Lab 7

Author: Lukas Kmitas

1. Linux compression commands:

Here is the compression commands demoed in the class. Please find a linux terminal to practice the commands using different files.

Following links are recommended: <https://explainshell.com/> and https://tldr.sh/

7za a tensorflow-master

tar -zcf tensor.tar.gz tensorflow-master

tar -jcf tensor.tar.bz2 tensorflow-master

tar -Zcf tensor.tar.Z tensorflow-master

7za x tensorflow-master.7z

tar -Zxf tensor.tar.Z

tar -jxf tensor.tar.bz2

tar -zxf tensor.tar.gz

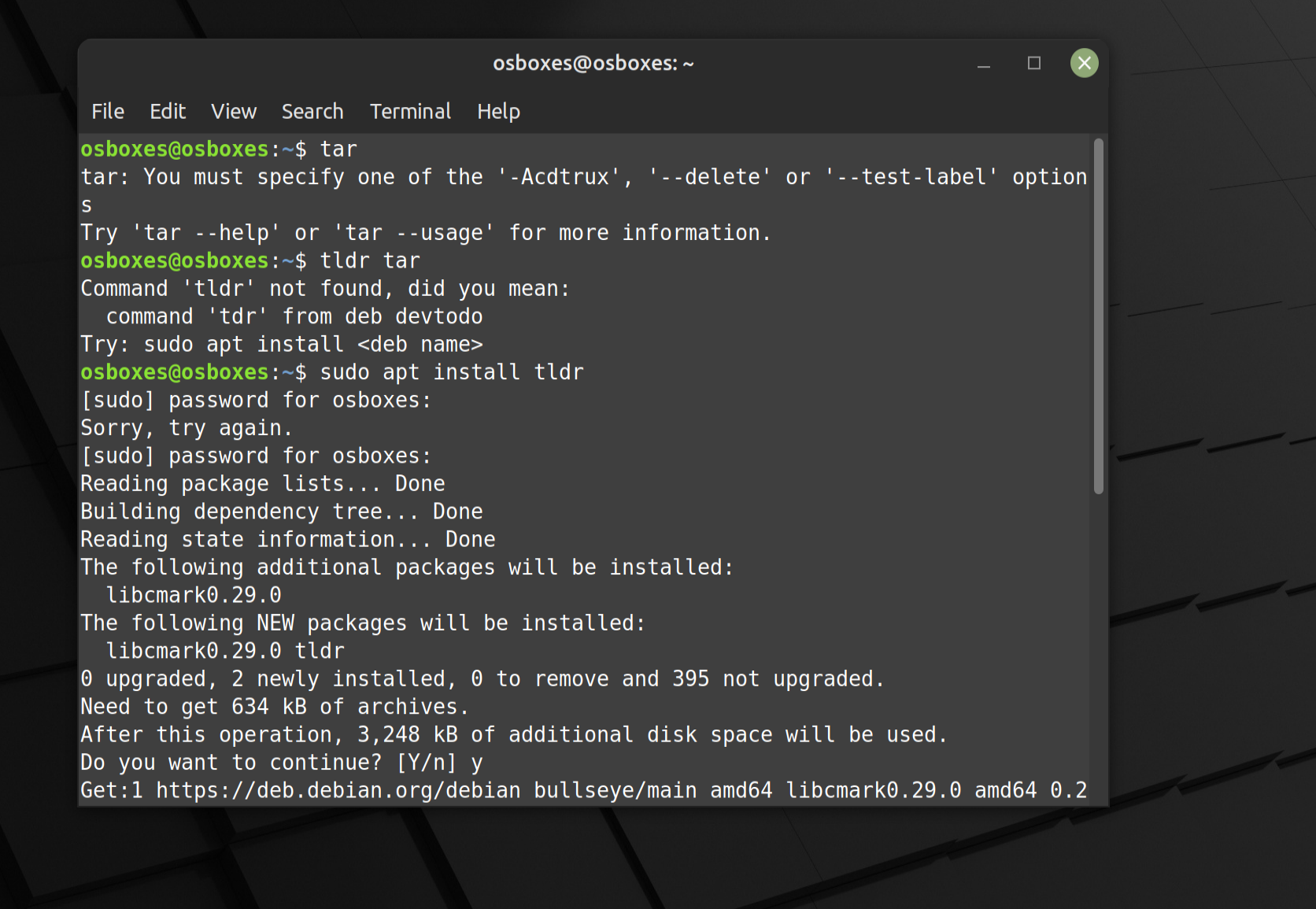
1. One method of reducing bandwidth use is to compress the data being transmitted. Let A = {a/20, b/15, c/5, d/15, e/45} be the alphabet and its frequency distribution. Compute the optimal coding for each character. What is the average number of bits/symbol of the codes?
2. Please draw the information exchange flow chart according to the diagram for delta compression.

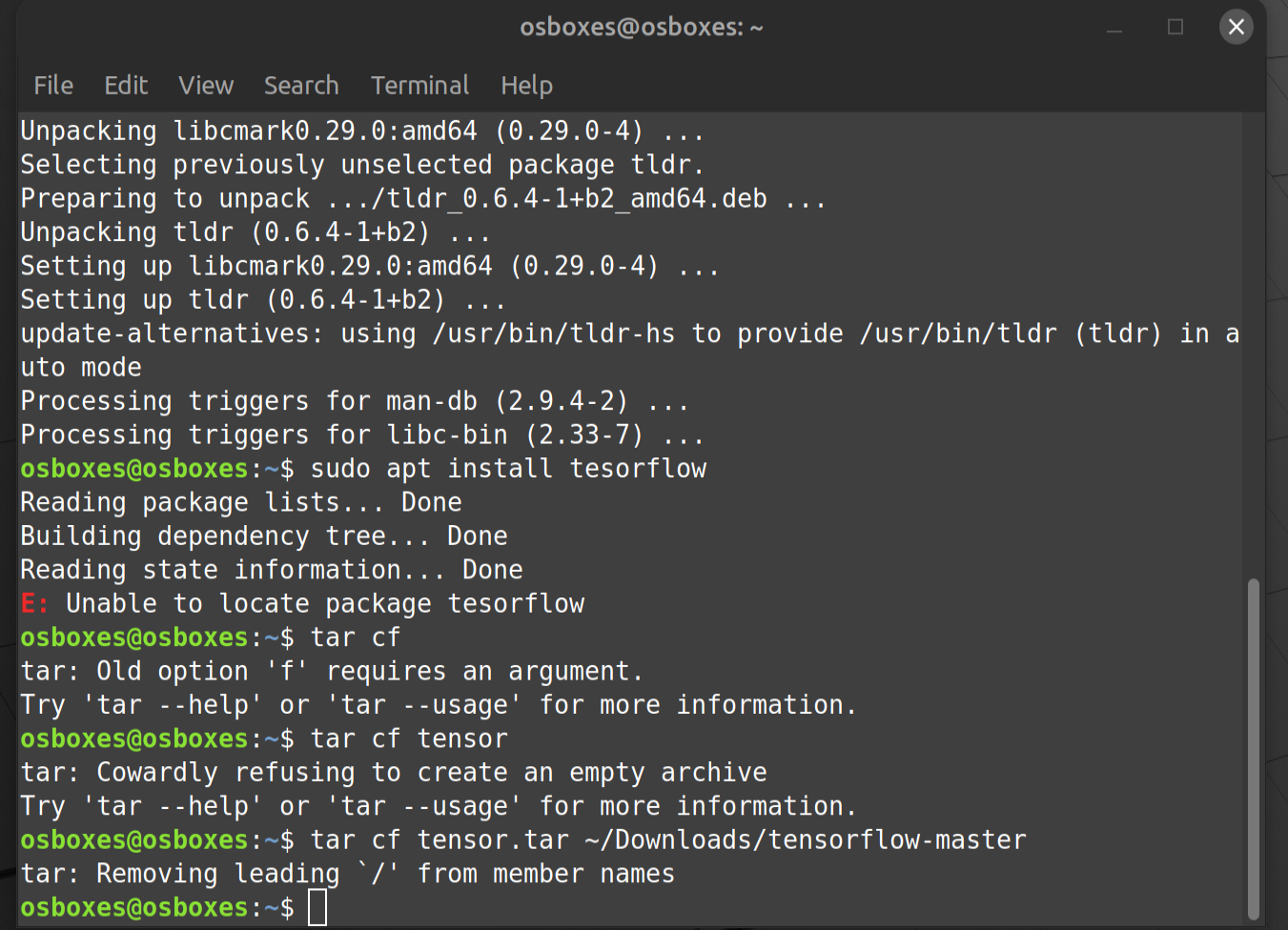
Chart

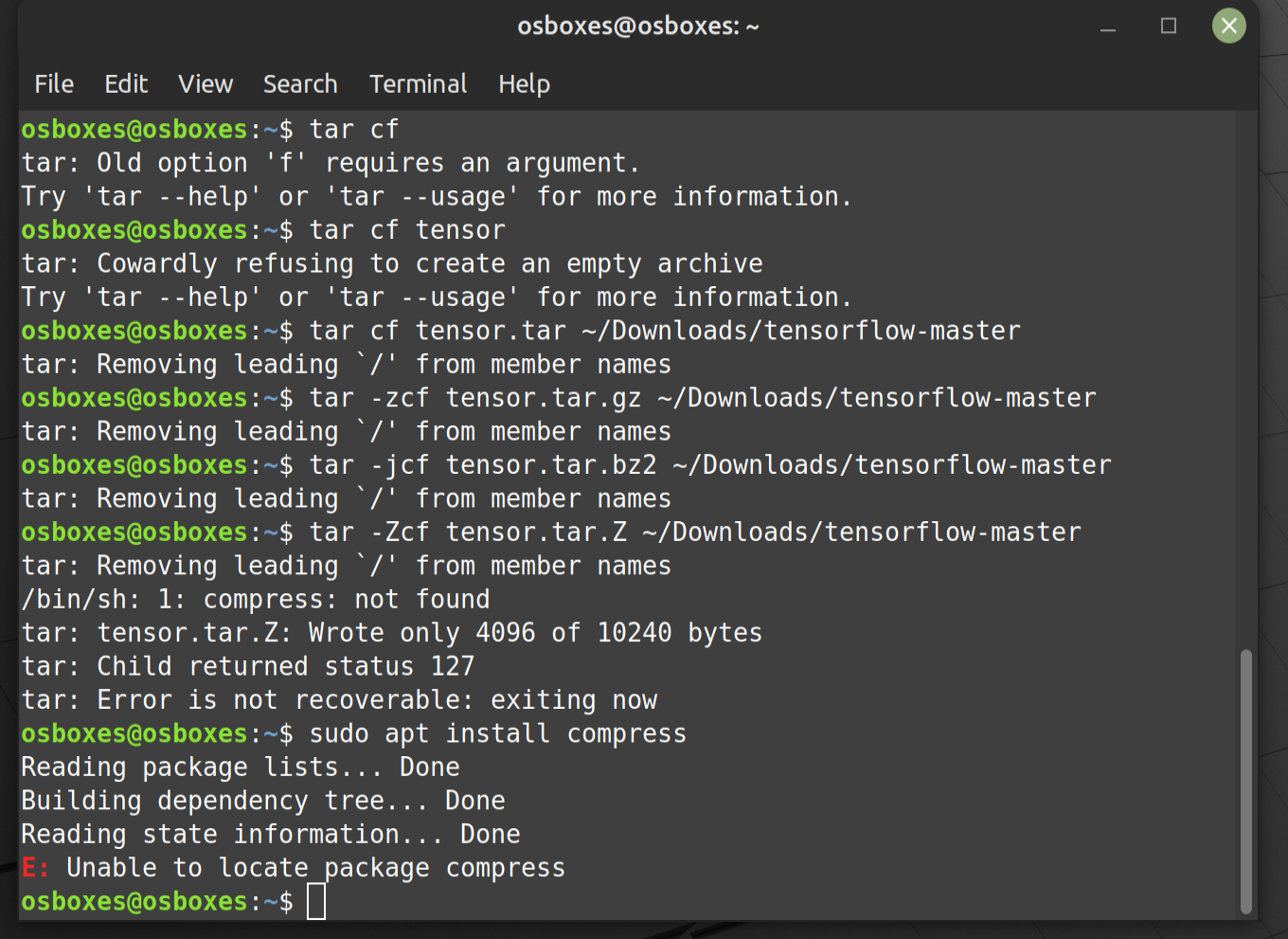
Description automatically generated with medium confidence

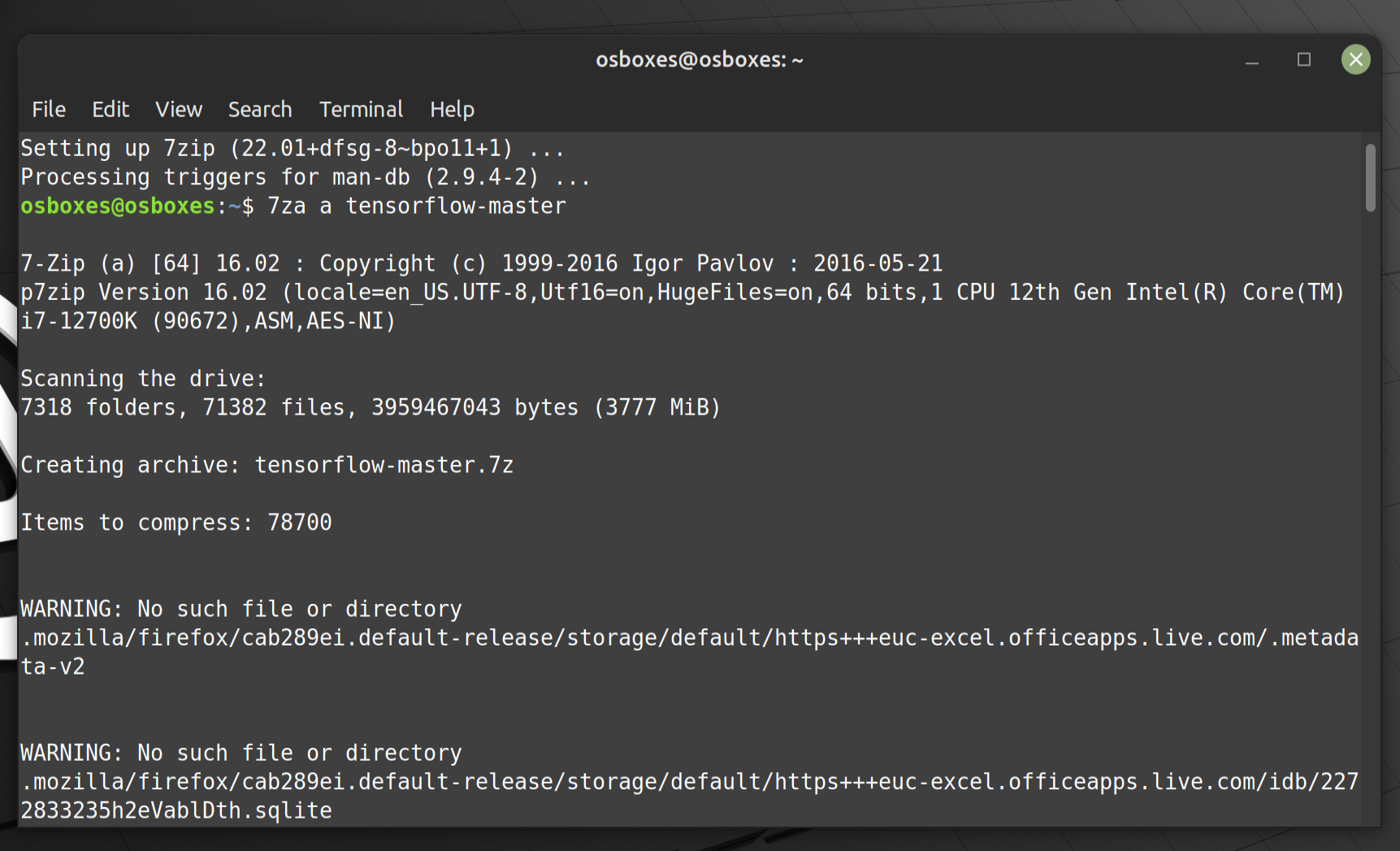
1. One method of reducing bandwidth use is to compress the data being transmitted. Use the LZW algorithm to compress the string: BABAABAAA. Note that Uppercase A has ASCII value 65 in decimal. Draw diagrams to aid your explanation if appropriate.

Q1) You have seen me doing this in class but I do have some screenshots I took.









**Q2)**

To compute the optimal coding for each character in the given alphabet A and its frequency distribution, we can use Huffman coding, a common method of lossless data compression. Huffman coding builds a binary tree where each character is assigned a unique binary code based on the frequency of occurrence, with more common characters having shorter codes.

Given A = {a/20, b/15, c/5, d/15, e/45},

computing the Huffman codes for each character:

1. Sort the characters by frequency:

c: 5

a: 20

b: 15

d: 15

e: 45

1. Build the Huffman tree by combining the nodes with the smallest frequencies and building up until there is a single tree covering all characters: Combine nodes with smallest frequencies and repeat until there is only one node.
2. Assign codes to each character by traversing the tree, with left edges representing a '0' and right edges representing a '1'.
3. Compute the average number of bits/symbols based on the assigned codes and the frequencies of each character.

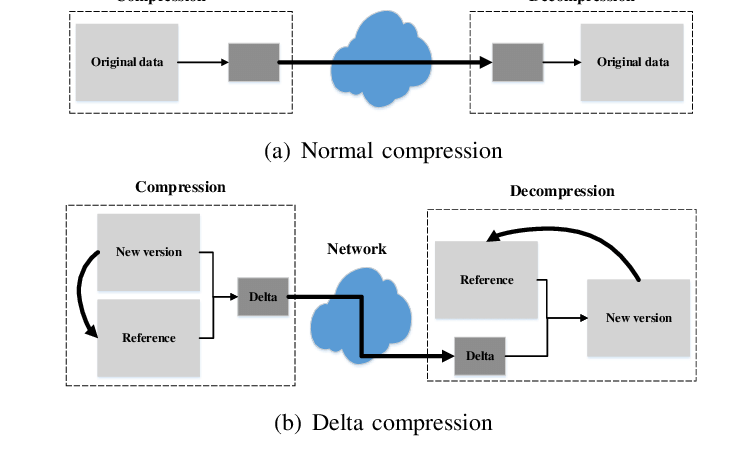
Let's perform these steps to determine the Huffman codes for each character and calculate the average number of bits per symbol.

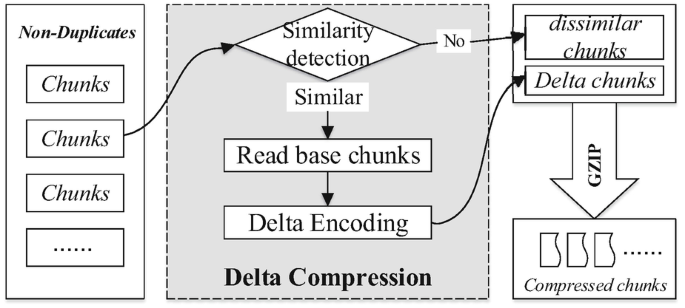
The optimal Huffman codes for each character in the alphabet A, given its frequency distribution, are as follows:

* e: 0
* a: 111
* b: 101
* c: 100
* d: 110

The average number of bits per symbol of these codes is 2.1. This means, on average, each symbol in the given alphabet can be represented using 2.1 bits, which illustrates the efficiency of Huffman coding in reducing bandwidth use by compressing data.

**Q3)**





**Q4)**

The LZW (Lempel-Ziv-Welch) algorithm is a lossless data compression algorithm that works by reading a sequence of symbols, grouping them into strings, and converting the strings into codes. It's based on the idea that repeated sequences can be represented by a single code, thus saving space. It creates a dictionary of these sequences dynamically while encoding the data.

compressing the string "BABAABAAA" using the LZW algorithm.

1. Initialize the dictionary with single characters and their ASCII values. Since we're dealing with uppercase letters, and we know that “A” has an ASCII value of 65, we can infer that “B” has an ASCII value of 66.

Dictionary looks like this:

Dictionary: {A: 65, B: 66}

Now read the string "BABAABAAA" one symbol at a time and add new sequences to the dictionary, assigning them the next available code.

1. Begin by reading “B”, which is already in our dictionary with a code of 66. Next, read 'A', also in the dictionary with a code of 65. Combine 'B' and 'A' to check if 'BA' is in the dictionary:

'BA' is not in the dictionary, so we add it with the next available code and output the code for 'B'.

Dictionary: {A: 65, B: 66, BA: 67}

Output: 66

1. Now the previous character was “A” we read the next character, “B”, forming “AB” “AB” is not in the dictionary, so add it with the next code (68) and output the code for 'A'.

Dictionary: {A: 65, B: 66, BA: 67, AB: 68}

Output: 66, 65

1. The previous sequence 'AB' was added to the dictionary. The next character is 'A', forming 'BA' again. But now 'BA' is in our dictionary with a code of 67. Read the next character, 'A', forming 'BAA'.

'BAA' is not in the dictionary, so add it (code 69) and output the code for 'BA'.

Dictionary: {A: 65, B: 66, BA: 67, AB: 68, BAA: 69}

Output: 66, 65, 67

1. The previous character was 'A'. The next character is also 'A', which is already in the dictionary. Read the next character, 'A', so we have 'AA'.   
   'AA' is not in the dictionary, add it (code 70) and output the code for 'A'.

Dictionary: {A: 65, B: 66, BA: 67, AB: 68, BAA: 69, AA: 70}

Output: 66, 65, 67, 65

1. 'AAA' is not in the dictionary, add it (code 71) and output the code for 'AA'.

Dictionary: {A: 65, B: 66, BA: 67, AB: 68, BAA: 69, AA: 70, AAA: 71}

Output: 66, 65, 67, 65, 70

Now we've reached the end of the string. The last sequence 'AAA' is already in the dictionary, so output its code.

Output: 66, 65, 67, 65, 70, 71

This output shows that we've reduced the original 9-character string "BABAABAAA" to a 6-number sequence. Each number corresponds to a dictionary entry that the decompression algorithm can use to reconstruct the original string. This example illustrates how the LZW algorithm saves bandwidth by compressing the data.