Project 3

Adaptive Arithmetic Encoding and Decoding

Lempel-Ziv Encoding and Decoding

Omar Tarek Ahmed 1180004

Hussein Ahmed Ibrahim 1180594



Contents

[List of Figures 1](#_Toc103910277)

[User Manual 3](#_Toc103910278)

[Encoding 3](#_Toc103910279)

[Decoding 4](#_Toc103910280)

[Adaptive Arithmetic coding (Heading 1) 6](#_Toc103910281)

[Use headings (heading 2) 6](#_Toc103910282)

[From the top left corner ya s7s (heading 3) 6](#_Toc103910283)

[Lempel-Ziv 7](#_Toc103910284)

[Encoding 7](#_Toc103910285)

[Process 8](#_Toc103910286)

[Getting the unique sequences 8](#_Toc103910287)

[Getting the Index/Suffix Pairs 9](#_Toc103910288)

[Sequence Enumeration 10](#_Toc103910289)

[Get the number of bits per sequence 10](#_Toc103910290)

[Turning the pairs into bits 10](#_Toc103910291)

[Calculating the compression ratio 11](#_Toc103910292)

[Decoding 12](#_Toc103910293)

[Calculating the number of bits for each sequence 13](#_Toc103910294)

[Decoding the codeword 13](#_Toc103910295)

[Get the Prefix/Suffix Pairs 13](#_Toc103910296)

[From Bits to text 14](#_Toc103910297)

[Results 14](#_Toc103910298)

[Adaptive Arithmetic 14](#_Toc103910299)

[Lempel-Ziv 14](#_Toc103910300)

[Test Case #1 14](#_Toc103910301)

[Test Case #2 16](#_Toc103910302)

[Test Case #3 17](#_Toc103910303)

# List of Figures

[Figure 1 : Main Screen 3](#_Toc103910304)

[Figure 2 : Encoding Output Example 4](#_Toc103910305)

[Figure 3 : Decoding Tab | Lempel-Ziv 4](#_Toc103910306)

[Figure 4 : Decoding Tab | Adaptive Arithmetic 5](#_Toc103910307)

[Figure 5 : Decoder Output 5](#_Toc103910308)

[Figure 6 : Lempel- Ziv Encoding Function 7](#_Toc103910309)

[Figure 7 : Getting The Unique Sequences 8](#_Toc103910310)

[Figure 8 : Getting the (Index, Suffix) Pairs 9](#_Toc103910311)

[Figure 9 : Turning The Pairs Matrix Into An Array Of Numbers 10](#_Toc103910312)

[Figure 10 : Getting The Number Of Bits That Each Sequence Will Take 10](#_Toc103910313)

[Figure 11 : Encoding To Bits 10](#_Toc103910314)

[Figure 12 : Compression Ratio Code 11](#_Toc103910315)

[Figure 13 : Main Decoder Function | Lempel-Ziv 12](#_Toc103910316)

[Figure 14 : Dividing The Encoded Bit Stream Into Sequences of Variable Lengths 13](#_Toc103910317)

[Figure 15 : Retrieving The (Prefix, Suffix) Pairs 13](#_Toc103910318)

[Figure 16 : Test Case 1 | Lempel-Ziv 14](#_Toc103910319)

[Figure 17 :Test Case #2 | Lempel-Ziv 16](#_Toc103910320)

[Figure 18 : Test Case #3 | Lempel-Ziv 17](#_Toc103910321)

# User Manual

## Encoding

When you open the program, you are greeted with a screen that looks like this:

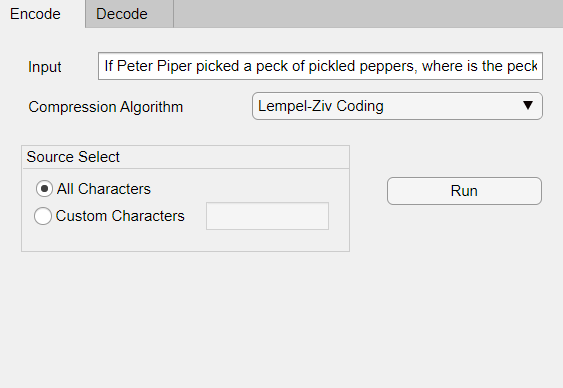


Figure 1 : Main Screen

Here we have *two* tabs, one for encoding, and the other for encoding. You can input the text to be encoded in the *input* edit field, and choose the compression algorithm from its respected *combo-box.* The famous tongue twister is used as the default input for convenience of testing.

We also have the ability to choose between custom characters, and *all characters*. Should the user choose *all characters,* the following characters are selected:

charList = “abcdefghijklmnopqrstuvwxyz,? ”

Upon pressing run, this pop-up appears:

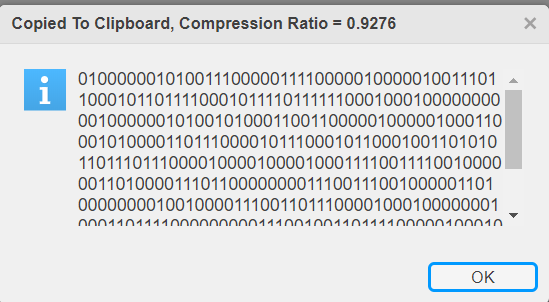


Figure 2 : Encoding Output Example

The pop-up shows the compression result and copies it to the clipboard. Notice that the compression ratio is shown in the title of the pop up.

## Decoding

We use a similar UI for decoding:

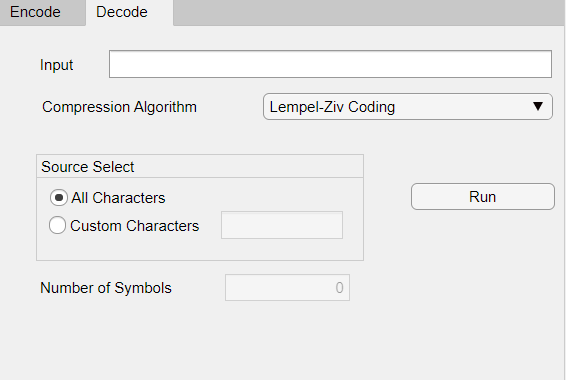


Figure 3 : Decoding Tab | Lempel-Ziv

The user can enter the bits required to be decoded in the input edit field, and can choose the compression algorithm used in the encoding of said bits. The *number of symbols* field is used only when the encoding algorithm is the *adaptive arithmetic technique* and as a result, has been disabled in when the *Lempel-Ziv* *technique* is selected.

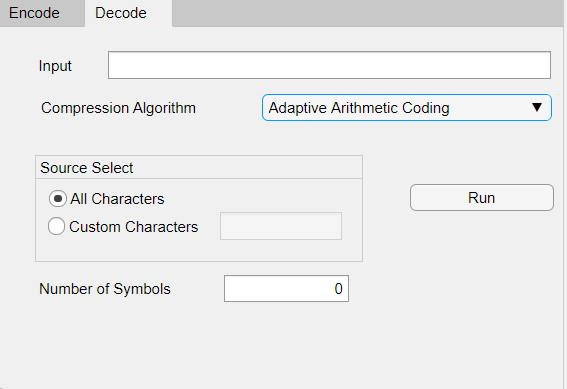


Figure 4 : Decoding Tab | Adaptive Arithmetic

After using the encoder, paste the output in the *input field* in the decoder tab then press *Run*. This should output the same text entered in the encoder *input field.*

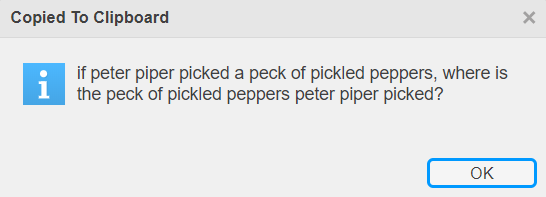


Figure 5 : Decoder Output

# Adaptive Arithmetic coding (Heading 1)

## Use headings (heading 2)

### From the top left corner ya s7s (heading 3)

Then go to table of contents and press update table this automatically update the table

Good luck

# Lempel-Ziv

## Encoding

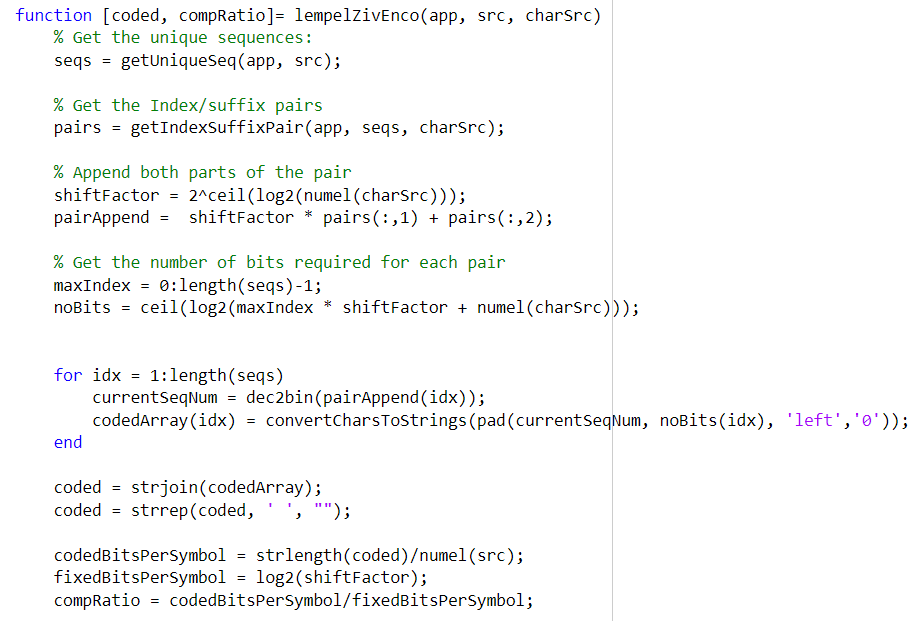


Figure 6 : Lempel- Ziv Encoding Function

This function takes as inputs the text to encode, and the possible source characters, and outputs the encoded text and the compression ratio.

## Process

### Getting the unique sequences

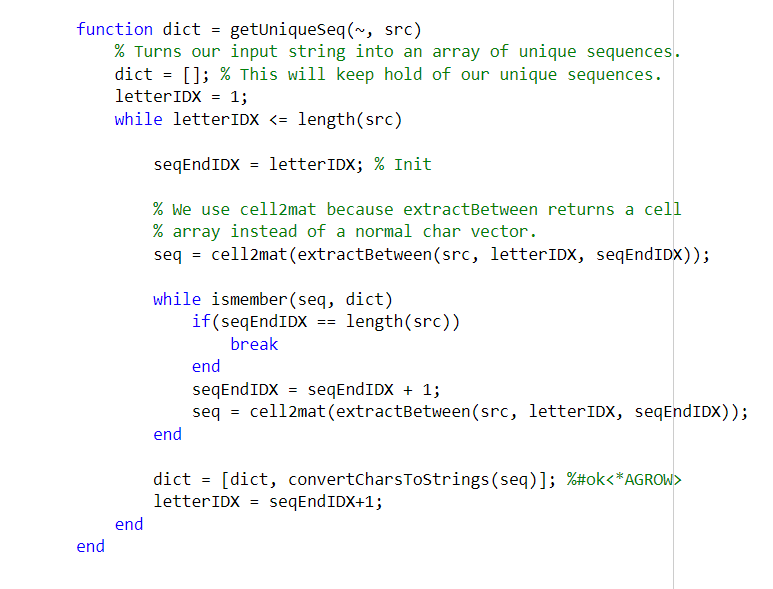
We need to divide out text into as much unique segments as we can. The function in the figure below does just that.

Figure 7 : Getting The Unique Sequences

We loop on each letter in the text, and we check all the possible sequences it can make. We check if these sequences are already in our *sequence dictionary*, if they are members of our dictionary we check if the next sequence is a member or not until we find a sequence that doesn’t belong to our dictionary yet. We add this symbol to our dictionary and we keep looping on the remaining letters of the text.

### Getting the Index/Suffix Pairs

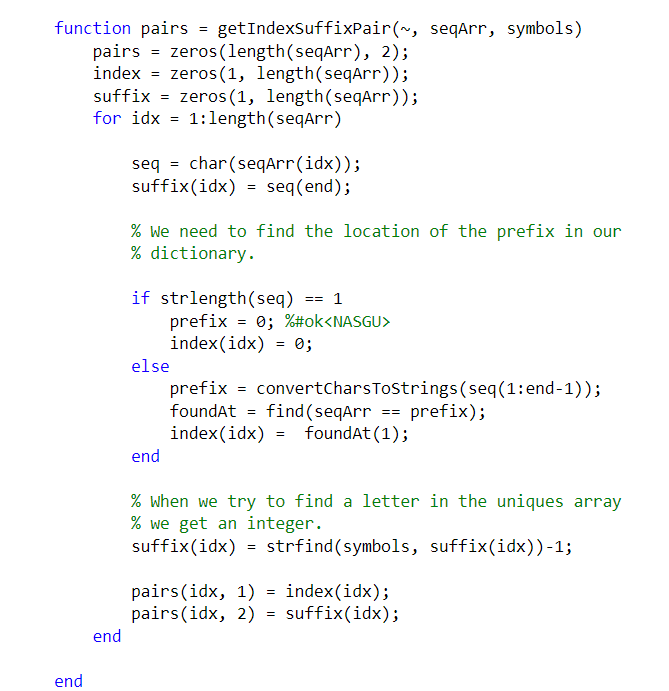


Figure 8 : Getting the (Index, Suffix) Pairs

This function gets the (Index, Suffix) pairs by finding the index of the prefix for each sequence, then it enumerates the suffix letter, and puts the result in a *n x 2* matrix, where n is the number of sequences, and the first column is for the indices and the second for suffixes. Keep in mind that we do not convert the pairs to binary. While the conversion to binary makes things easier for a human performing the Lempel-Ziv technique by hand, on paper, it is an overhead to convert the pairs to binary on a computer program.

### Sequence Enumeration

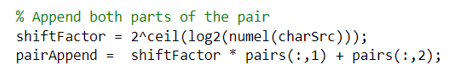


Figure 9 : Turning The Pairs Matrix Into An Array Of Numbers

For each row of the pairs matrix, we need to append both its columns together in a bitwise fashion. This means we need to *logical shift left* the indices by the number of bits it takes to represent the source characters (let’s call it shiftFactor), and then add the suffix number. This eliminates the need for binary conversion.

### Get the number of bits per sequence

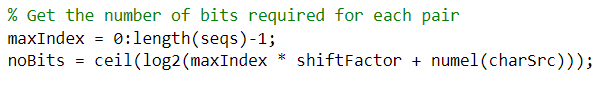


Figure 10 : Getting The Number Of Bits That Each Sequence Will Take

We know the maximum index for any sequence is the index of the sequence before it. So we shift this maximum index left by the number of bits it takes to represent the source characters (by multiplying it by shiftFactor) then we add the maximum possible suffix value, which is the number of characters available. This is because we number the characters starting from 1, up to their length.

### Turning the pairs into bits

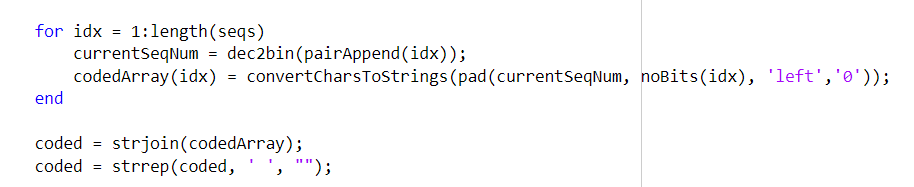


Figure 11 : Encoding To Bits

We loop on each sequence, turn its appended pair into a binary number, and then pad this binary number with the required number of zeros calculated in the previous step.

### Calculating the compression ratio

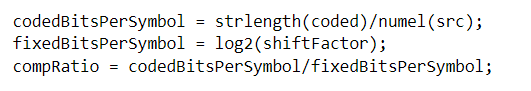
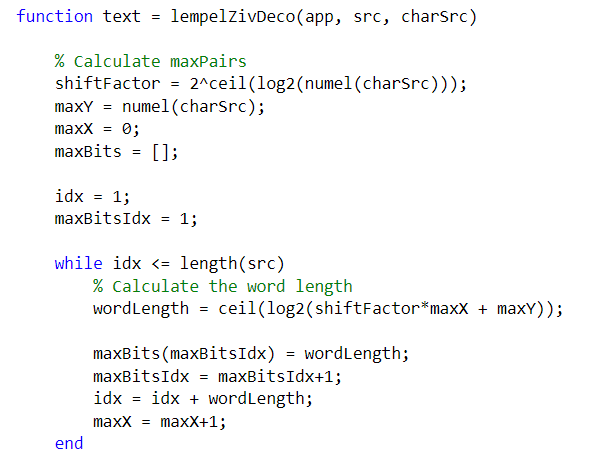


Figure 12 : Compression Ratio Code

We simply divide the number of bits by the number of encoded symbols, and we divide that by the number of bits per symbol it would have taken to encode the text using fixed length encoding.

## Decoding



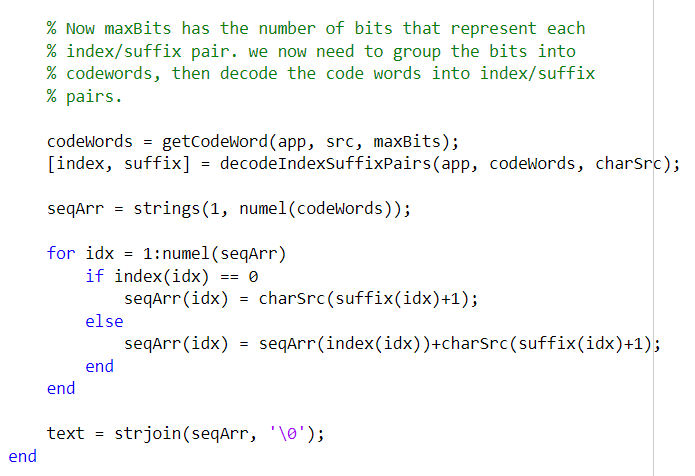


Figure 13 : Main Decoder Function | Lempel-Ziv

### Calculating the number of bits for each sequence

We calculate the number of bits like we did in the encoding part.

### Decoding the codeword

We divide the encoded bits into sequences of bits, each of a length that corresponds to the length calculated in the previous step.

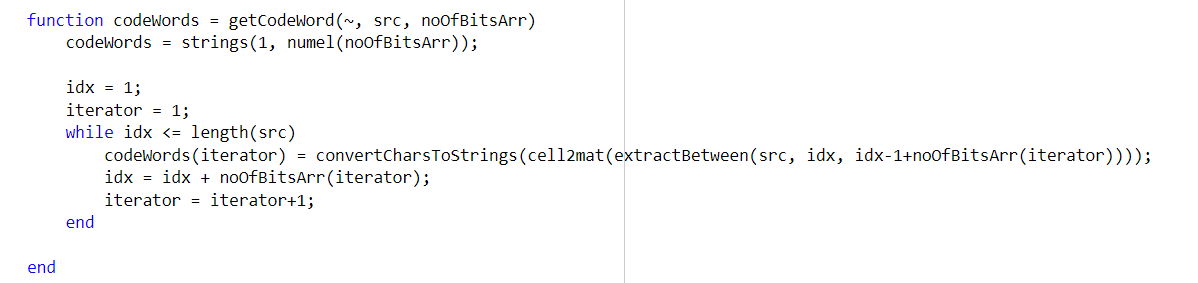


Figure 14 : Dividing The Encoded Bit Stream Into Sequences of Variable Lengths

### Get the Prefix/Suffix Pairs

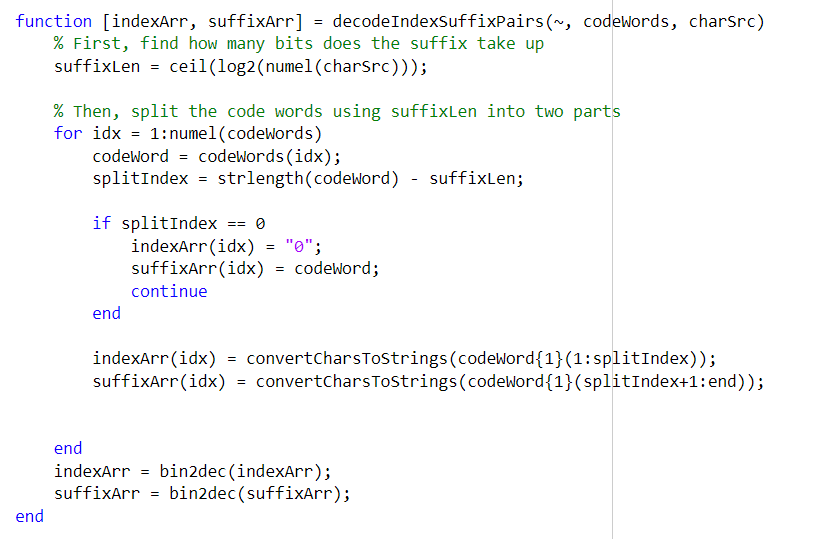
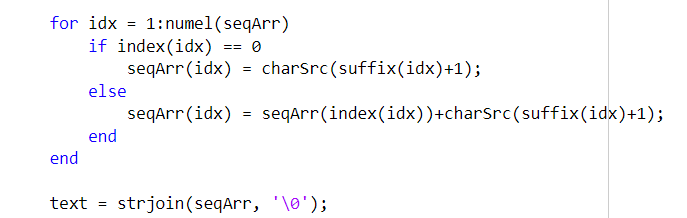


Figure 15 : Retrieving The (Prefix, Suffix) Pairs

We split each binary codeword into a prefix and a suffix based on the length of the suffix, then we return an array of indices and suffixes after we convert them to decimal.

### From Bits to text



We have reached the final step. Each index points to an entry in our decoded code word array. We loop on each index, and we replace it with its decoded codeword appended to it its respected suffix, appending the result of each index as we iterate.

# Results

## Adaptive Arithmetic

## Lempel-Ziv

### Test Case #1

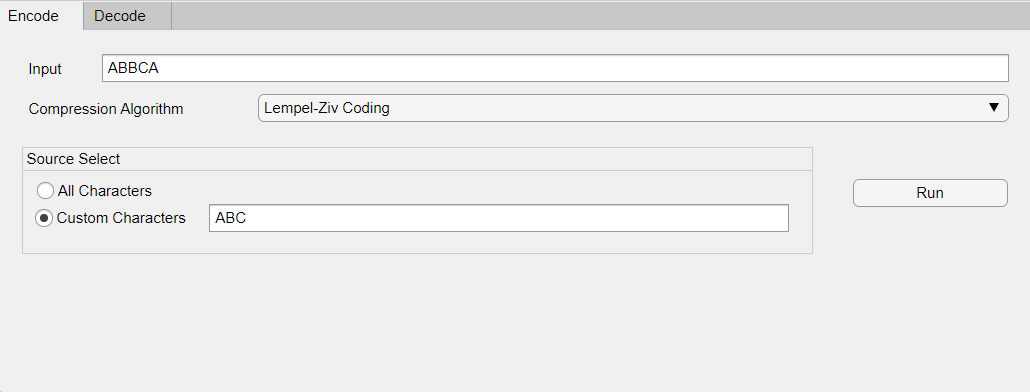
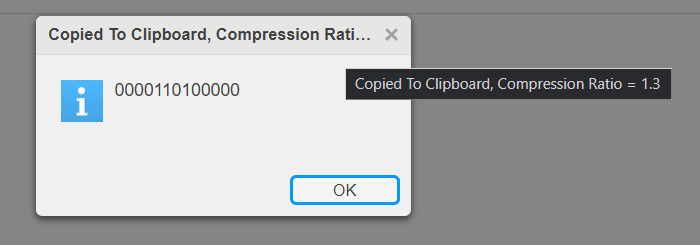
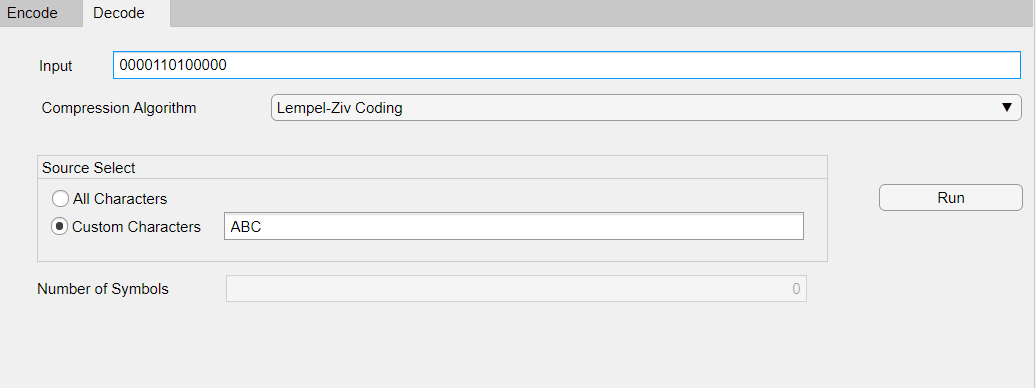
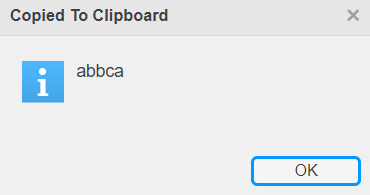


Figure 16 : Test Case #1 | Lempel-Ziv







### Test Case #2

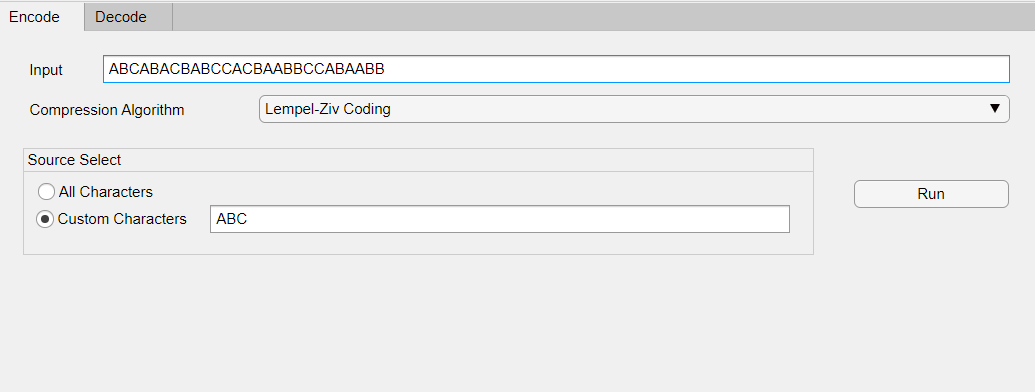
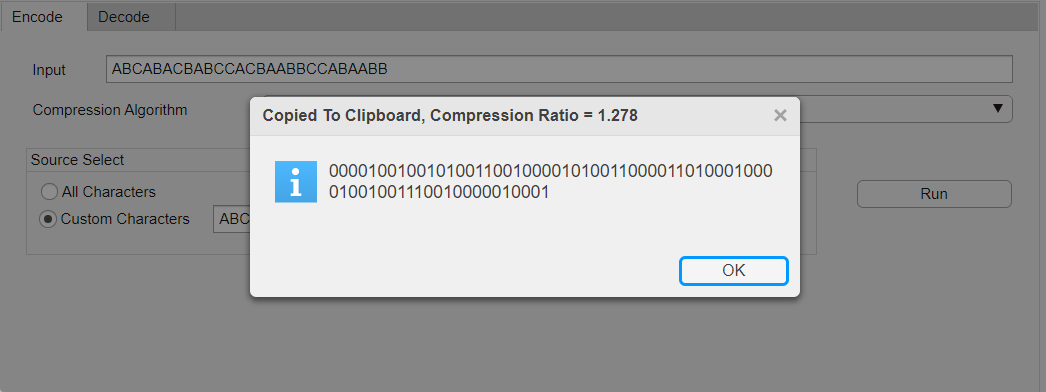
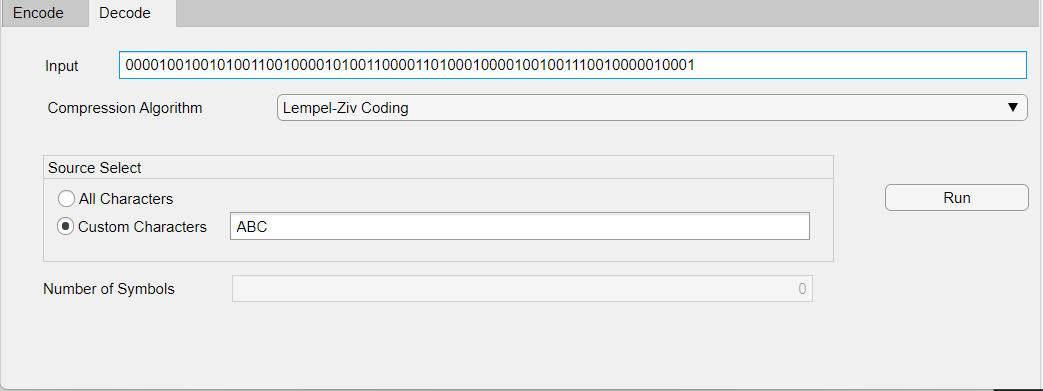
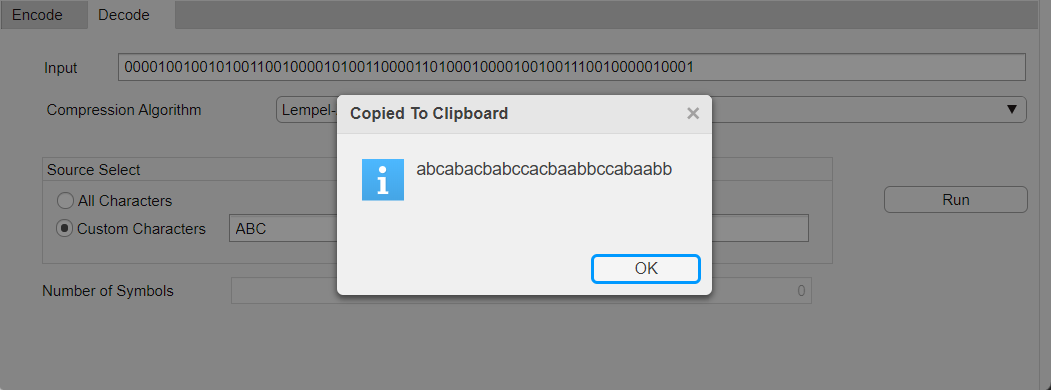


Figure 17 :Test Case #2 | Lempel-Ziv

### Test Case #3

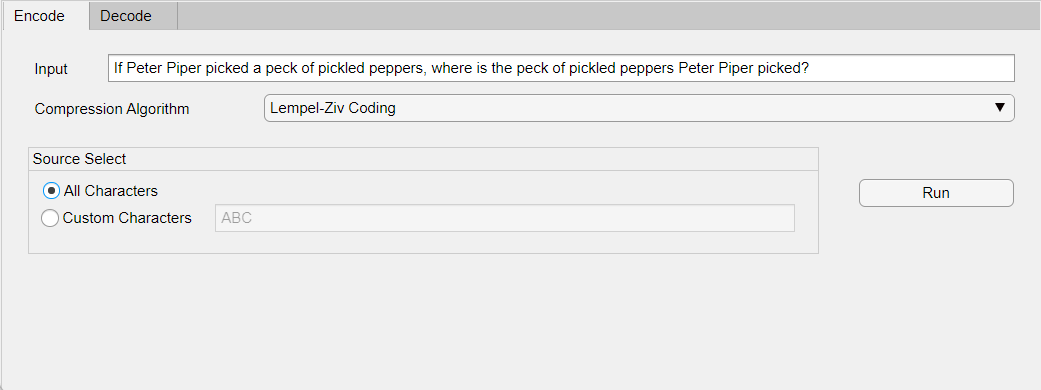
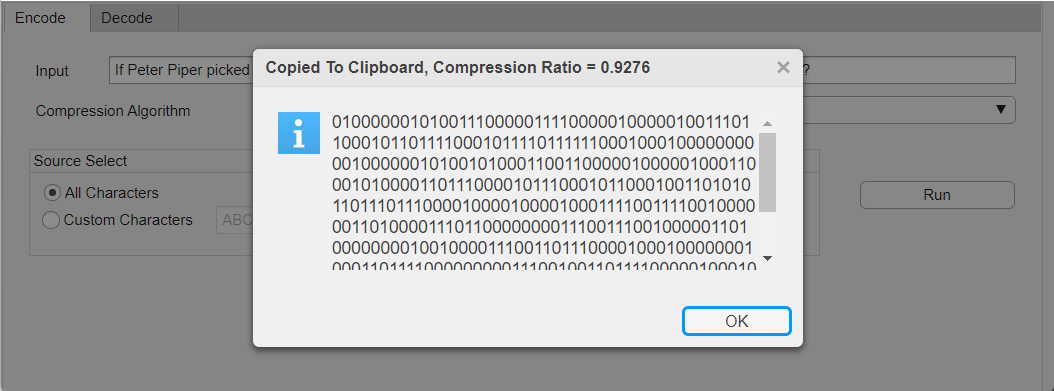
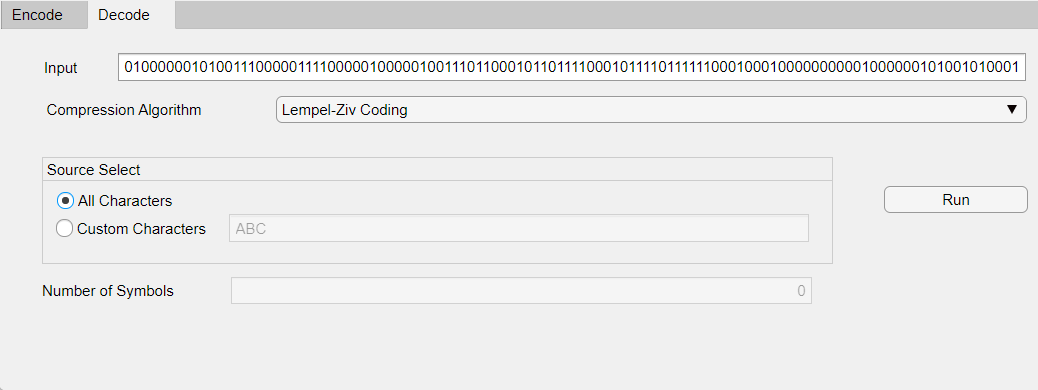
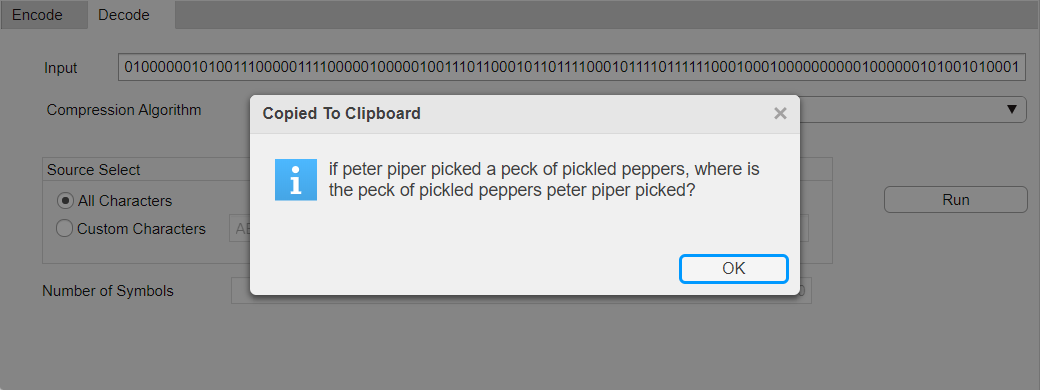


Figure 18 : Test Case #3 | Lempel-Ziv



# Discussion and Comments

## Adaptive Arithmetic

## Lempel-Ziv

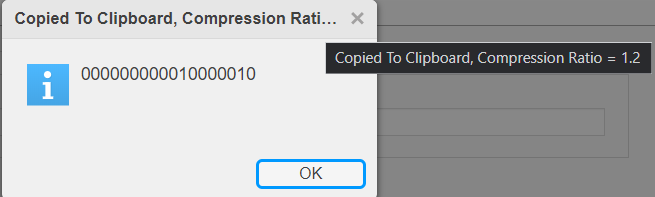
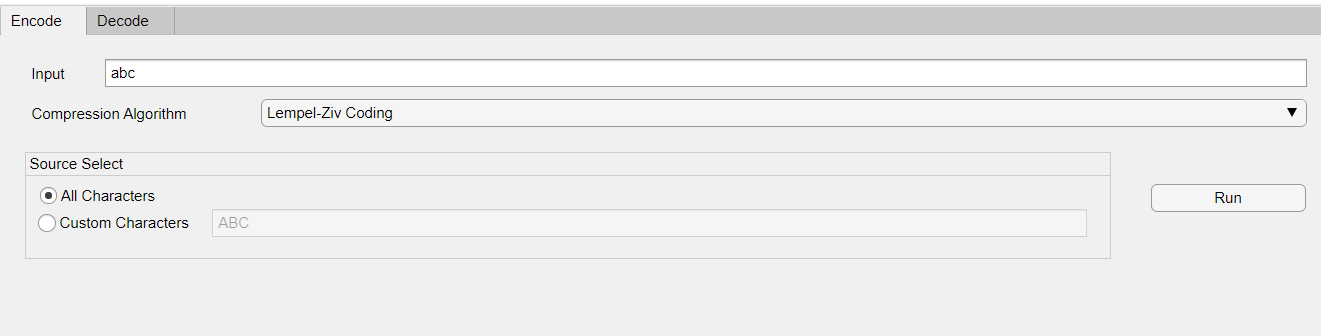
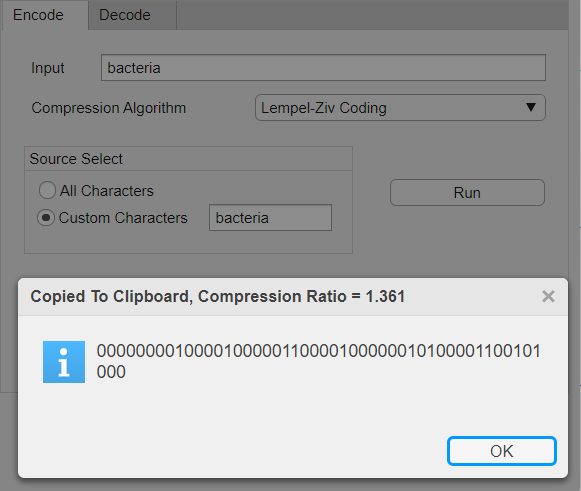
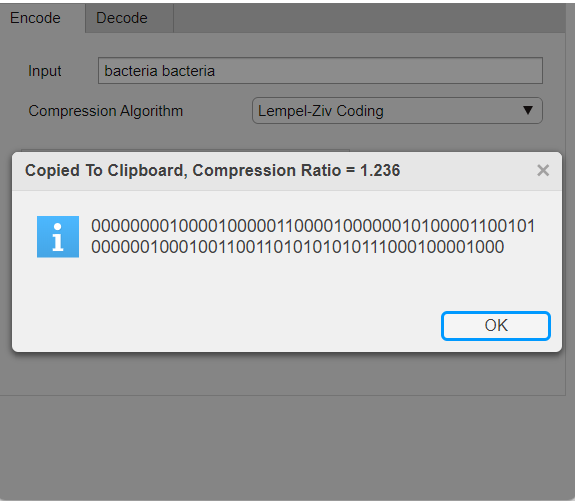
* The more characters we have to encode, the more efficient the compression will be, under the condition that the sequences of these characters are as common as possible.
  + If we have a relatively low number of characters to encode: 

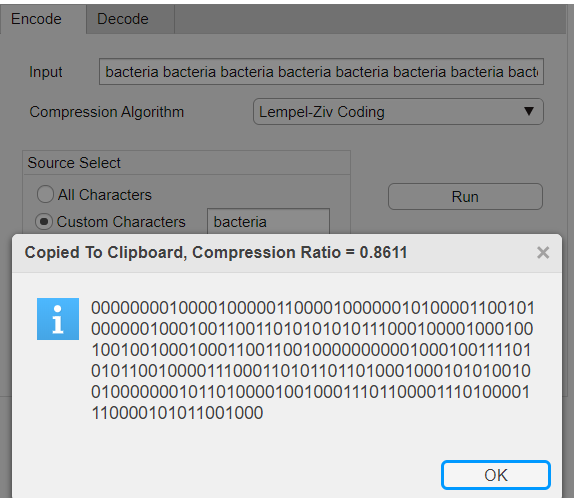
Figure 19 : Low number of characters, bad compression ratio



* + If we have a relatively high number of characters to encode (way higher than the number of source characters used) we get a better compression ratio.







* + We try compressing one and five paragraphs of *lorem ipsum* to prove our point

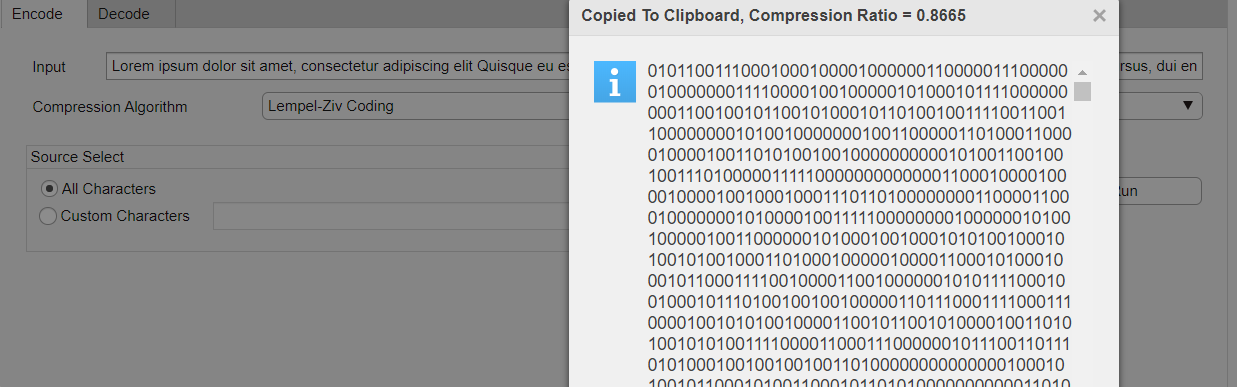


Figure 20 : Five Compressed Paragraphs of Lorem Ipsum

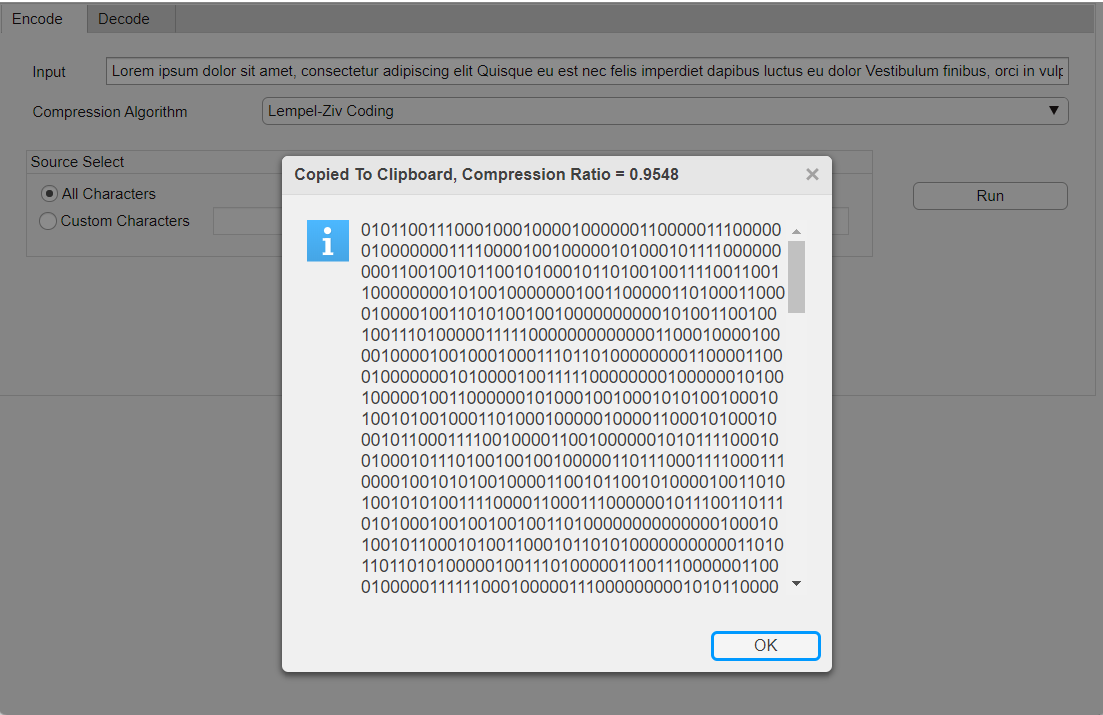
Notice how when we compressed five paragraphs the compression ratio went from 0.95 (when compressing one paragraph) to 0.86 (when compressing five paragraphs).

Figure 21 : One Compressed Paragraph of Lorem Ipsum