv, J., & Lempel, A. (1978). "Compression of Individual Sequences via Variable-Rate Coding".  
[ IEEE Transactions on Information Theory ]
- GitHub: "LZW implementations in C++" <https://github.com/search?q=LZW+C%2B%2B>
- Google: "Zopfli Compression Algorithm" <https://github.com/google/zopfli>
- Willcock, J., et al. (2006). "Trie-based Data Structures for Accelerating IP Routing".  
[ IEEE Symposium on High Performance Interconnects ]