LZW Compression of Video

TEAM S - Elsa Morina, Teodor Kostovski, Lucas Jakin

LZW - Lempel Ziv Welch

- **What is LZW?**

- LZW (Lempel-Ziv-Welch) compression is a lossless method for reducing the size of data files. It works by creating a dynamic dictionary of sequences from the input data, starting with single characters and adding longer sequences as it processes the file. This allows it to replace repetitive sequences with shorter codes, making it efficient for compressing data with repeated patterns.
- LZW is commonly used in formats like GIF images and some types of TIFF files. Because it doesn't lose any data during compression, it's ideal for situations where maintaining the original data is important, such as in text files and high-precision images. It's particularly effective for compressing text documents, simple graphics, and certain video frames that have a lot of repetition. The algorithm is straightforward and quick, requiring only a dictionary and basic sequence handling to operate.

How does it work?

- Initialization:
    - Create a dictionary with all possible single-character strings and their corresponding codes.
- Compression:
    - Read the input data character by character.
    - Concatenate characters to form the longest possible string that exists in the dictionary.
    - Output the code for this string.
    - Add a new entry to the dictionary: the current string concatenated with the next character.

      Repeat the process for the remaining input data.

Best case scenario vs worst case scenario

Best Case Scenario for LZW:

- LZW compression excels when data exhibits significant redundancy and minimal randomness, such as repeated sequences and patterns.
- It efficiently identifies and encodes these recurring patterns by maintaining a dictionary of frequently encountered sequences.
- For example, text files containing repetitive phrases or simple graphics with uniform areas and repeating patterns benefit immensely from LZW compression.
- In such instances, LZW's ability to recognize and compress repetitive patterns leads to significant compression ratios, helping in efficient storing and transmitting of data.

Worst Case Scenario for LZW:

- Conversely, LZW compression encounters challenges when confronted with high-entropy data—data with little repetition, large amounts of change and a lot of complexity
- In cases such as dynamic videos with different frames or encrypted files, the effectiveness of LZW diminishes significantly.
- The absence of repetitive sequences hinders LZW's ability to efficiently compress the data, resulting in minimal compression or even an increase in file size.
- For instance, attempting to compress a colorful animation with audio using LZW may yield unfavorable outcomes, as each frame and audio segment lacks the repetition necessary for effective compression.
- Consequently, the compressed file may exhibit little to no reduction in size or, in some cases, surpass the original size due to the overhead of storing dictionary entries.

How we did everything?

Step 1: Finding MKV Videos:

- Our first task was to find MKV video files suitable for our compression experiments. We managed to find various 5 to 10 second clips with different characteristics, including a black screen with no audio, animations with repetitive patterns and audio, and clips with non-repetitive patterns.

Step 2: Code Implementation (will be explained in more detail in the screenshots):

- We searched GitHub for an existing implementation of the LZW compression and then we made necessary modifications to make it fit to our specific requirements, such as handling bit string conversion and compression of MKV files.
- With the modified LZW code, we integrated it into our project, we made sure it could convert our MKV files correctly. We also wrote additional code to handle file input/output, byte array conversion, and other essential operations to facilitate the compression process.

Step 3: Testing and Results:

- After completing the code pipeline, we executed it on the different MKV files we obtained. The results varied depending on the characteristics of each video clip, with compression effectiveness reflecting the redundancy of patterns within the data. Then we also compared the compression results to the MP4 version of the same video which we converted it online by using this link: https://cloudconvert.com/mkv-to-mp4

Comparison between MKV & MP4

- **MKV (Matroska Video)**
  - MKV is an open standard format, which supports a wide variety of video/audio codecs.
  - Often used for **high-definition video** files due to its flexibility in storing multiple video tracks in a single file.
  - Known to support **lossless video compression**, which results in **larger files** but **better video quality** preservation.
  - Format is more resistant to corruption and in case of file errors, it is **easier to recover** playable content
  - MKV format may not be supported by video editing softwares.
  - MKV has slightly **less compatibility** with platforms compared to MP4.

- **MP4 (MPEG-4 Part 14)**
  - MP4 is a widely used and supported multimedia storage format for storing video & streaming.
  - Acts as a **wrapper around the video**, not the video itself.
  - MP4 supports progressive playback, allowing videos to be watched while being downloaded.
  - Associated with **efficient(high) compression**, leading to **smaller file sizes** → increased popularity.
  - More suitable for streaming and sharing on the internet.
  - Encoding & Decoding require **intensive ability.**
  - Although MP4 offers good quality at smaller sizes, its **compression** is **lossy**, resulting in **lower quality** compared to **lossless** formats(MKV).

Hypothesis: Due to the fact that MP4 utilizes lossy data compression, it will excel in compressing complex, high definition, high resolution videos, with fast-changing pixels and high quality audio. On the other hand, LZW compression, due to being Lossless, will excel in simple videos with minimal or no change in pixels, and simple or no audio. such as static image videos

Code explanation (1)

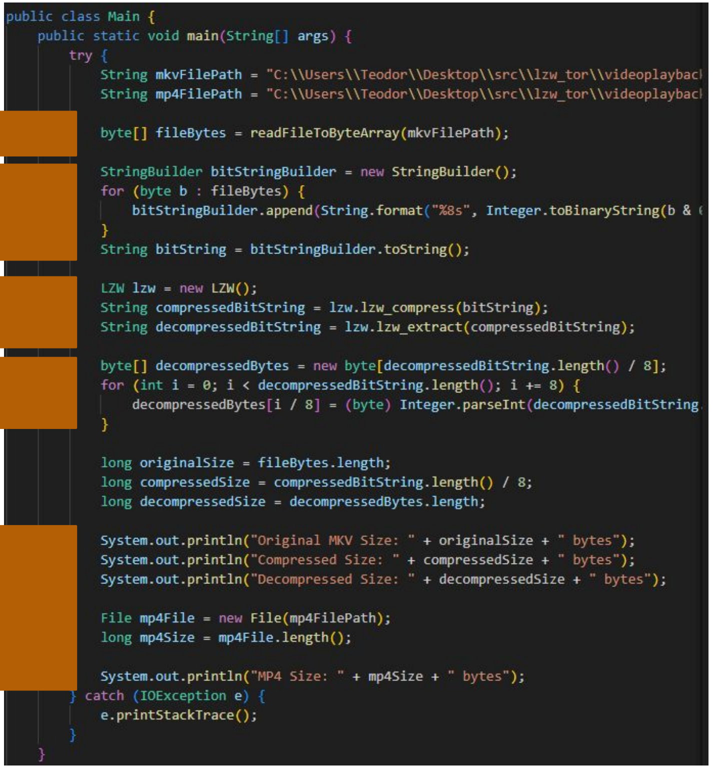
Function call for function at bottom

Reads bytes into binary strings (bits)

Function call to compress bit strings and decompress them

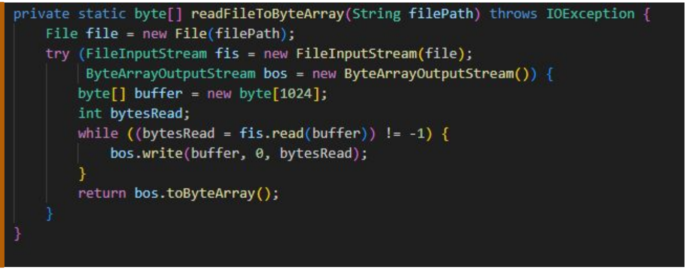
Decompression unused in testing results

originalSize = original MKV file, **CompressedSize** = bit array after LZW compression, **mp4Size** = file after MP4 compression



```java
public class Main {
    public static void main(String[] args) {
        try {
            String mkvFilePath = "C:\\Users\\Teodor\\Desktop\\src\\lzw_tor\\videoplaybac
            String mp4FilePath = "C:\\Users\\Teodor\\Desktop\\src\\lzw_tor\\videoplaybac

            byte[] fileBytes = readFileToByteArray(mkvFilePath);

            StringBuilder bitStringBuilder = new StringBuilder();
            for (byte b : fileBytes) {
                bitStringBuilder.append(String.format("%8s", Integer.toBinaryString(b &
            }
            String bitString = bitStringBuilder.toString();

            LZW lzw = new LZW();
            String compressedBitString = lzw.lzw_compress(bitString);
            String decompressedBitString = lzw.lzw_extract(compressedBitString);

            byte[] decompressedBytes = new byte[decompressedBitString.length() / 8];
            for (int i = 0; i < decompressedBitString.length(); i += 8) {
                decompressedBytes[i / 8] = (byte) Integer.parseInt(decompressedBitString
            }

            long originalSize = fileBytes.length;
            long compressedSize = compressedBitString.length() / 8;
            long decompressedSize = decompressedBytes.length;

            System.out.println("Original MKV Size: " + originalSize + " bytes");
            System.out.println("Compressed Size: " + compressedSize + " bytes");
            System.out.println("Decompressed Size: " + decompressedSize + " bytes");

            File mp4File = new File(mp4FilePath);
            long mp4Size = mp4File.length();

            System.out.println("MP4 Size: " + mp4Size + " bytes");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Firstly reads file as an input stream, then the input stream is converted as a byte array to be used in the above call



```java
private static byte[] readFileToByteArray(String filePath) throws IOException {
    File file = new File(filePath);
    try (FileInputStream fis = new FileInputStream(file);
         ByteArrayOutputStream bos = new ByteArrayOutputStream()) {
        byte[] buffer = new byte[1024];
        int bytesRead;
        while ((bytesRead = fis.read(buffer)) != -1) {
            bos.write(buffer, 0, bytesRead);
        }
        return bos.toByteArray();
    }
}
```

Code explanation (2)

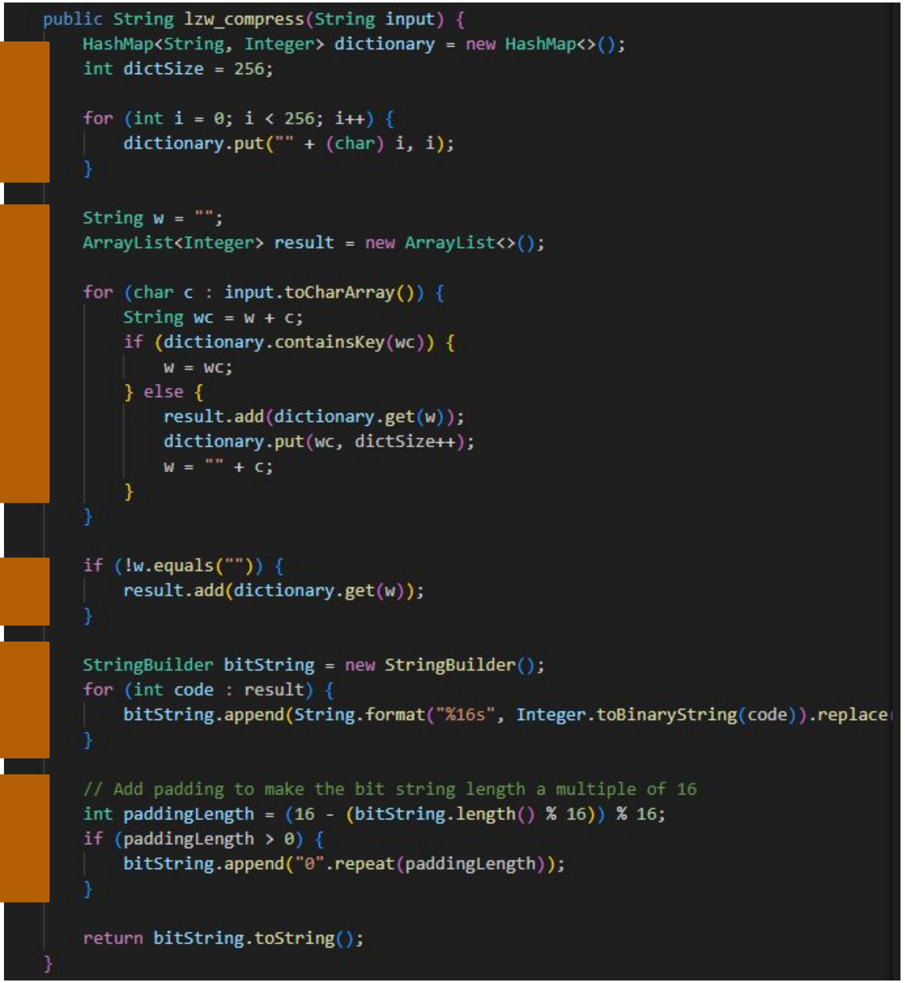
Initial dictionary initialization, where the initial dictionary is 256, for all possible values of a byte. The dictionary maps a specific value of a byte to a value i

w starts off as an empty string, a (b)single character c is read from our input. wc is the concatenation of w and c, if wc is already contained in the dictionary, wc is saved as w, so then another character can be appended (ex. W = 10, c1 = 1, but wc = 101 already exists, so w = 101, and c2 = 1, so wc = 1011). If wc does not exist, w is added to the resulting array sequence, and wc is added to the dictionary, w becomes c

Once loop ends, if w is not empty, add the final part of the string to results

Result is read as binary string of type int (bits)

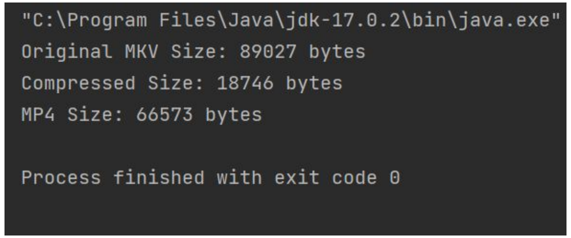
Added padding for sake of decoding, results aren't affected much, only by a few bits



```java
public String lzw_compress(String input) {
    HashMap<String, Integer> dictionary = new HashMap<>();
    int dictSize = 256;

    for (int i = 0; i < 256; i++) {
        dictionary.put("" + (char) i, i);
    }

    String w = "";
    ArrayList<Integer> result = new ArrayList<>();

    for (char c : input.toCharArray()) {
        String wc = w + c;
        if (dictionary.containsKey(wc)) {
            w = wc;
        } else {
            result.add(dictionary.get(w));
            dictionary.put(wc, dictSize++);
            w = "" + c;
        }
    }

    if (!w.equals("")) {
        result.add(dictionary.get(w));
    }

    StringBuilder bitString = new StringBuilder();
    for (int code : result) {
        bitString.append(String.format("%16s", Integer.toBinaryString(code)).replace
    }

    // Add padding to make the bit string length a multiple of 16
    int paddingLength = (16 - (bitString.length() % 16)) % 16;
    if (paddingLength > 0) {
        bitString.append("0".repeat(paddingLength));
    }

    return bitString.toString();
}
```

Explanation of the Results

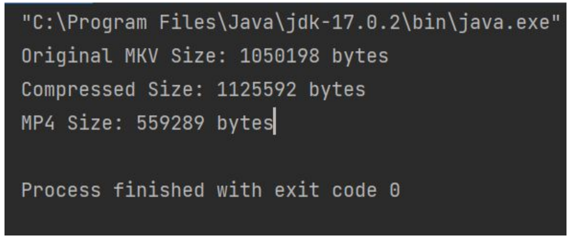
5 sec black screen video with no audio:



```
"C:\Program Files\Java\jdk-17.0.2\bin\java.exe"
Original MKV Size: 89027 bytes
Compressed Size: 18746 bytes
MP4 Size: 66573 bytes

Process finished with exit code 0
```

Here we tested an audioless black screen video and we see that the compression is really efficient when using LZW, even compared to the MP4 version of it, Due to the fact that it is only a black screen, all pixels are encoded as the same value, as such there is minimal to no randomness.
Compression rate (MKV -> LZW): 4.75:1
Compression rate (MKV -> MP4): 1.33:1 (Possibly due to how MP4 encodes overhead, and possible faults with online converter.)

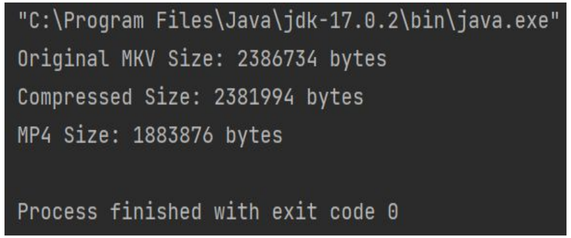
5 sec video with audio of the bunny animation:



```
"C:\Program Files\Java\jdk-17.0.2\bin\java.exe"
Original MKV Size: 1050198 bytes
Compressed Size: 1125592 bytes
MP4 Size: 559289 bytes

Process finished with exit code 0
```

Here we tested a bunny animation that lasts 5 seconds and it also contains audio in it. The results show that the compression was actually worse. This could be due to the fact that the original MKV was already compressed in a sophisticated manner, due to the randomness and complexity of the audio and video, or due to some large overhead data that the algorithm could not encode.
Compression rate (MKV -> LZW): 0.93:1
Compression rate (MKV -> MP4): 1.877:1 (Possibly due to the fact that the MP4 converter used lossy compression, which could handle the random data better**)**

10 sec video without audio of rainbow:



```
"C:\Program Files\Java\jdk-17.0.2\bin\java.exe"
Original MKV Size: 2386734 bytes
Compressed Size: 2381994 bytes
MP4 Size: 1883876 bytes

Process finished with exit code 0
```

Here we tested an audioless colorful/rainbow screen and we see that the compression is inefficient when using LZW, it is almost the same size as the file itself, compared to the MP4 compression which does a better job at compressing random patterns.
Compression rate (MKV -> LZW): 1.001:1
Compression rate (MKV -> MP4): 1.2669:1 (Again, possibly due to the fact that the fact that the MP4 converter used lossy compression **)**

Conclusion/overview

- The LZW algorithm, while effective for certain type of data such as text and images and simple videos, is not well-suited for compressing large multimedia content.

- As seen from the results, we proved that our hypothesis (from slide 5) is correct and that data with minimal variability, such as a simple video of a black screen, works best with LZW compression.