Group O - Lempel-Ziv Compression of a Song

Nenad Jakovchevski 89221061 | Ekaterina Bochvaroska 89221049 | Kristina Piiarska 89221337

Motivation & background

Why use LZW on WAV audio?

LZW is a simple, lossless compression algorithm that builds a dictionary of repeating bit-patterns as it processes raw data. Applied to PCM audio, it should exploit any exact repetition—silence, loops or recurring waveforms—without discarding a single sample.

We test this on four very different clips—a pure sine wave, cricket chirps, repeated speech and complex music—by comparing LZW-compressed WAV files against their MP3 counterparts.

WAV	MP3
Lossless Quality	***Lossy Compression***
Tends to be Significantly LARGER in File Size, As it stores audio in a a raw, uncompressed format	***Practical with Significantly SMALLER in File Size***, As it discards some audio data when compressing
Considered a standard audio format	***Popular choice for online music services***

Example: 1 min of CD-quality audio = ~10 MB WAV vs. ~1 MB MP3

Hypothesis: For every type of audio—from a pure sine tone, to natural sounds, to spoken words, to full-band music—a lossless, bit-level LZW compression of the WAV file will always produce a smaller file than the MP3 version of the same clip.

Data preparation and processing

1. Select an Audio Files.

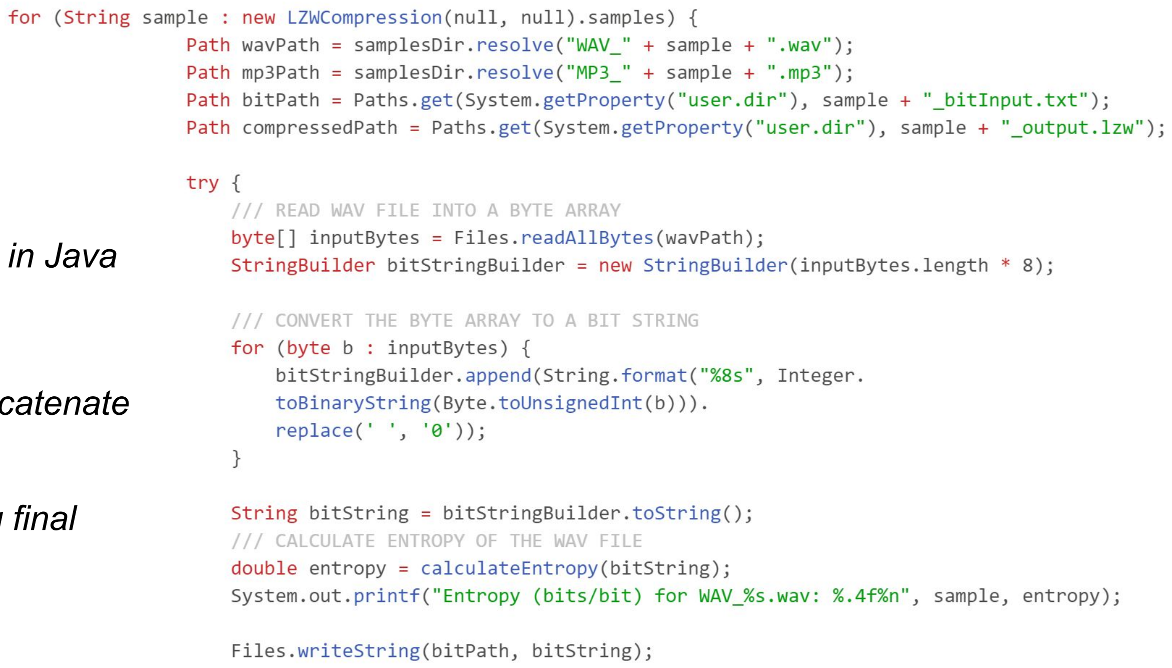
2. Read Raw Audio Data.

- *load the WAV file into a byte array in Java*

3. Convert to Bit Representation

- *each byte → 8-bit binary form concatenate to form a continuous bit string*

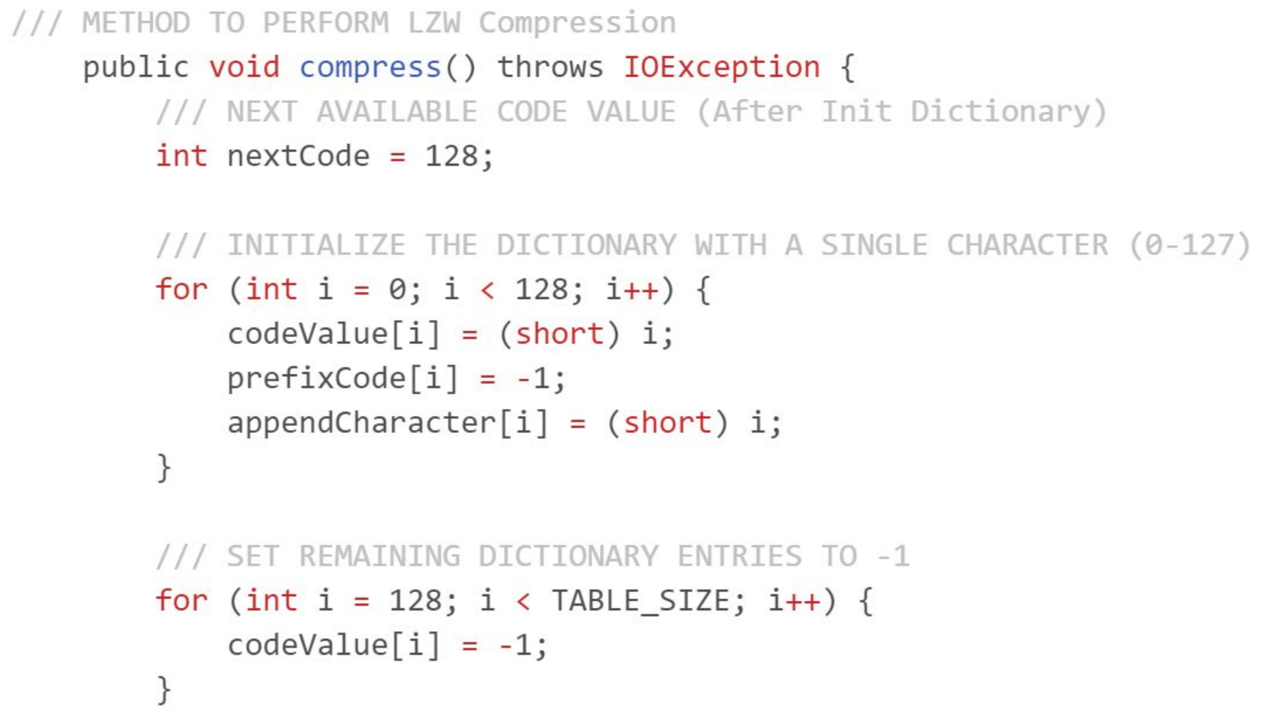
4. Calculate entropy H before comparing final results



```
for (String sample : new LZWCompression(null, null).samples) {
                Path wavPath = samplesDir.resolve("WAV_" + sample + ".wav");
                Path mp3Path = samplesDir.resolve("MP3_" + sample + ".mp3");
                Path bitPath = Paths.get(System.getProperty("user.dir"), sample + "_bitInput.txt");
                Path compressedPath = Paths.get(System.getProperty("user.dir"), sample + "_output.lzw");

                try {
                    /// READ WAV FILE INTO A BYTE ARRAY
                    byte[] inputBytes = Files.readAllBytes(wavPath);
                    StringBuilder bitStringBuilder = new StringBuilder(inputBytes.length * 8);

                    /// CONVERT THE BYTE ARRAY TO A BIT STRING
                    for (byte b : inputBytes) {
                        bitStringBuilder.append(String.format("%8s", Integer.
                        toBinaryString(Byte.toUnsignedInt(b))).
                        replace(' ', '0'));
                    }

                    String bitString = bitStringBuilder.toString();
                    /// CALCULATE ENTROPY OF THE WAV FILE
                    double entropy = calculateEntropy(bitString);
                    System.out.printf("Entropy (bits/bit) for WAV_%s.wav: %.4f%n", sample, entropy);

                    Files.writeString(bitPath, bitString);
```

Initialize the dictionary

Start with a base dictionary of 128 single-bit entries



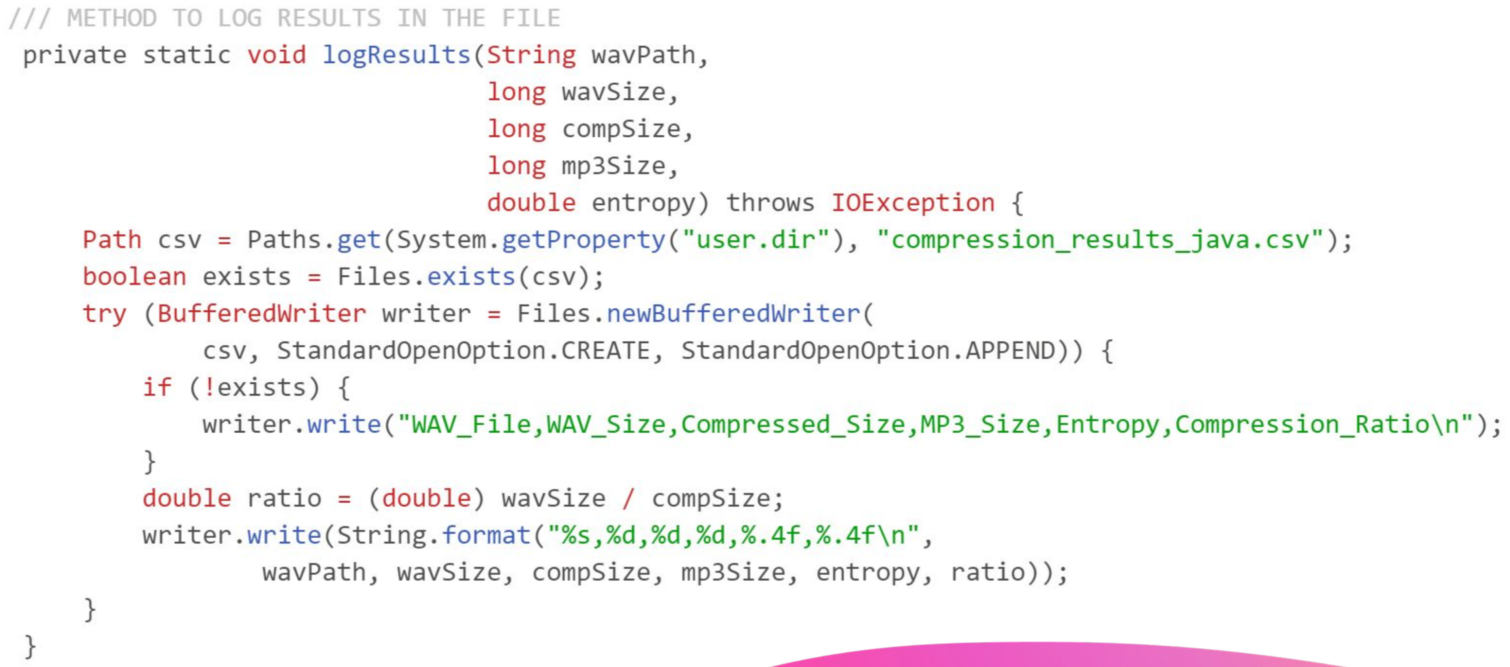
```
/// METHOD TO PERFORM LZW Compression
    public void compress() throws IOException {
        /// NEXT AVAILABLE CODE VALUE (After Init Dictionary)
        int nextCode = 128;

        /// INITIALIZE THE DICTIONARY WITH A SINGLE CHARACTER (0-127)
        for (int i = 0; i < 128; i++) {
            codeValue[i] = (short) i;
            prefixCode[i] = -1;
            appendCharacter[i] = (short) i;
        }

        /// SET REMAINING DICTIONARY ENTRIES TO -1
        for (int i = 128; i < TABLE_SIZE; i++) {
            codeValue[i] = -1;
        }
```

LZW Compression ratio Helper Functions

```
/// METHOD TO OUTPUT A CODE TO THE OUTPUT STREAM
private void outputCode(int code) throws IOException {
    /// ADD THE CODE TO THE OUTPUT BIT BUFFER
    outputBitBuffer = (outputBitBuffer << BITS) | code;
    outputBitCount += BITS;

    /// WRITE BYTES TO THE OUTPUT STREAM WHEN BUFFER IS FULL
    while (outputBitCount >= 8) {
        outputBitCount -= 8;
        output.write(outputBitBuffer >> outputBitCount);
    }
}
```

```
/// METHOD TO FIND A MATCH IN THE DICTIONARY FOR THE GIVEN PREFIX CODE
 private int findMatch(int prefixCodeVal, int character) {
     /// COMPUTE INIT INDEX USING HASHING
     int index = (character << HASHING_SHIFT) ^ prefixCodeVal;
     int offset = (index == 0) ? 1 : TABLE_SIZE - index;

     /// SEARCH FOR AN UNUESED ENTRY OR A MATCH
     while (codeValue[index] != -1) {
         if (prefixCodeVal == prefixCode[index] && character == appendCharacter[index]) {
             return index; /// RETURN INDEX MATCH
         }
         index -= offset; /// PROBE FOR NEXT INDEX
         if (index < 0) index += TABLE_SIZE; /// WRAP AROUND IF INDEX NEGATIVE
     }
     return index;
 }
```

Storing the Data into a CSV file for Output



```
/// METHOD TO LOG RESULTS IN THE FILE
 private static void logResults(String wavPath,
                                long wavSize,
                                long compSize,
                                long mp3Size,
                                double entropy) throws IOException {
     Path csv = Paths.get(System.getProperty("user.dir"), "compression_results_java.csv");
     boolean exists = Files.exists(csv);
     try (BufferedWriter writer = Files.newBufferedWriter(
             csv, StandardOpenOption.CREATE, StandardOpenOption.APPEND)) {
         if (!exists) {
             writer.write("WAV_File,WAV_Size,Compressed_Size,MP3_Size,Entropy,Compression_Ratio\n");
         }
         double ratio = (double) wavSize / compSize;
         writer.write(String.format("%s,%d,%d,%d,%.4f,%.4f\n",
                 wavPath, wavSize, compSize, mp3Size, entropy, ratio));
     }
 }
```

Results and Comparison

Type of Audio to Compress	WAV Original File Size	LZW Compressed File Size	MP3 File Size	Entropy
Sine Wave	5300302 bytes	6142172 bytes	722997 bytes	1
Cricket Noise	1769550 bytes	1545636 bytes	242135 bytes	0.9957
Repeating Word	5299466 bytes	4938666 bytes	481626 bytes	0.9993
Full Song	40665166 bytes	51194742 bytes	5534125 bytes	1

Following we will observe for each of our findings for the cases!

Case Analysis

Sine Wave

Analyzing a pure sine wave in 16-bit PCM (440Hz sine) still leads to almost no exact repeating bit-patterns, where its binary representation is “smooth” but not periodic at the bit level, so the Shannon entropy comes out near the theoretical maximum (1).

WAV	LZW	MP3
5300302	6142172	722997

Causing the LZW Compressed file to expand more than the WAV file. With having a Compression Ratio:

1 : 8

Cricket Noise

Analyzing the unique sounds of constant crickets noises in intervals (having long silent stretches or very repetitive chirps) caused a lower entropy then the previous case (0.9957), offering a modest improvement to give more information.

WAV	LZW	MP3
1769550	1545636	242135

In this case that LZW Compressed file was capable to compress over the WAV file, With having a Compression Ratio:

1 : 6

Compression Ratio = Original (MP3) Compression / LZW Compression

Case Analysis

Repeating Words

After seeing that the repetitiveness of the cricket noises caused lower entropy, we analyzed a audio of the famous internet debate: “Laurel” or “Yanny”, giving us more information on that the entropy was lower than the sine wave case (0.9993).

WAV	LZW	MP3
5299466	4938666	481626

Giving a slight lower LZW Compressed file compared to the WAV file. With having a Compression Ratio:

1 : 10

Complex Song

The final analysis consisted on testing the algorithm with a full-band song with vocals, drums, no loops (AREA - Site moi sliki). Which delivered exactly what we expected and wanted to see, with the no loops causing high entropy (1).

WAV	LZW	MP3
40665166	51194742	5534125

Where the no repetitiveness caused the LZW file to be worse than the WAV file. With having a Compression Ratio:

1 : 9

Compression Ratio = Original (MP3) Compression / LZW Compression

Conclusion

Why the results are like they are?

Investigating whether LZW compression outperforms MP3 on WAV audio, we hypothesized that LZW's lossless nature would exploit repetitions effectively.

However, ***the results disproved this***.

For repetitive patterns like sine waves and repeating words, LZW showed limited success than theorized due to its dictionary-based approach, and also struggling with continuous or highly variable data.

Whereas, the MP3, using perceptual coding, consistently achieved smaller file sizes by discarding inaudible data, regardless of repetition. For complex audio like the full song, LZW's size increase highlighted its inefficiency with high-entropy data, while MP3 thrived.

Thus, the hypothesis was disproved, affirming MP3's superiority in audio compression.

Thanks for your attention! :)

Github:
https://github.com/Nan0NJ/Waveform-Audio-Compression-LZW