Project 3

Lempel-Ziv Encoding and Decoding

Adaptive Arithmetic Encoding and Decoding

Omar Tarek Ahmed 1180004

Hussein Ahmed Ibrahim 1180594



Contents

[List of Figures 2](#_Toc103976775)

[User Manual 3](#_Toc103976776)

[Encoding 3](#_Toc103976777)

[Decoding 4](#_Toc103976778)

[Adaptive Arithmetic coding 6](#_Toc103976779)

[Encoding 6](#_Toc103976780)

[Process 9](#_Toc103976781)

[Decoding 11](#_Toc103976782)

[Process 12](#_Toc103976783)

[Lempel-Ziv 13](#_Toc103976784)

[Encoding 13](#_Toc103976785)

[Process 14](#_Toc103976786)

[Getting the unique sequences 14](#_Toc103976787)

[Getting the Index/Suffix Pairs 15](#_Toc103976788)

[Sequence Enumeration 16](#_Toc103976789)

[Get the number of bits per sequence 16](#_Toc103976790)

[Turning the pairs into bits 16](#_Toc103976791)

[Calculating the compression ratio 17](#_Toc103976792)

[Decoding 18](#_Toc103976793)

[Calculating the number of bits for each sequence 19](#_Toc103976794)

[Decoding the codeword 19](#_Toc103976795)

[Get the Prefix/Suffix Pairs 19](#_Toc103976796)

[From Bits to text 20](#_Toc103976797)

[Results 20](#_Toc103976798)

[Adaptive Arithmetic 20](#_Toc103976799)

[Test Case #1 20](#_Toc103976800)

[Test Case #2 22](#_Toc103976801)

[Test Case #3 24](#_Toc103976802)

[Lempel-Ziv 26](#_Toc103976803)

[Test Case #1 26](#_Toc103976804)

[Test Case #2 28](#_Toc103976805)

[Test Case #3 29](#_Toc103976806)

[Discussion and Comments 31](#_Toc103976807)

[Adaptive Arithmetic 31](#_Toc103976808)

[Lempel-Ziv 32](#_Toc103976809)

# List of Figures

[Figure 1 : Main Screen 3](#_Toc103978412)

[Figure 2 : Encoding Output Example 4](#_Toc103978413)

[Figure 3 : Decoding Tab | Lempel-Ziv 4](#_Toc103978414)

[Figure 4 : Decoding Tab | Adaptive Arithmetic 5](#_Toc103978415)

[Figure 5 : Decoder Output 5](#_Toc103978416)

[Figure 6 part 1 Adaptive arithmetic Encoding Function 6](#_Toc103978417)

[Figure 7:part 2 Adaptive arithmetic Encoding Function 7](#_Toc103978418)

[Figure 8:part 3 Adaptive arithmetic Encoding Function 8](#_Toc103978419)

[Figure 9:part 4 Adaptive arithmetic Encoding Function 9](#_Toc103978420)

[Figure 10: part 1 Adaptive arithmetic decoding Function 11](#_Toc103978421)

[Figure 11:part 2 Adaptive arithmetic decoding Function 12](#_Toc103978422)

[Figure 12 : Lempel- Ziv Encoding Function 13](#_Toc103978423)

[Figure 13 : Getting The Unique Sequences 14](#_Toc103978424)

[Figure 14 : Getting the (Index, Suffix) Pairs 15](#_Toc103978425)

[Figure 15 : Turning The Pairs Matrix Into An Array Of Numbers 16](#_Toc103978426)

[Figure 16 : Getting The Number Of Bits That Each Sequence Will Take 16](#_Toc103978427)

[Figure 17 : Encoding To Bits 16](#_Toc103978428)

[Figure 18 : Compression Ratio Code 17](#_Toc103978429)

[Figure 19 : Main Decoder Function | Lempel-Ziv 18](#_Toc103978430)

[Figure 20 : Dividing The Encoded Bit Stream Into Sequences of Variable Lengths 19](#_Toc103978431)

[Figure 21 : Retrieving The (Prefix, Suffix) Pairs 19](#_Toc103978432)

[Figure 22: Test Case #1 | Adaptive Arithmetic 20](#_Toc103978433)

[Figure 23: Test Case #2 | Adaptive Arithmetic 22](#_Toc103978434)

[Figure 24: Test Case #3 | Adaptive Arithmetic 24](#_Toc103978435)

[Figure 25 : Test Case #1 | Lempel-Ziv 26](#_Toc103978436)

[Figure 26 :Test Case #2 | Lempel-Ziv 28](#_Toc103978437)

[Figure 27 : Test Case #3 | Lempel-Ziv 29](#_Toc103978438)

[Figure 28: Low number of characters, Good compression ratio 31](#_Toc103978439)

[Figure 29 : Low number of characters, bad compression ratio 32](#_Toc103978440)

[Figure 30 : Five Compressed Paragraphs of Lorem Ipsum 34](file:///E:\Senior-1\Spring%202022\ELCN446%20Advanced%20Topics%20in%20Communications-1\Project%203\Universal-Codes\Project%203.docx#_Toc103978441)

[Figure 31 : One Compressed Paragraph of Lorem Ipsum 35](file:///E:\Senior-1\Spring%202022\ELCN446%20Advanced%20Topics%20in%20Communications-1\Project%203\Universal-Codes\Project%203.docx#_Toc103978442)

# User Manual

## Encoding

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

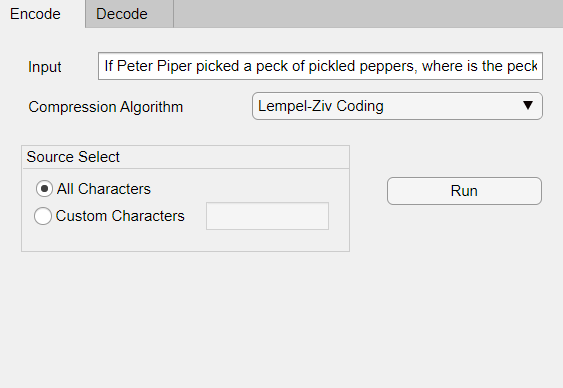


Figure : Main Screen

Here we have *two* tabs, one for encoding, and the other for decoding. 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:

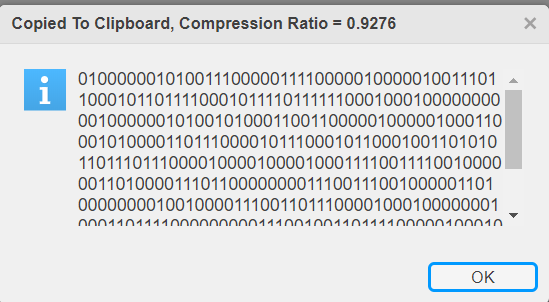


Figure : Encoding Output Example

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:

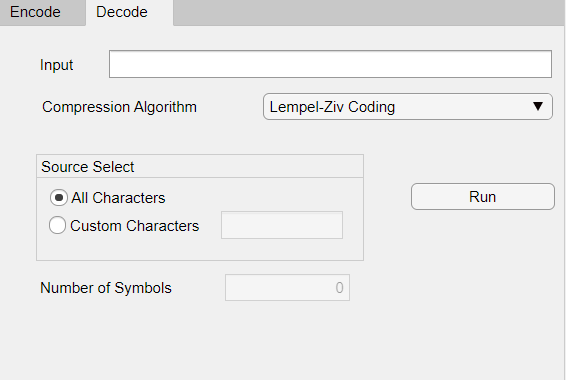


Figure : Decoding Tab | Lempel-Ziv

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.

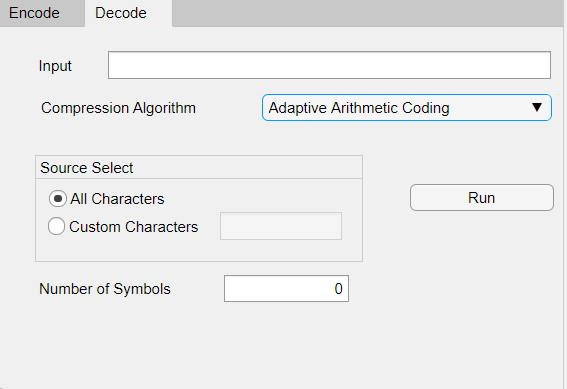


Figure : Decoding Tab | Adaptive Arithmetic

For Adaptive Arithmetic, After using the encoder, paste the output in the *input field* in the decoder tab then enter the number of sequence symbol to stop at it then press *Run*. This should output the same text entered in the encoder *input field.*

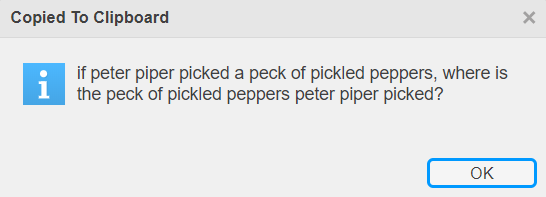


Figure : Decoder Output

# Adaptive Arithmetic coding

## Encoding

The encoding function is classified into 4 parts, will be explained in details in the process section:

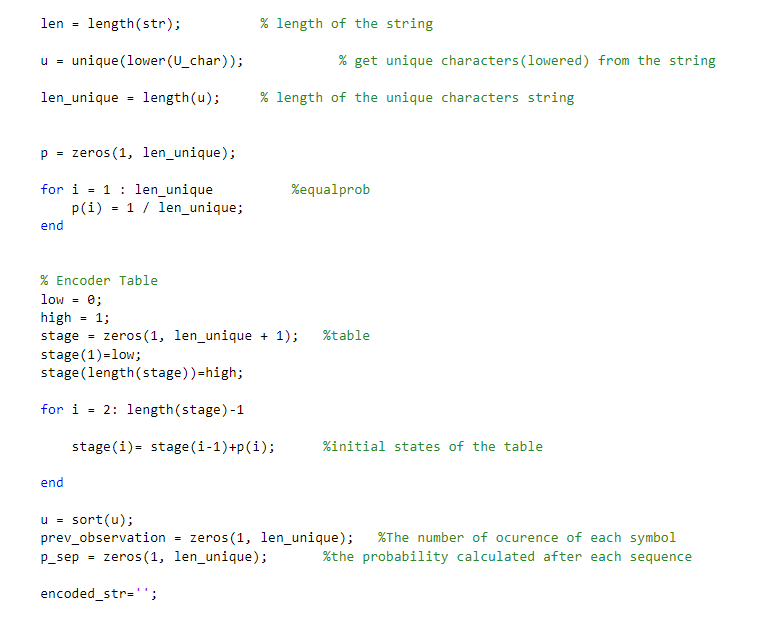


Figure part 1 Adaptive arithmetic Encoding Function

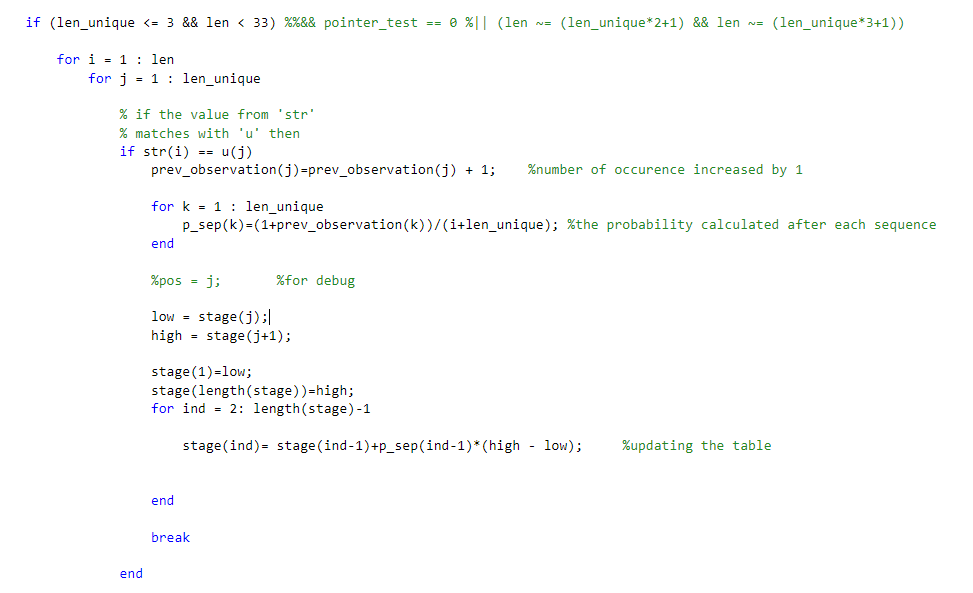


Figure :part 2 Adaptive arithmetic Encoding Function

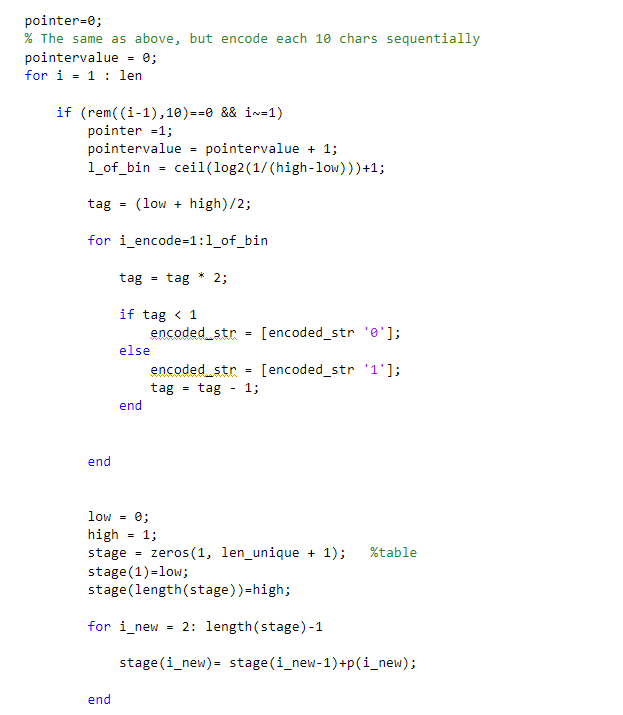


Figure :part 3 Adaptive arithmetic Encoding Function

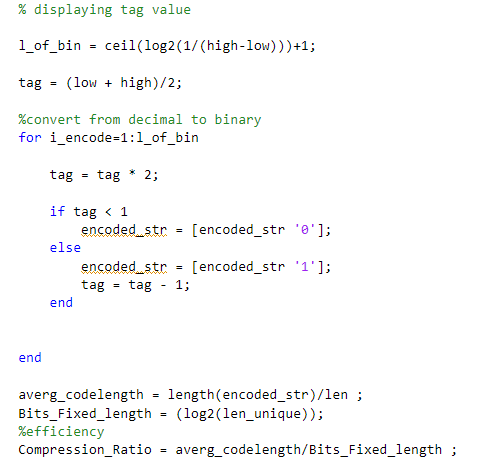


Figure :part 4 Adaptive arithmetic Encoding Function

### Process

To begin with part 1, start initializing the variables with the inputs and make sure to convert the chars to be insensitive and get the length of the string entered and of the unique chars. Then get the initial probability which is equal to all of them (divide by length of unique). Create a stage array which will be our table its first index is 0 and last element equals 1 initially and the in between element will be the calculated probability. Define 2 arrays and initialize them with 0, the first one is the number of occurrences of unique elements while the other is the new probability which will be calculated to separate between the low (first element in stage array) and high (last element in stage array).

In the second part, we will begin the encoding process. In the first for loop till the length of the string, we will have another loop till the length of the unique elements. Check if the value of string matches the value of unique element then increase the observation of the unique element to 1 (only the one observed). Then a small loop to calculate the probability of separation and finally prepare our table in which the first value is the current value of the string and the last view is the value after the current value of the string and refill the remaining the values of the array stage (table) with the calculated probability.

In the third part, we will work in it if the length of unique chars is greater than 3 (a text) or length of string is greater than 33. We will work on 10 symbols simultaneously. But we add an if statement to enter it every 10 iterations to initialize the values and combine the encoded strings and convert them to binary.

In the fourth part, we combine the encoded string and convert them in binary for part 2 and for the remaining in part 3 then calculate the efficiency by calculating by calculating average code length divided by the fixed length bits.

## Decoding

The decoding function is classified into 3 parts. Part 1 will be the same as encoding initializing but added to it the conversion from binary into decimal fraction. Part 2 & 3 will be for the decoding and part 3 will be for the text. More details will be explained in the process section:

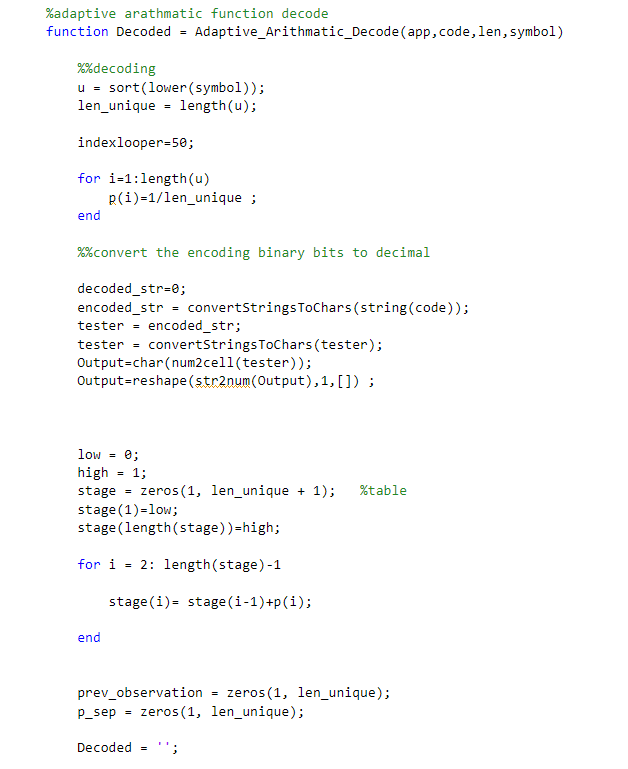


Figure : part 1 Adaptive arithmetic decoding Function

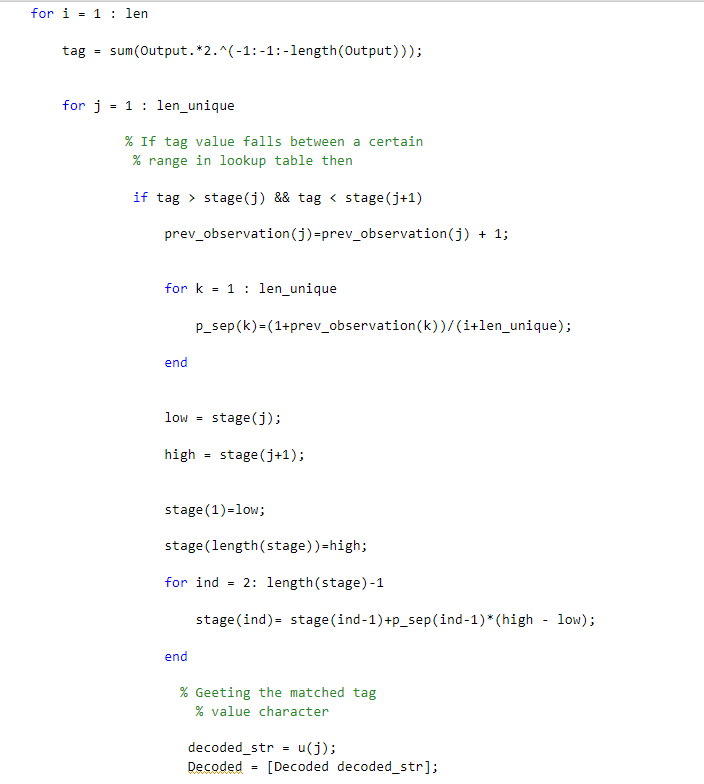


Figure :part 2 Adaptive arithmetic decoding Function

### Process

Part 1 will be similar to part 1 in encode, but there’s some differences. The similarity is initializing the variables, but the differences is converting the bits into decimal fraction, but the exact number will be in part 2. In this part only converting string into chars into cells into numbers.

In Part 2, we will check that the values in which range and in each iteration, we will do similar to encoding, increase the observation of the unique element to 1 (only the one observed). Then a small loop to calculate the probability of separation and finally prepare our table in which the first value is the current value of the string and the last view is the value after the current value of the string then store the decoded string in the decoded variable.

In the second part, we will begin the encoding process. In the first for loop till the length of the string, we will have another loop till the length of the unique elements. Check if the value of string matches the value of unique element then e in which and refill the remaining the values of the array stage (table) with the calculated probability.

In the third part, we will work in it if it was a text. We will distribute it combine them.

# Lempel-Ziv

## Encoding

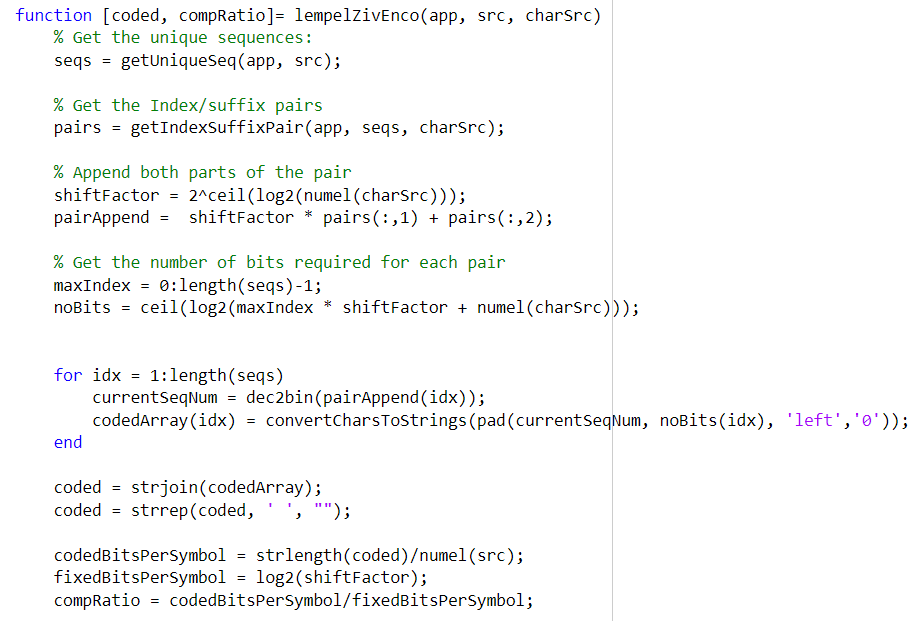


Figure : Lempel- Ziv Encoding Function

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

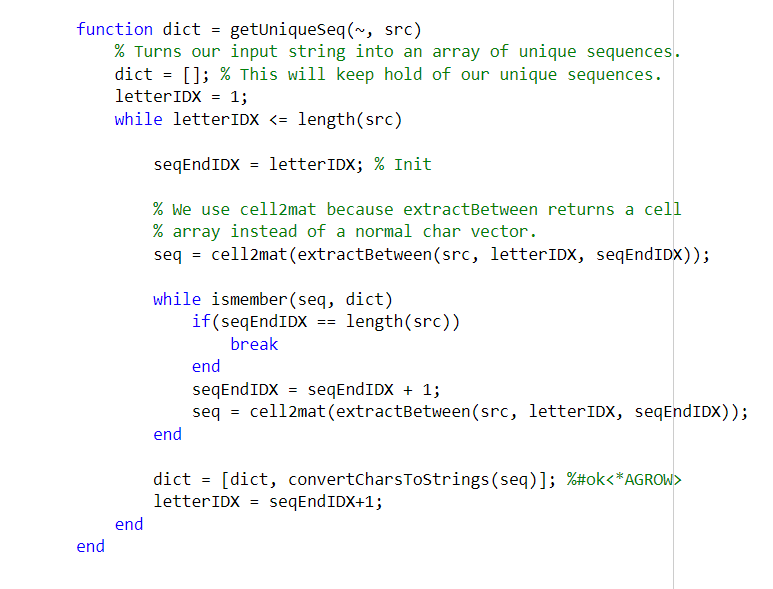
We need to divide out text into as much unique segments as we can. The function in the figure below does just that.

Figure : 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

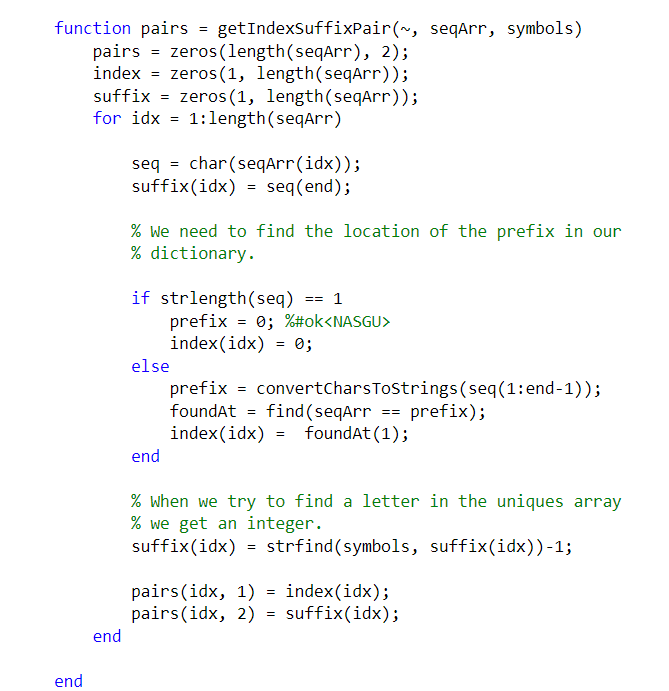


Figure : Getting the (Index, Suffix) Pairs

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

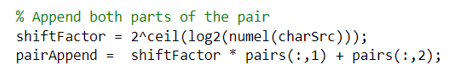


Figure : Turning The Pairs Matrix Into An Array Of Numbers

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

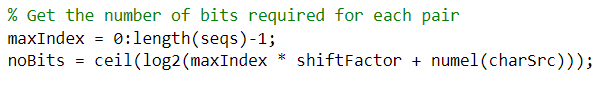


Figure : Getting The Number Of Bits That Each Sequence Will Take

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

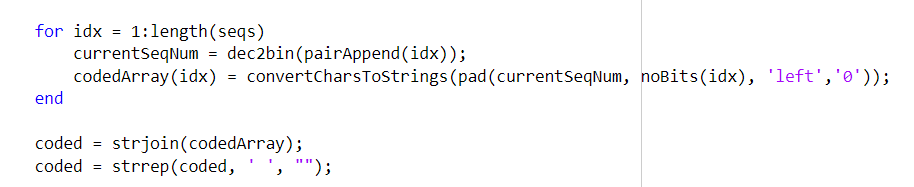


Figure : Encoding To Bits

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

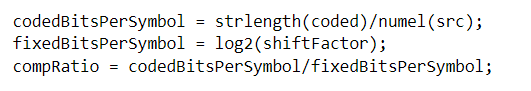
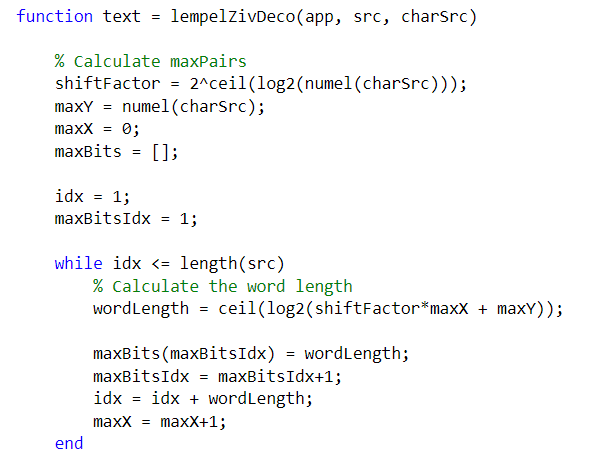


Figure : Compression Ratio Code

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



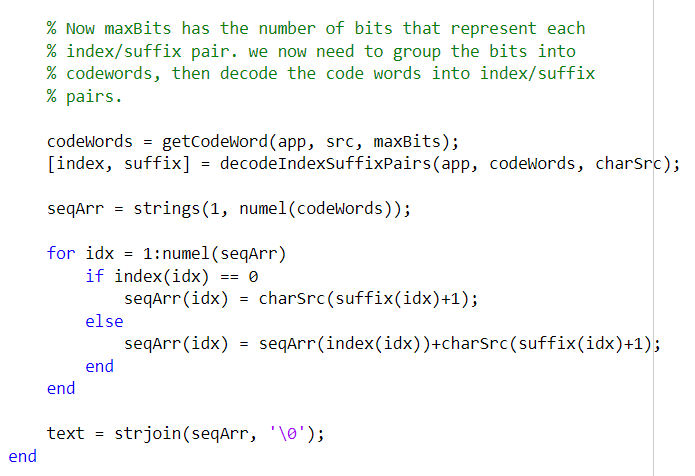


Figure : Main Decoder Function | Lempel-Ziv

### 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.

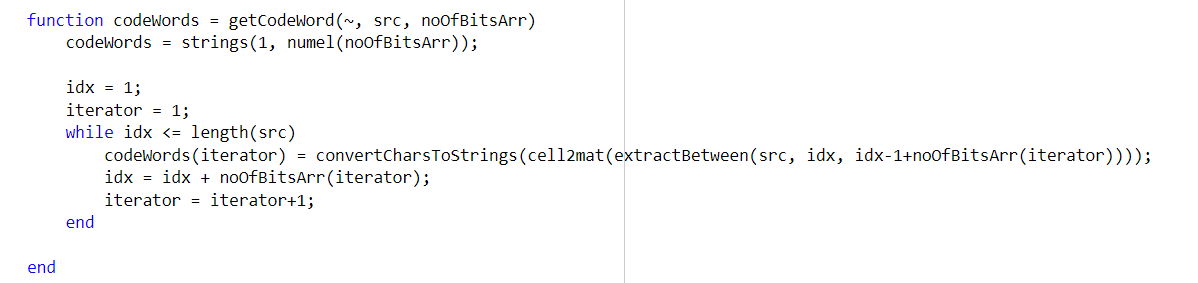


Figure : Dividing The Encoded Bit Stream Into Sequences of Variable Lengths

### Get the Prefix/Suffix Pairs

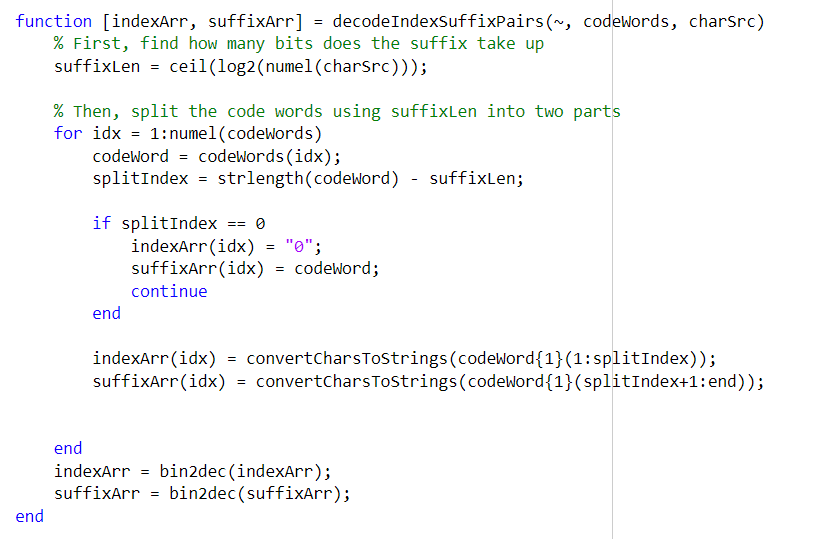
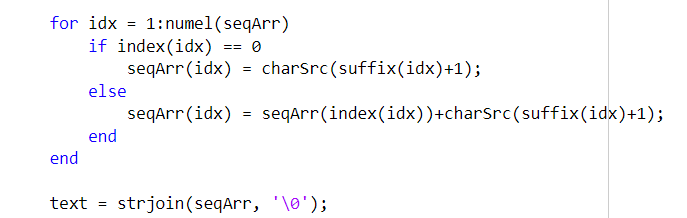


Figure : Retrieving The (Prefix, Suffix) Pairs

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

### Test Case #1

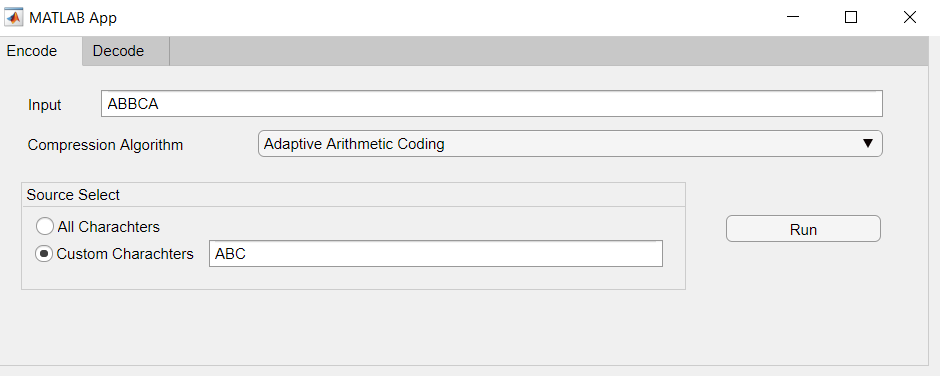
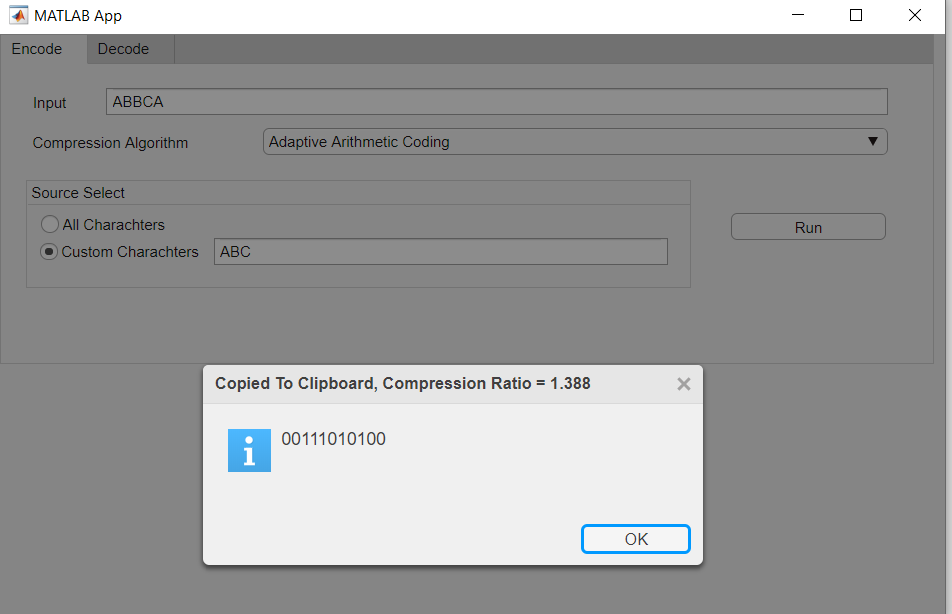
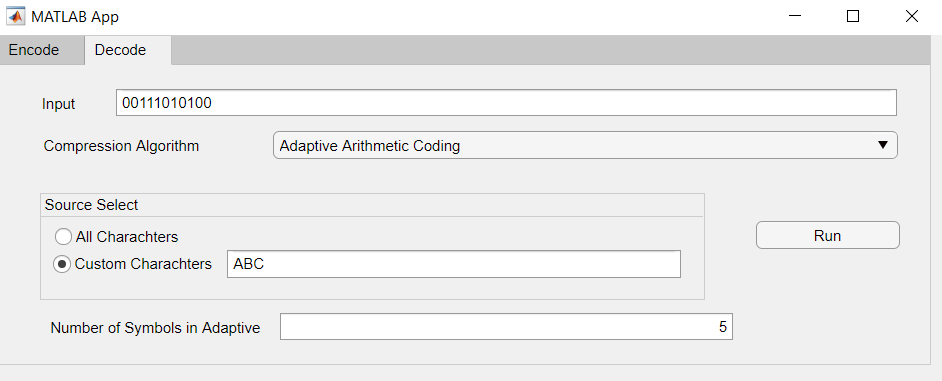
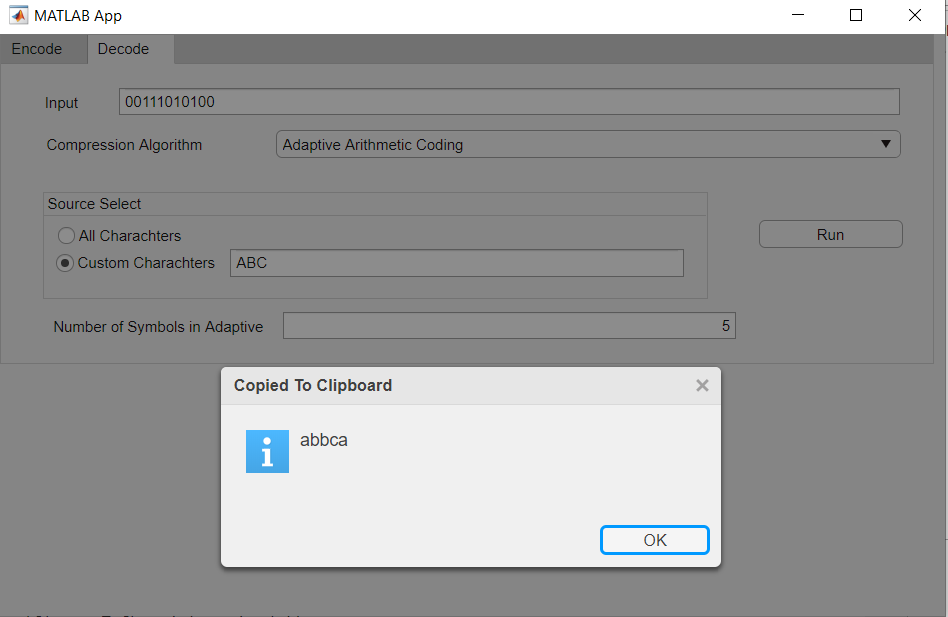


Figure : Test Case #1 | Adaptive Arithmetic







### Test Case #2

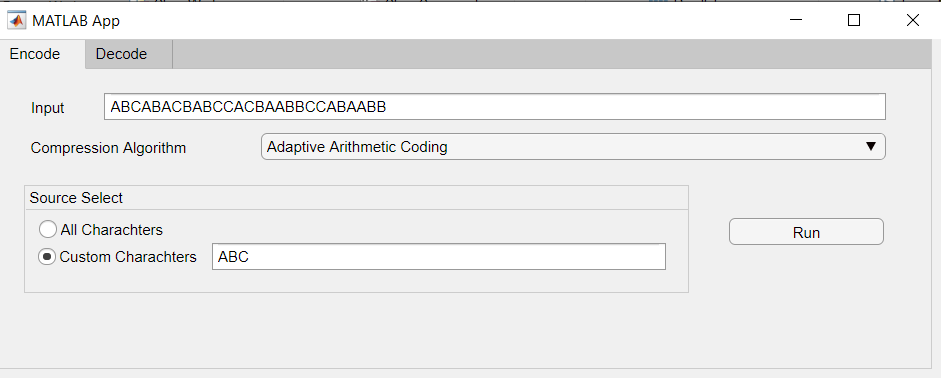
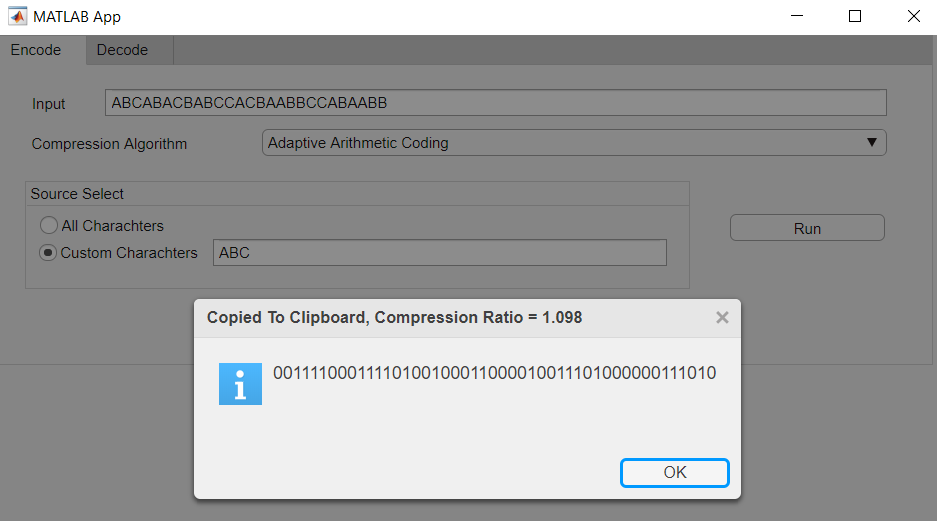
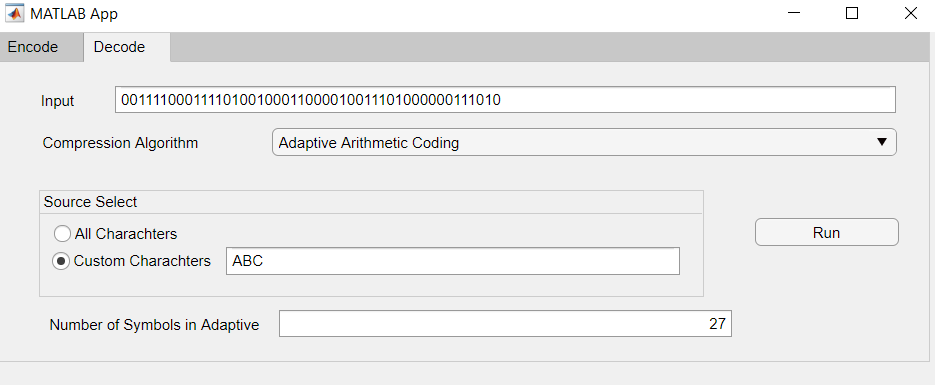
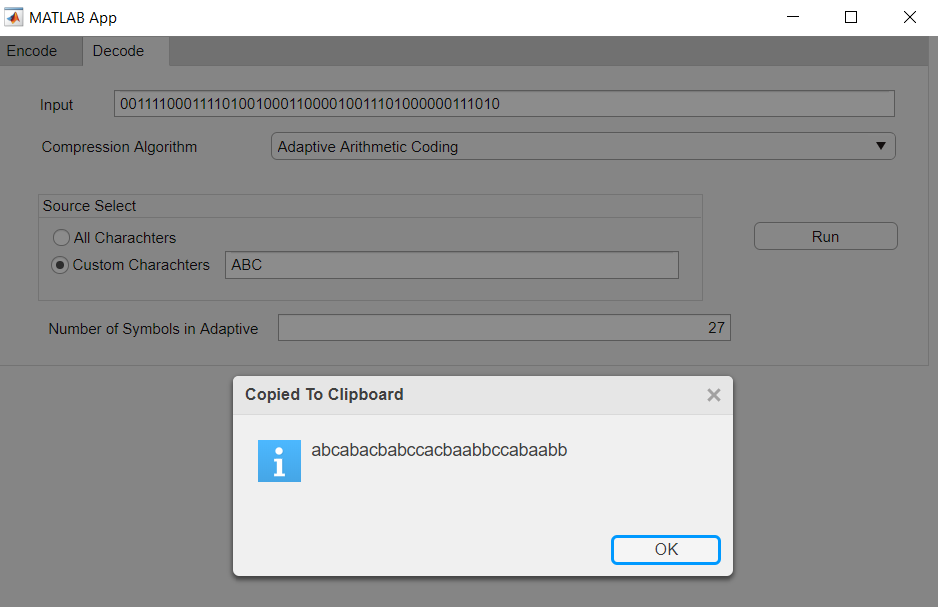


Figure : Test Case #2 | Adaptive Arithmetic







### Test Case #3

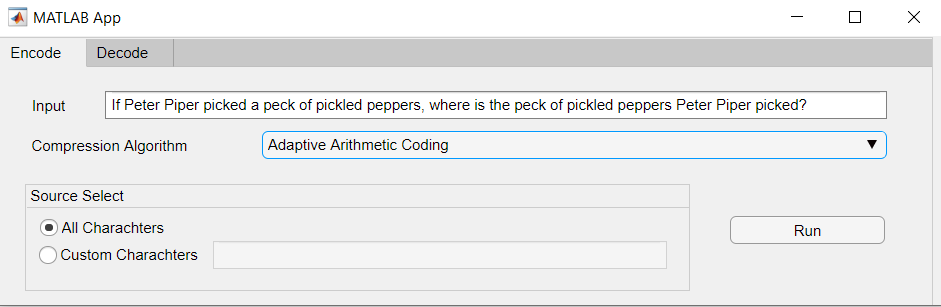
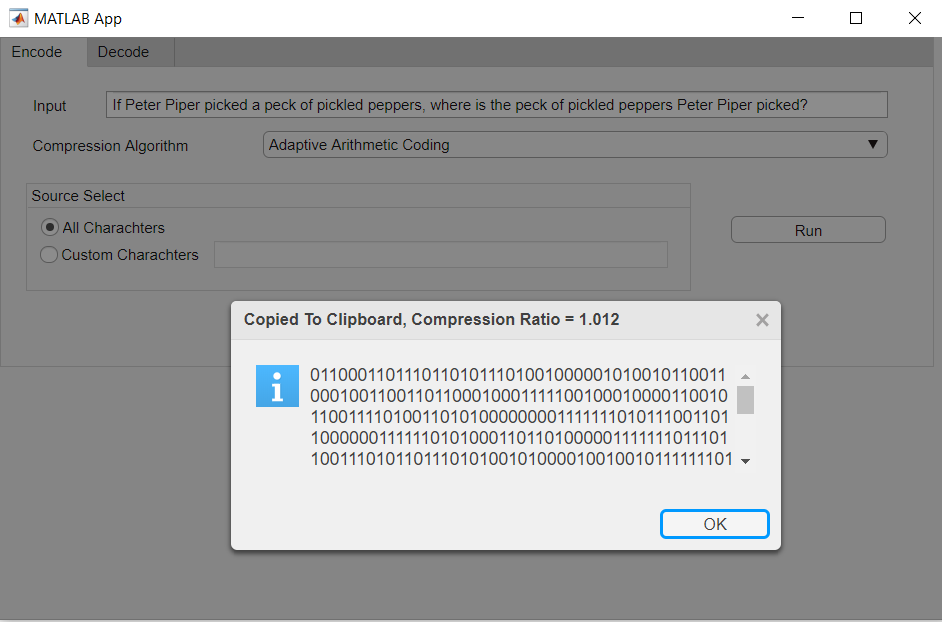
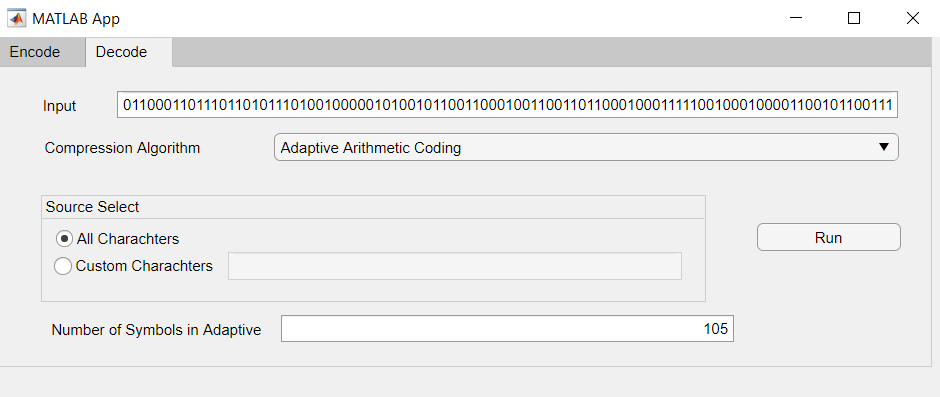
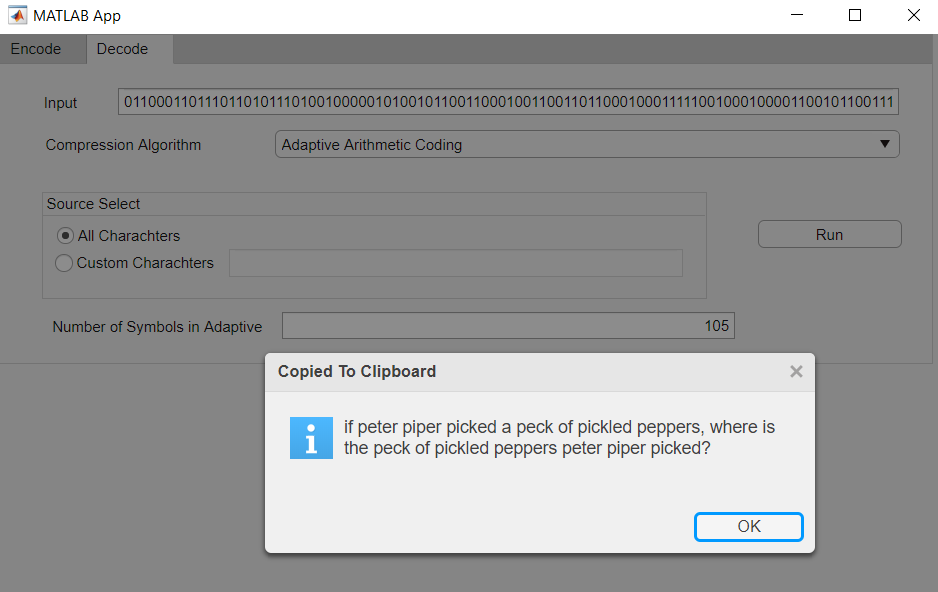


Figure : Test Case #3 | Adaptive Arithmetic







## Lempel-Ziv

### Test Case #1

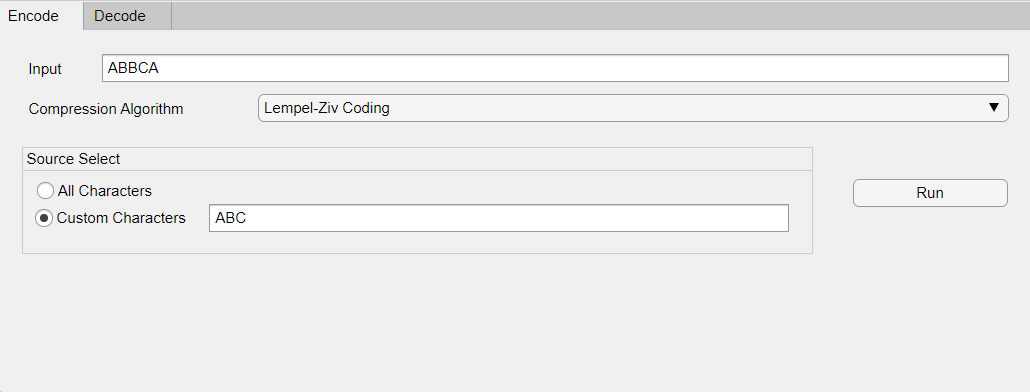
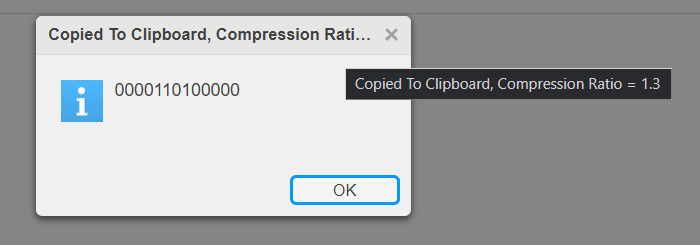
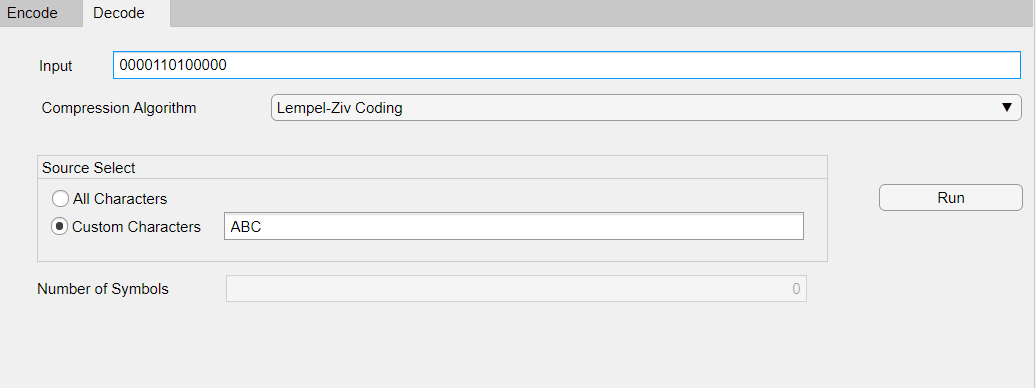
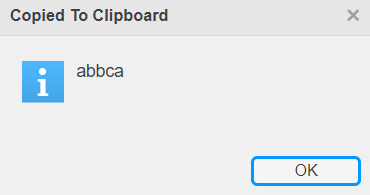


Figure : Test Case #1 | Lempel-Ziv







### Test Case #2

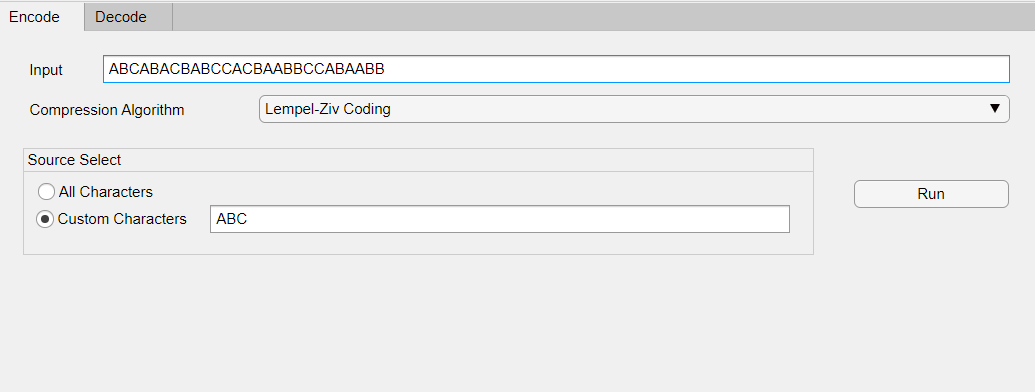
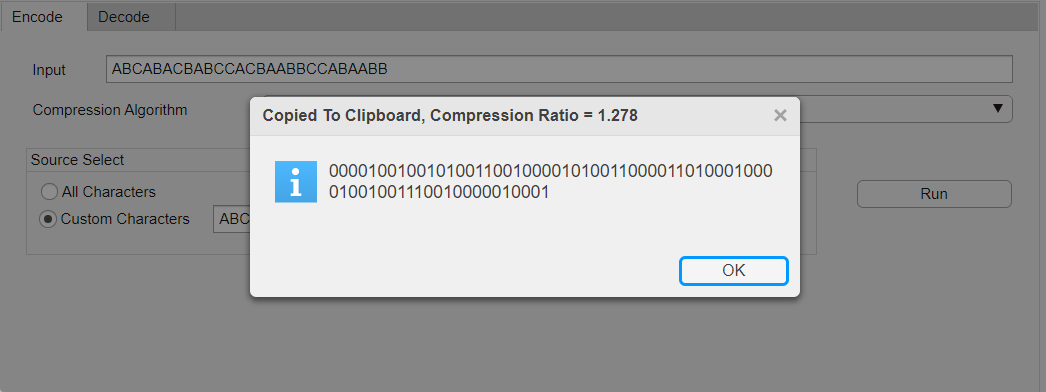
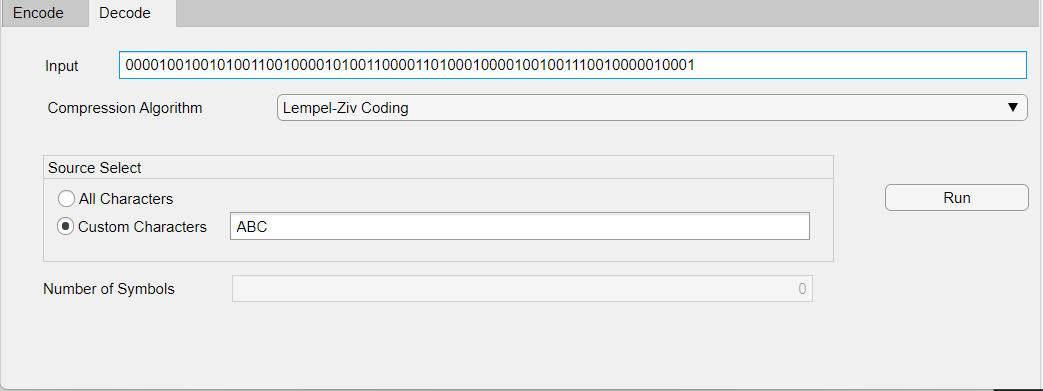
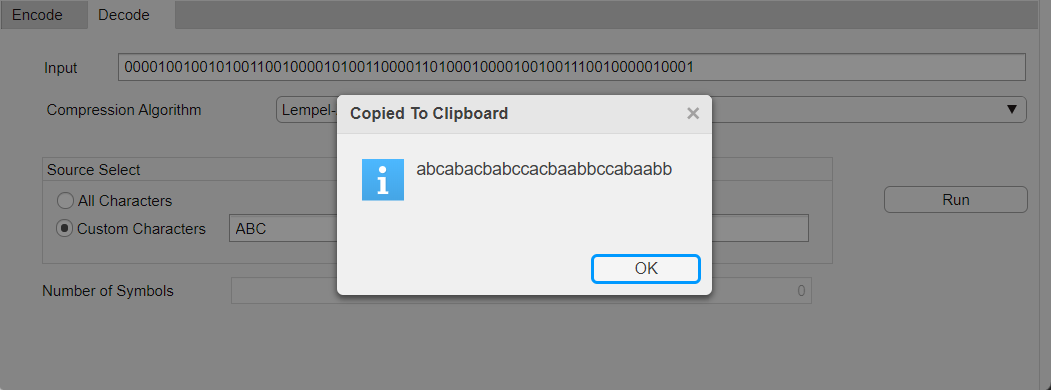


Figure :Test Case #2 | Lempel-Ziv

### Test Case #3

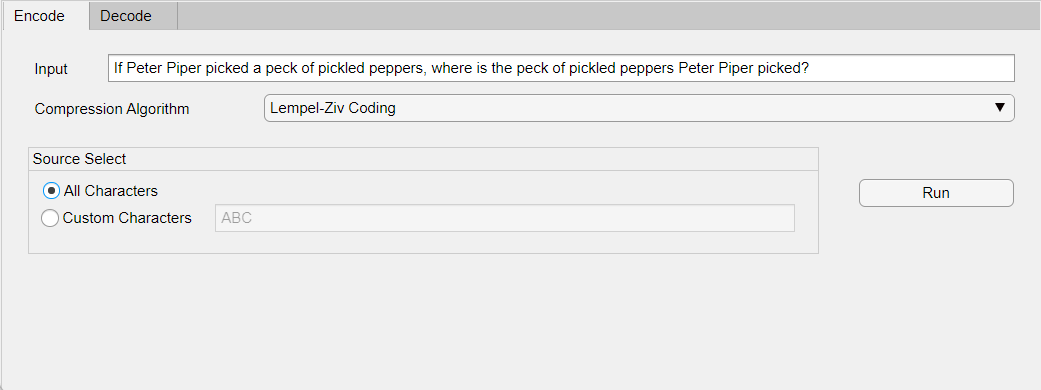
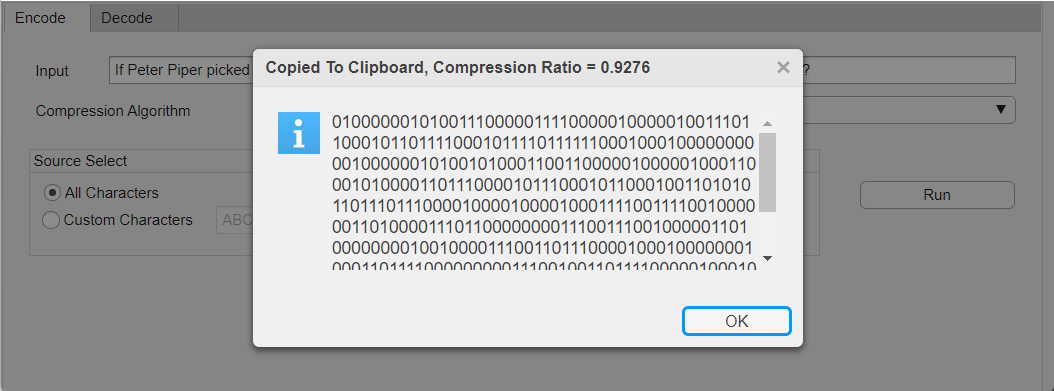
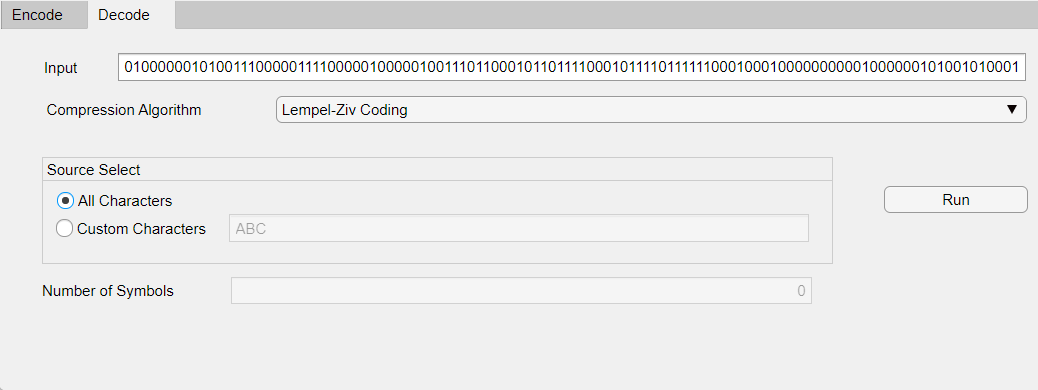
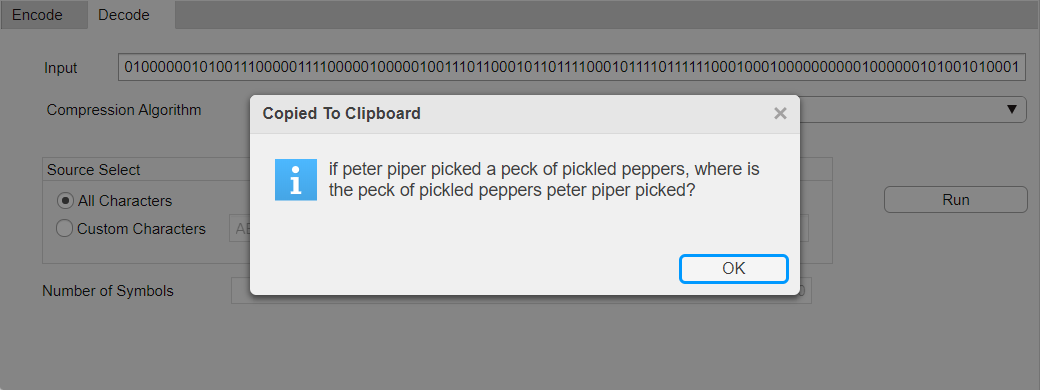


Figure : Test Case #3 | Lempel-Ziv



# Discussion and Comments

## Adaptive Arithmetic

* For a small length input with a smaller number of unique chars, it is better to use adaptive arithmetic encoding since it will be more efficient.
* If we have a relatively low number of characters to encode:

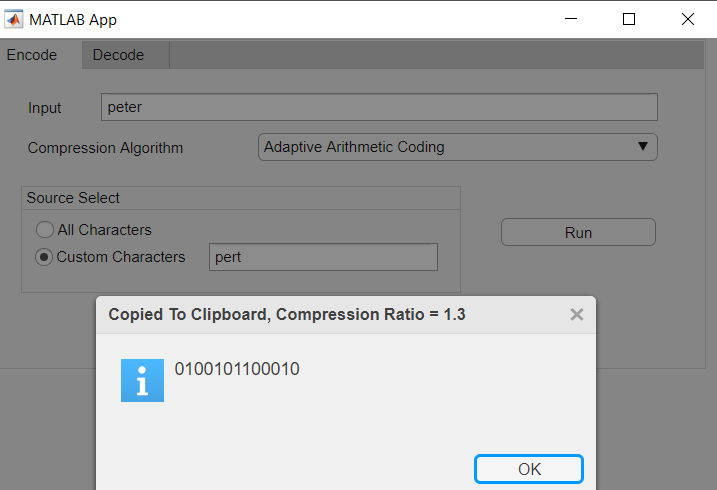
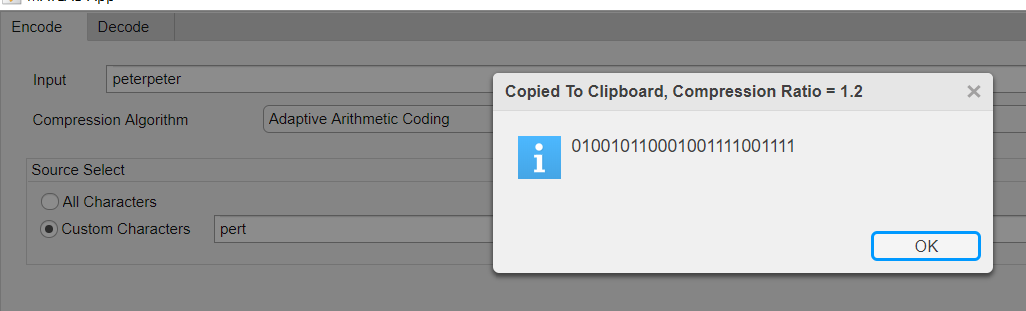


Figure : Low number of characters, Good compression ratio

* + 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.



## Lempel-Ziv

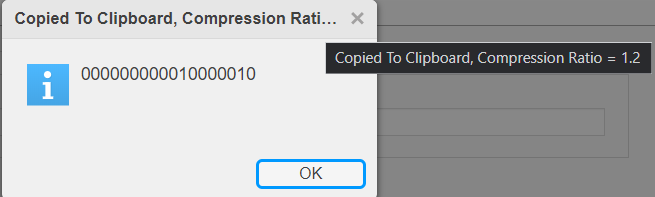
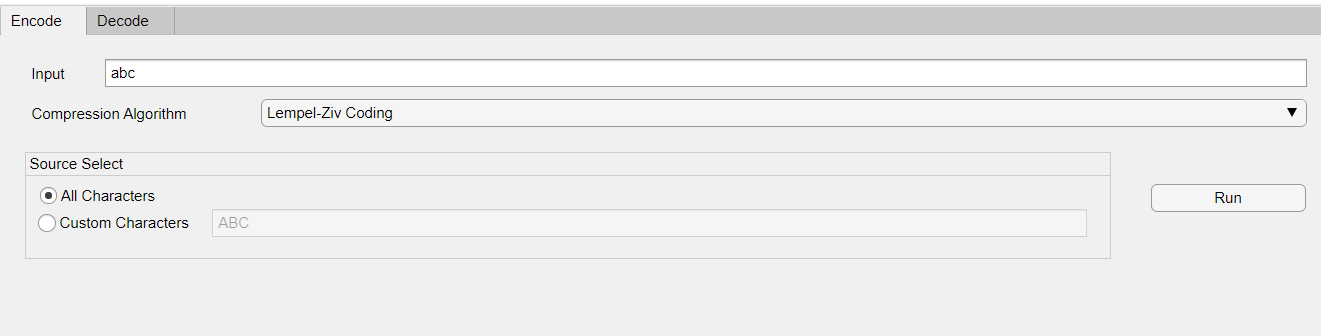
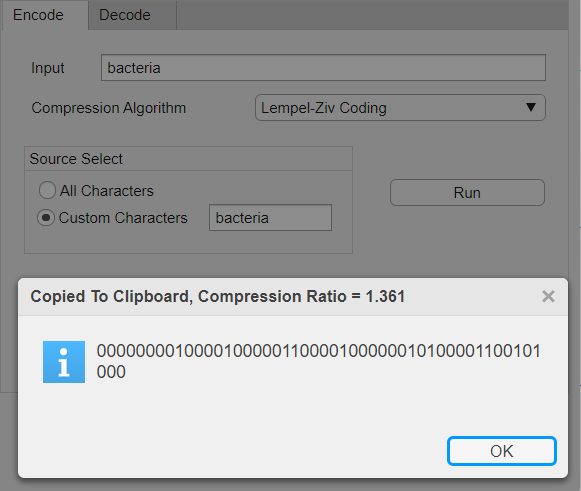
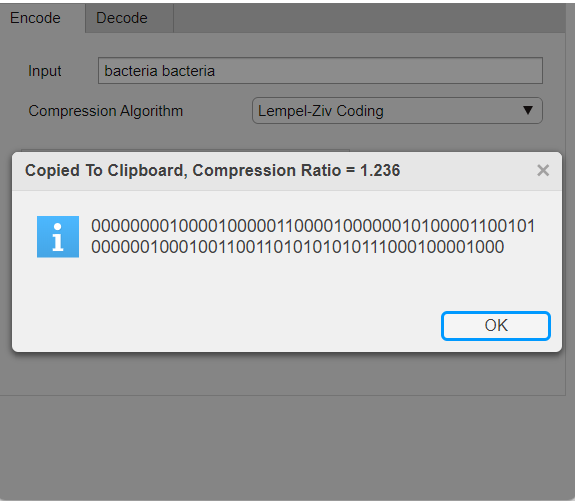
* The more characters we have to encode, the more efficient the compression will be, under the condition that the sequences of these characters are as common as possible.
  + If we have a relatively low number of characters to encode: 

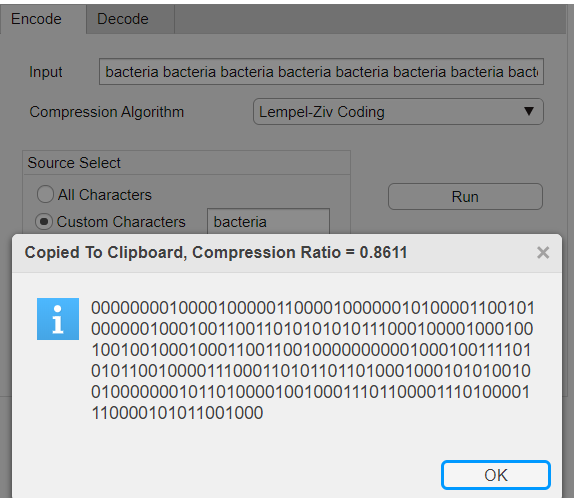
Figure : Low number of characters, bad compression ratio



* + 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

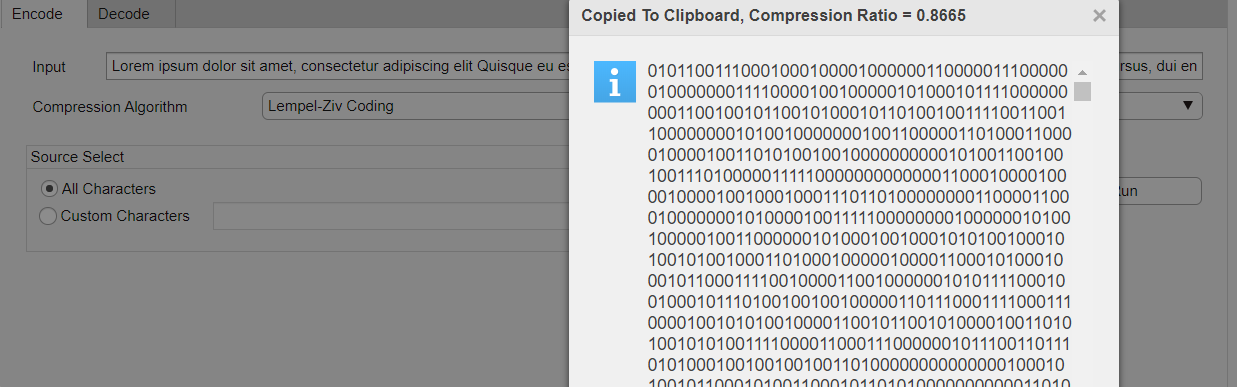


Figure : Five Compressed Paragraphs of Lorem Ipsum

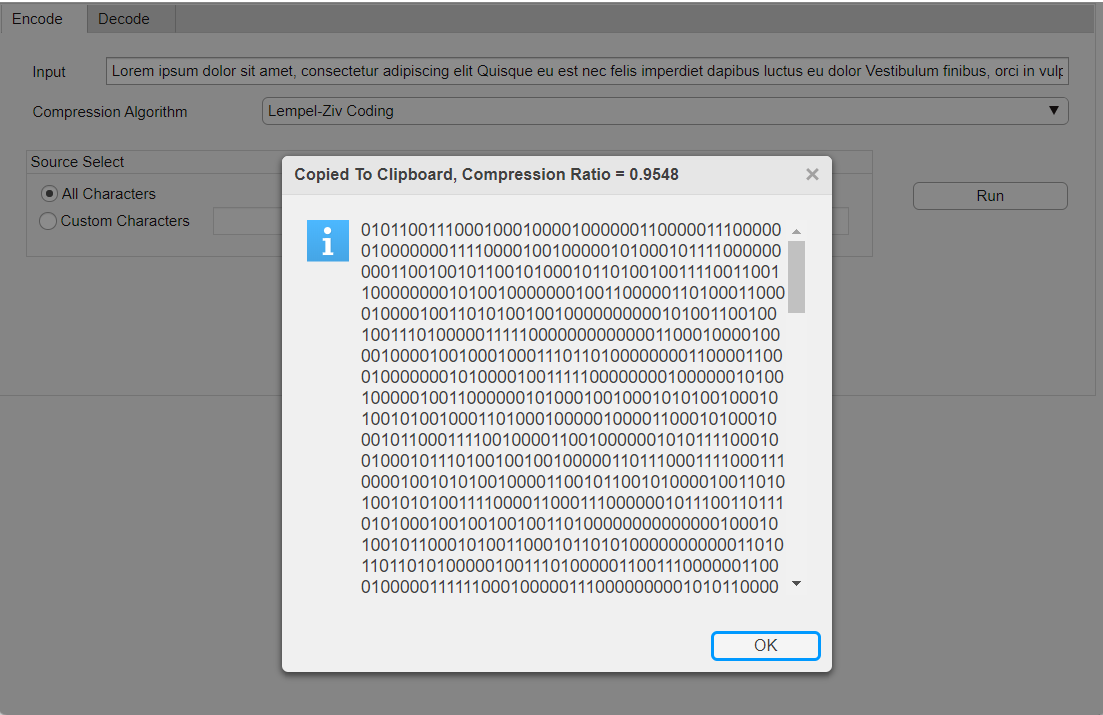
Notice how when we compressed five paragraphs the compression ratio went from 0.95 (when compressing one paragraph) to 0.86 (when compressing five paragraphs). This does not necessarily mean that the final size will be smaller.

Figure : One Compressed Paragraph of Lorem Ipsum