LZW coding (compression)

LZW (Lempel-Ziv-Welch) is a lossless data compression algorithm that builds a dictionary of substrings during encoding. It compresses data by replacing repeated occurrences of data (patterns) with shorter codes that represent those patterns. The basic idea is to replace substrings with shorter codes as the compression progresses, so the output size becomes smaller.

Key Steps in LZW Compression

1. **Initialize a dictionary** with all single characters in the input data (e.g., all ASCII characters if the input is text).

2. **Scan the input** string character by character, building longer and longer substrings that exist in the dictionary.

3. **Add new substrings** to the dictionary and output the dictionary index for the current substring when it can no longer be extended by the next character in the input.

Example of LZW Compression

Let’s take the input string:

```
ABABABA
```

We will step through the LZW compression process.

Step 1: Initialize the Dictionary

At the beginning, the dictionary contains all single characters from the input string.

For this example, let’s assume we use indices 0–255 for the ASCII table, so initially, our dictionary contains:

Index	Character
65	A
66	B

Step 2: Start Encoding

We start reading the input string character by character and follow these steps:

1. **Current substring**: `“A”`

    1. It exists in the dictionary (index 65).

2. Read the next character: `“B”`.
3. The new substring `“AB”` does not exist in the dictionary.
4. **Output** the code for `“A”` (65) and add `“AB”` to the dictionary with the next available index (256).

2. **Current substring**: `“B”`

1. It exists in the dictionary (index 66).
2. Read the next character: `“A”`.
3. The new substring `“BA”` does not exist in the dictionary.
4. **Output** the code for `“B”` (66) and add `“BA”` to the dictionary with the next available index (257).

3. **Current substring**: `“A”`

1. It exists in the dictionary (index 65).
2. Read the next character: `“B”`.
3. The substring `“AB”` **already exists** in the dictionary (index 256).
4. Read the next character: `“A”`, forming `“ABA”`.
5. The new substring `“ABA”` does not exist in the dictionary.
6. **Output** the code for `“AB”` (256) and add `“ABA”` to the dictionary with the next available index (258).

4. **Current substring**: `“A”`

1. It exists in the dictionary (index 65).
2. Read the next character: `“B”`.
3. The substring `“AB”` **already exists** in the dictionary (index 256).
4. Read the next character: `“A”`, forming `“ABA”`.
5. No more characters left to read.
6. **Output** the code for `“ABA”` (258).

Final Dictionary After Compression

The final dictionary looks like this:

Index	Character
65	A
66	B
256	AB
257	BA
258	ABA

Encoded Output

The final output (in terms of dictionary indices) is:

```
65 66 256 258
```

This represents the compressed version of the original string `“ABABABA”`.

—

Decoding LZW

To decode an LZW-compressed string, you use the same dictionary-based approach but in reverse. The decoder reconstructs the dictionary entries as it processes the encoded output, using the indices to retrieve the corresponding substrings.

Let’s go through the decoding process of the encoded sequence `65 66 256 65`:

1.) **65**: Look up in the dictionary. It maps to `“A”`.

- **Decoded string so far**: `“A”`

2.) **66**: Look up in the dictionary. It maps to `“B”`.

- **Decoded string so far**: `“AB”`

3.) **256**: Look up in the dictionary. It maps to `“AB”`.

- **Decoded string so far**: `“ABAB”`

4.) **258**: Look up in the dictionary. It maps to `“ABA”`.

- **Decoded string so far**: `“ABABABA”`

The original string ABABABA is reconstructed from the compressed codes.

—

Summary of LZW Compression

1. **Dictionary Initialization**: The dictionary starts with all individual characters.

2. **Scanning Input**: Input data is scanned character by character, grouping together characters to form substrings.

3. **Dictionary Expansion**: New substrings are added to the dictionary as the input is processed.

4. **Output**: The dictionary indices corresponding to the substrings are output, resulting in a compressed sequence of codes.

LZW is widely used in formats like **GIF** images and **TIFF** files for lossless compression. It is particularly effective for data that contains repeated patterns or symbols.

C implementation of LZW encoding:

```
#include <stdio.h>
#include <string.h>

#define MAX_sxhS 100

char sxhTable[MAX_sxhS][10] = {"a","b","c","d","e"};

int tableElements = 5;

main()
{
    char text[] = "dabbacdabbacdabbacdabbacdee";
    char *p = text;
    int bufferEnd = 1;

    int lastFoundIndex = -1;
    while(*p != '\0')
    {
        char subStr[10];
        strncpy(subStr, p, bufferEnd);
        subStr[bufferEnd] = '\0';
        int foundIndex = isInsxhTable(subStr);
        if(foundIndex != -1)
        {
           bufferEnd++;
           lastFoundIndex = foundIndex;
           continue;
        }
        p += strlen(subStr) - 1;
        bufferEnd = 1;
        strcpy(sxhTable[tableElements++], subStr);
        printf("%d,", lastFoundIndex + 1);
    }
}

int isInsxhTable(char *s)
{
    for(int i = 0; i < tableElements; i++)
    {
        if(strcmp(sxhTable[i], s) == 0)
        {
            return i;
        }
    }
    return -1;
}
```