

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 |
| User Manual..... | 3 |
| Encoding..... | 3 |
| Decoding | 4 |
| Adaptive Arithmetic coding | 6 |
| Encoding..... | 6 |
| Process | 9 |
| Decoding | 11 |
| Process | 12 |
| Lempel-Ziv | 13 |
| Encoding..... | 13 |
| Process | 14 |
| Getting the unique sequences | 14 |
| Getting the Index/Suffix Pairs | 15 |
| Sequence Enumeration | 16 |
| Get the number of bits per sequence..... | 16 |
| Turning the pairs into bits..... | 16 |
| Calculating the compression ratio | 17 |
| Decoding | 18 |
| Calculating the number of bits for each sequence | 19 |
| Decoding the codeword..... | 19 |
| Get the Prefix/Suffix Pairs | 19 |
| From Bits to text | 20 |
| Results..... | 20 |
| Adaptive Arithmetic | 20 |
| Test Case #1 | 20 |
| Test Case #2 | 22 |
| Test Case #3 | 24 |
| Lempel-Ziv | 26 |
| Test Case #1 | 26 |
| Test Case #2 | 28 |
| Test Case #3 | 29 |

| | |
|-------------------------------|----|
| Discussion and Comments | 31 |
| Adaptive Arithmetic | 31 |
| Lempel-Ziv | 32 |

List of Figures

| | |
|--|----|
| Figure 1 : Main Screen | 3 |
| Figure 2 : Encoding Output Example..... | 4 |
| Figure 3 : Decoding Tab Lempel-Ziv | 4 |
| Figure 4 : Decoding Tab Adaptive Arithmetic | 5 |
| Figure 5 : Decoder Output | 5 |
| Figure 6 : Lempel- Ziv Encoding Function | 13 |
| Figure 7 : Getting The Unique Sequences..... | 14 |
| Figure 8 : Getting the (Index, Suffix) Pairs | 15 |
| Figure 9 : Turning The Pairs Matrix Into An Array Of Numbers | 16 |
| Figure 10 : Getting The Number Of Bits That Each Sequence Will Take..... | 16 |
| Figure 11 : Encoding To Bits | 16 |
| Figure 12 : Compression Ratio Code | 17 |
| Figure 13 : Main Decoder Function Lempel-Ziv | 18 |
| Figure 14 : Dividing The Encoded Bit Stream Into Sequences of Variable Lengths | 19 |
| Figure 15 : Retrieving The (Prefix, Suffix) Pairs | 19 |
| Figure 16 : Test Case #1 Lempel-Ziv | 26 |
| Figure 17 :Test Case #2 Lempel-Ziv | 28 |
| Figure 18 : Test Case #3 Lempel-Ziv | 29 |
| Figure 19 : Low number of characters, bad compression ratio | 32 |
| Figure 20 : Five Compressed Paragraphs of Lorem Ipsum..... | 33 |
| Figure 21 : One Compressed Paragraph of Lorem Ipsum | 33 |

User Manual

Encoding

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

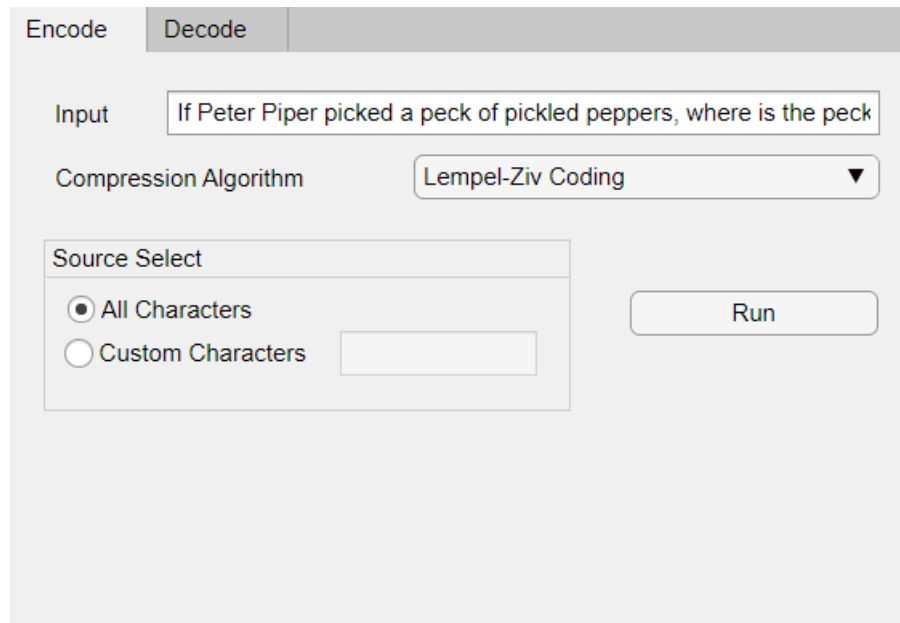


Figure 1 : Main Screen

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

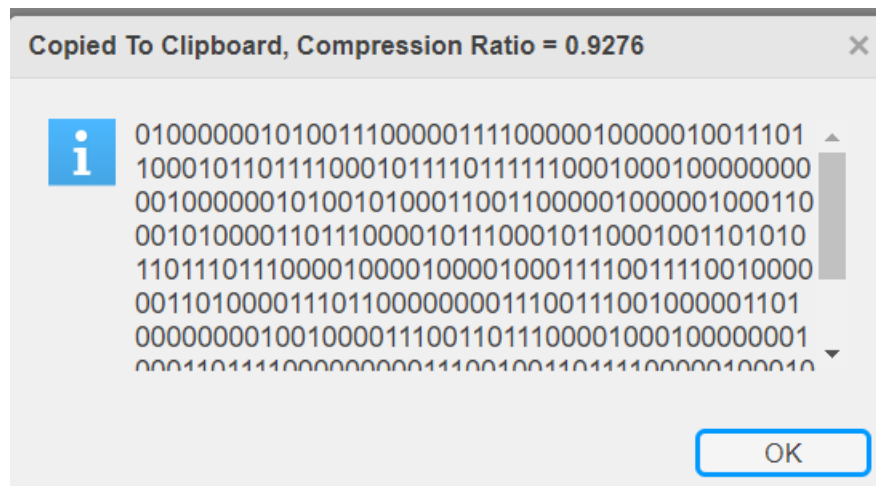


Figure 2 : Encoding Output Example

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:

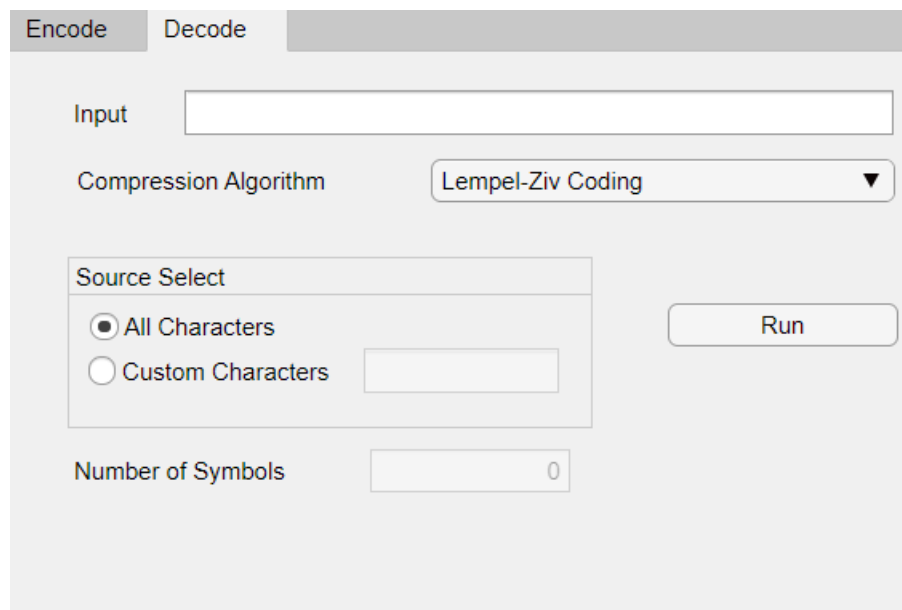


Figure 3 : Decoding Tab | Lempel-Ziv

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.

The image shows a software window with two tabs: 'Encode' and 'Decode'. The 'Decode' tab is active. It contains the following elements:

- An 'Input' text field.
- A 'Compression Algorithm' dropdown menu set to 'Adaptive Arithmetic Coding'.
- A 'Source Select' section with two radio buttons: 'All Characters' (selected) and 'Custom Characters' (unselected). Next to 'Custom Characters' is an empty text field.
- A 'Run' button.
- A 'Number of Symbols' text field with the value '0'.

Figure 4 : 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*.

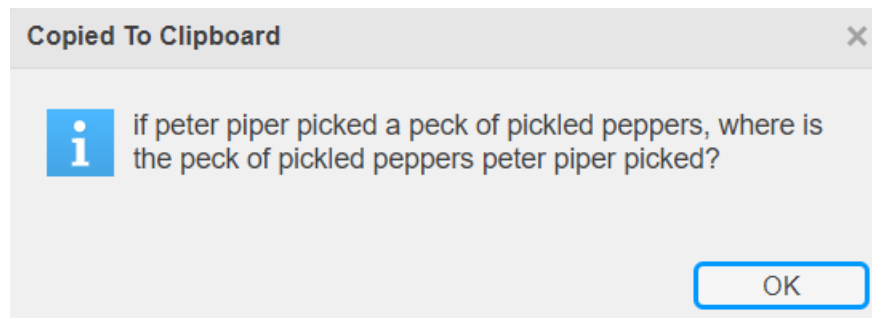


Figure 5 : Decoder Output

Adaptive Arithmetic coding

Encoding

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

```

len = length(str);          % length of the string

u = unique(lower(U_char));  % get unique characters(lowered) from the string

len_unique = length(u);    % length of the unique characters string

p = zeros(1, len_unique);

for i = 1 : len_unique      %equalprob
    p(i) = 1 / len_unique;
end

% Encoder Table
low = 0;
high = 1;
stage = zeros(1, len_unique + 1); %table
stage(1)=low;
stage(length(stage))=high;

for i = 2: length(stage)-1

    stage(i)= stage(i-1)+p(i);    %initial states of the table

end

u = sort(u);
prev_observation = zeros(1, len_unique); %The number of ocurence of each symbol
p_sep = zeros(1, len_unique);          %the probability calculated after each sequence

encoded_str='';

```

Figure: part 1 Adaptive arithmetic Encoding Function

```

for i = 1 : len
    for j = 1 : len_unique

        % if the value from 'str'
        % matches with 'u' then
        if str(i) == u(j)
            prev_observation(j)=prev_observation(j) + 1;    %number of occurrence increased by 1

            for k = 1 : len_unique
                p_sep(k)=(1+prev_observation(k))/(i+len_unique); %the probability calculated after each sequence
            end

            %pos = j;      %for debug

            low = stage(j);
            high = stage(j+1);

            stage(1)=low;
            stage(length(stage))=high;
            for ind = 2: length(stage)-1

                stage(ind)= stage(ind-1)+p_sep(ind-1)*(high - low);    %updating the table

            end

            break
        end
    end
end
end
end

```

Figure: part 2 Adaptive arithmetic Encoding Function


```

pointer=0;
% The same as above, but encode each 10 chars sequentially
pointervalue = 0;
for i = 1 : len

    if (rem((i-1),10)==0 && i~=1)
        pointer =1;
        pointervalue = pointervalue + 1;
        l_of_bin = ceil(log2(1/(high-low)))+1;

        tag = (low + high)/2;

        for i_encode=1:l_of_bin

            tag = tag * 2;

            if tag < 1
                encoded_str = [encoded_str '0'];
            else
                encoded_str = [encoded_str '1'];
                tag = tag - 1;
            end

        end

        end

        low = 0;
        high = 1;
        stage = zeros(1, len_unique + 1);    %table
        stage(1)=low;
        stage(length(stage))=high;

        for i_new = 2: length(stage)-1

            stage(i_new)= stage(i_new-1)+p(i_new);

        end
end

```

Figure: part 3 Adaptive arithmetic Encoding Function

```

% displaying tag value

l_of_bin = ceil(log2(1/(high-low)))+1;

tag = (low + high)/2;

%convert from decimal to binary
for i_encode=1:l_of_bin

    tag = tag * 2;

    if tag < 1
        encoded_str = [encoded_str '0'];
    else
        encoded_str = [encoded_str '1'];
        tag = tag - 1;
    end

end

averg_codelength = length(encoded_str)/len ;
Bits_Fixed_length = (log2(len_unique));
%efficiency
Compression_Ratio = averg_codelength/Bits_Fixed_length ;

```

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:

```
%adaptive arathmatic function decode
function Decoded = Adaptive_Arithmetic_Decode(app,code,len,symbol)

%%decoding
u = sort(lower(symbol));
len_unique = length(u);

indexlooper=50;

for i=1:length(u)
    p(i)=1/len_unique ;
end

%%convert the encoding binary bits to decimal

decoded_str=0;
encoded_str = convertStringsToChars(string(code));
tester = encoded_str;
tester = convertStringsToChars(tester);
Output=char(num2cell(tester));
Output=reshape(str2num(Output),1,[]) ;

low = 0;
high = 1;
stage = zeros(1, len_unique + 1); %table
stage(1)=low;
stage(length(stage))=high;

for i = 2: length(stage)-1

    stage(i)= stage(i-1)+p(i);

end

prev_observation = zeros(1, len_unique);
p_sep = zeros(1, len_unique);

Decoded = '';
```

Figure: part 1 Adaptive arithmetic decoding Function

```

for i = 1 : len

    tag = sum(Output.*2.^(-1:-1:-length(Output)));

    for j = 1 : len_unique

        % If tag value falls between a certain
        % range in lookup table then

        if tag > stage(j) && tag < stage(j+1)

            prev_observation(j)=prev_observation(j) + 1;

            for k = 1 : len_unique

                p_sep(k)=(1+prev_observation(k))/(1+len_unique);

            end

            low = stage(j);
            high = stage(j+1);

            stage(1)=low;
            stage(length(stage))=high;

            for ind = 2: length(stage)-1

                stage(ind)= stage(ind-1)+p_sep(ind-1)*(high - low);

            end

            % Getting the matched tag
            % value character

            decoded_str = u(j);
            Decoded = [Decoded decoded_str];

```

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

```
function [coded, compRatio]= lempelZivEnco(app, src, charSrc)
    % Get the unique sequences:
    seqs = getUniqueSeq(app, src);

    % Get the Index/suffix pairs
    pairs = getIndexSuffixPair(app, seqs, charSrc);

    % Append both parts of the pair
    shiftFactor = 2^ceil(log2(numel(charSrc)));
    pairAppend = shiftFactor * pairs(:,1) + pairs(:,2);

    % Get the number of bits required for each pair
    maxIndex = 0:length(seqs)-1;
    noBits = ceil(log2(maxIndex * shiftFactor + numel(charSrc)));

    for idx = 1:length(seqs)
        currentSeqNum = dec2bin(pairAppend(idx));
        codedArray(idx) = convertCharsToStrings(pad(currentSeqNum, noBits(idx), 'left','0'));
    end

    coded = strjoin(codedArray);
    coded = strep(coded, ' ', "");

    codedBitsPerSymbol = strlength(coded)/numel(src);
    fixedBitsPerSymbol = log2(shiftFactor);
    compRatio = codedBitsPerSymbol/fixedBitsPerSymbol;
```

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.

```
function dict = getUniqueSeq(~, src)
    % Turns our input string into an array of unique sequences.
    dict = []; % This will keep hold of our unique sequences.
    letterIDX = 1;
    while letterIDX <= length(src)

        seqEndIDX = letterIDX; % Init

        % We use cell2mat because extractBetween returns a cell
        % array instead of a normal char vector.
        seq = cell2mat(extractBetween(src, letterIDX, seqEndIDX));

        while ismember(seq, dict)
            if(seqEndIDX == length(src))
                break
            end
            seqEndIDX = seqEndIDX + 1;
            seq = cell2mat(extractBetween(src, letterIDX, seqEndIDX));
        end

        dict = [dict, convertCharsToStrings(seq)]; %#ok<*AGROW>
        letterIDX = seqEndIDX+1;
    end
end
```

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

```

function pairs = getIndexSuffixPair(~, seqArr, symbols)
    pairs = zeros(length(seqArr), 2);
    index = zeros(1, length(seqArr));
    suffix = zeros(1, length(seqArr));
    for idx = 1:length(seqArr)

        seq = char(seqArr(idx));
        suffix(idx) = seq(end);

        % We need to find the location of the prefix in our
        % dictionary.

        if strlength(seq) == 1
            prefix = 0; %#ok<NASGU>
            index(idx) = 0;
        else
            prefix = convertCharsToStrings(seq(1:end-1));
            foundAt = find(seqArr == prefix);
            index(idx) = foundAt(1);
        end

        % When we try to find a letter in the uniques array
        % we get an integer.
        suffix(idx) = strfind(symbols, suffix(idx))-1;

        pairs(idx, 1) = index(idx);
        pairs(idx, 2) = suffix(idx);
    end
end

```

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 $n \times 2$ 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

```
% Append both parts of the pair
shiftFactor = 2^ceil(log2(numel(charSrc)));
pairAppend = shiftFactor * pairs(:,1) + pairs(:,2);
```

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

```
% Get the number of bits required for each pair
maxIndex = 0:length(seqs)-1;
noBits = ceil(log2(maxIndex * shiftFactor + numel(charSrc)));
```

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

```
for idx = 1:length(seqs)
    currentSeqNum = dec2bin(pairAppend(idx));
    codedArray(idx) = convertCharsToStrings(pad(currentSeqNum, noBits(idx), 'left', '0'));
end

coded = strjoin(codedArray);
coded = strrep(coded, ' ', '');
```

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

```
codedBitsPerSymbol = strlen(coded)/numel(src);  
fixedBitsPerSymbol = log2(shiftFactor);  
compRatio = codedBitsPerSymbol/fixedBitsPerSymbol;
```

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

```

function text = lempelZivDeco(app, src, charSrc)

    % Calculate maxPairs
    shiftFactor = 2^ceil(log2(numel(charSrc)));
    maxY = numel(charSrc);
    maxX = 0;
    maxBits = [];

    idx = 1;
    maxBitsIdx = 1;

    while idx <= length(src)
        % Calculate the word length
        wordLength = ceil(log2(shiftFactor*maxX + maxY));

        maxBits(maxBitsIdx) = wordLength;
        maxBitsIdx = maxBitsIdx+1;
        idx = idx + wordLength;
        maxX = maxX+1;
    end

    % Now maxBits has the number of bits that represent each
    % index/suffix pair. we now need to group the bits into
    % codewords, then decode the code words into index/suffix
    % pairs.

    codeWords = getCodeWord(app, src, maxBits);
    [index, suffix] = decodeIndexSuffixPairs(app, codeWords, charSrc);

    seqArr = strings(1, numel(codeWords));

    for idx = 1:numel(seqArr)
        if index(idx) == 0
            seqArr(idx) = charSrc(suffix(idx)+1);
        else
            seqArr(idx) = seqArr(index(idx))+charSrc(suffix(idx)+1);
        end
    end

    text = strjoin(seqArr, '\0');
end

```

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.

```
function codewords = getCodeword(~, src, noOfBitsArr)
    codewords = strings(1, numel(noOfBitsArr));

    idx = 1;
    iterator = 1;
    while idx <= length(src)
        codewords(iterator) = convertCharsToStrings(cell2mat(extractBetween(src, idx, idx-1+noOfBitsArr(iterator))));
        idx = idx + noOfBitsArr(iterator);
        iterator = iterator+1;
    end
end
```

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

Get the Prefix/Suffix Pairs

```
function [indexArr, suffixArr] = decodeIndexSuffixPairs(~, codewords, charSrc)
    % First, find how many bits does the suffix take up
    suffixLen = ceil(log2(numel(charSrc)));

    % Then, split the code words using suffixLen into two parts
    for idx = 1:numel(codewords)
        codeword = codewords(idx);
        splitIndex = strlength(codeword) - suffixLen;

        if splitIndex == 0
            indexArr(idx) = "0";
            suffixArr(idx) = codeword;
            continue
        end

        indexArr(idx) = convertCharsToStrings(codeword{1}(1:splitIndex));
        suffixArr(idx) = convertCharsToStrings(codeword{1}(splitIndex+1:end));

    end
    indexArr = bin2dec(indexArr);
    suffixArr = bin2dec(suffixArr);
end
```

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

```
for idx = 1:numel(seqArr)
    if index(idx) == 0
        seqArr(idx) = charSrc(suffix(idx)+1);
    else
        seqArr(idx) = seqArr(index(idx))+charSrc(suffix(idx)+1);
    end
end

text = strjoin(seqArr, '\0');
```

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 its respected suffix, appending the result of each index as we iterate.

Results

Adaptive Arithmetic

Test Case #1

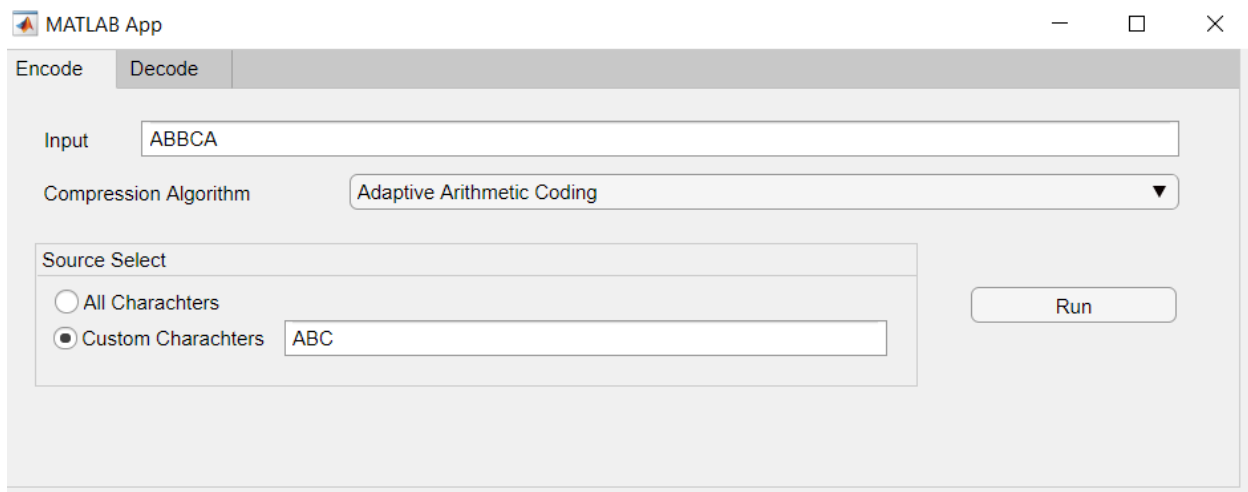
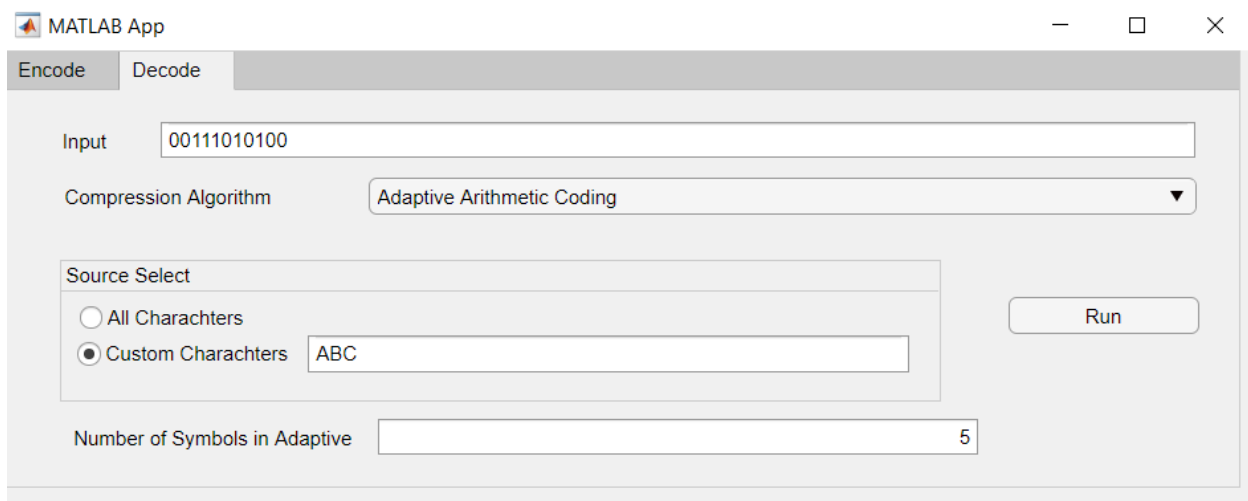
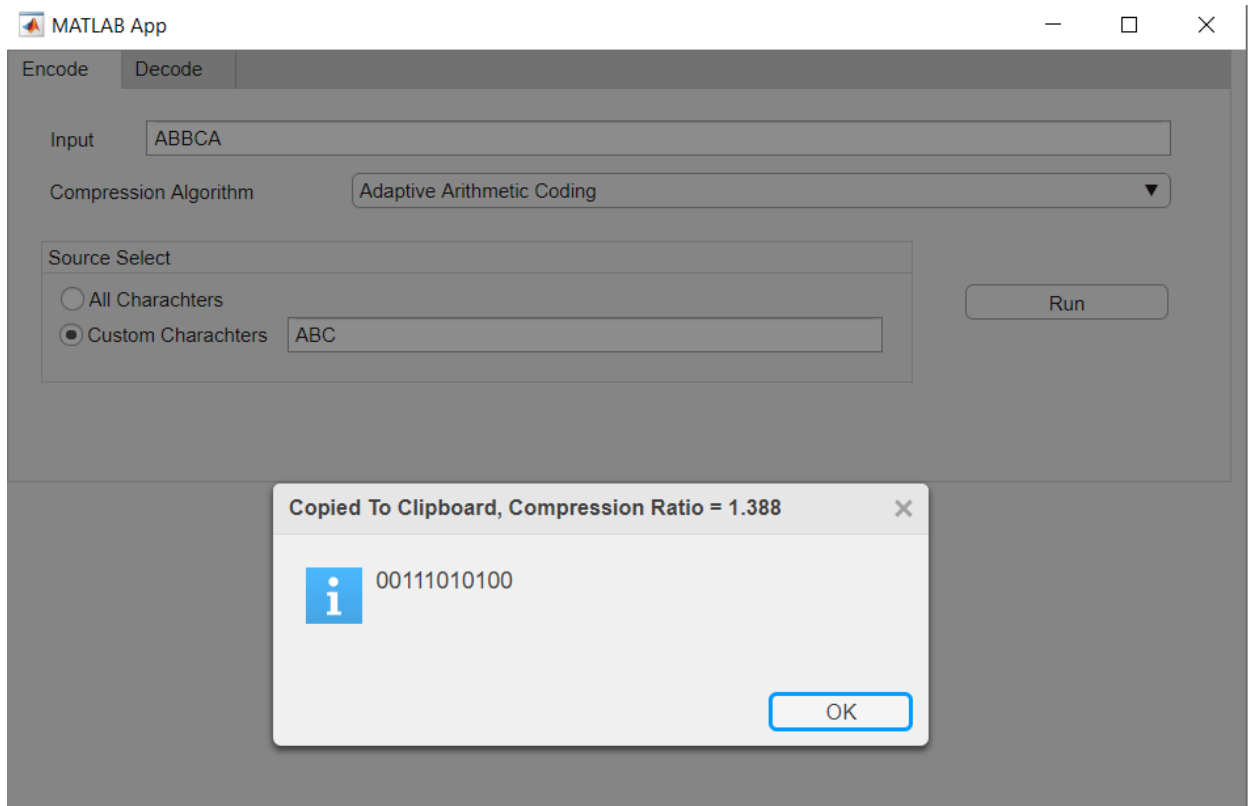
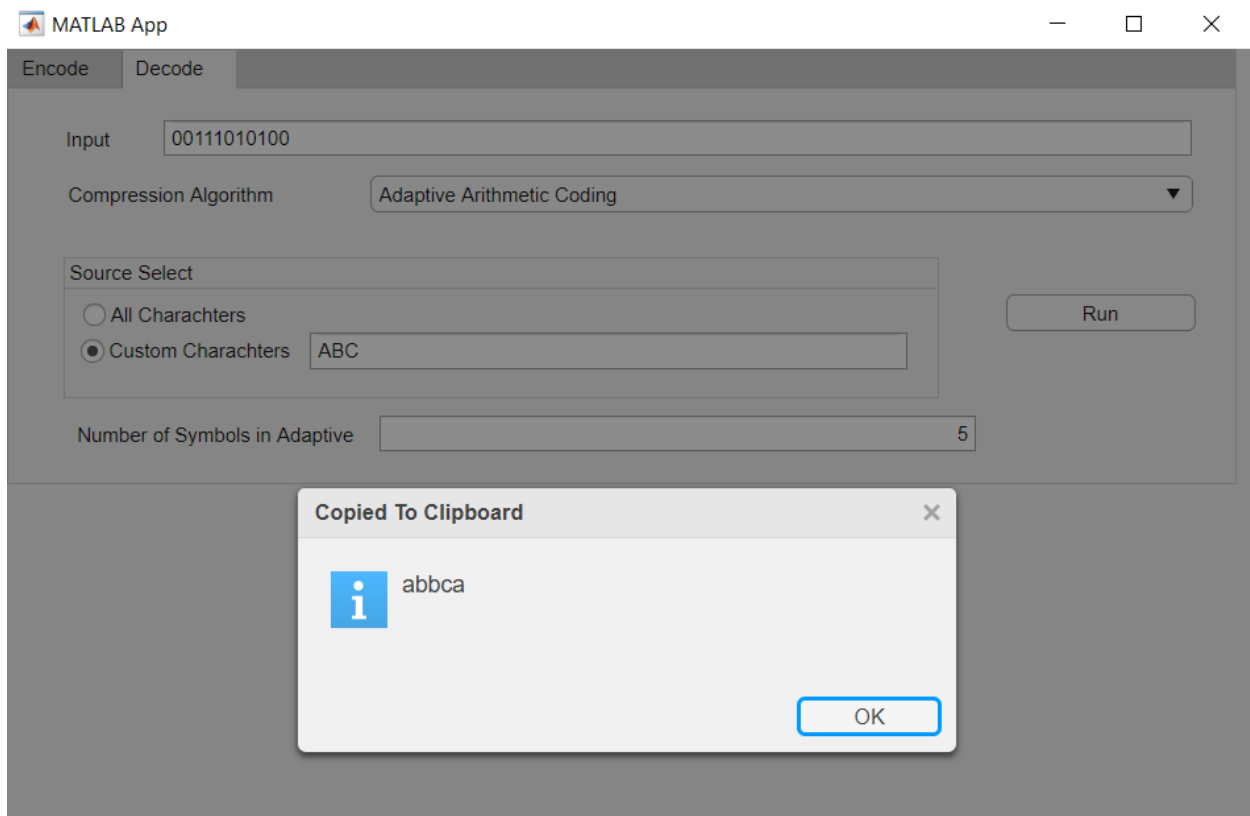
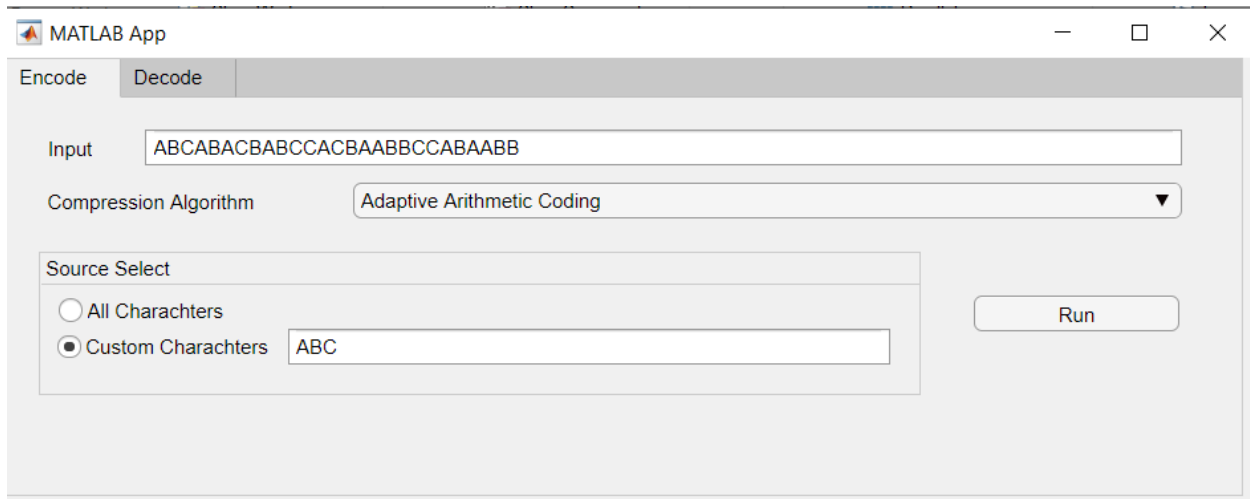


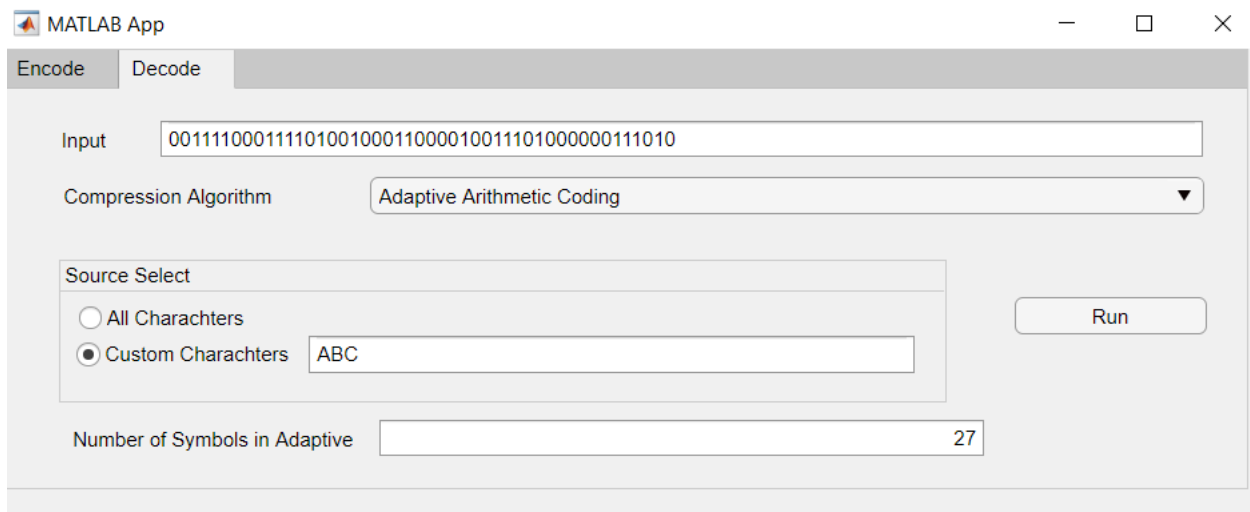
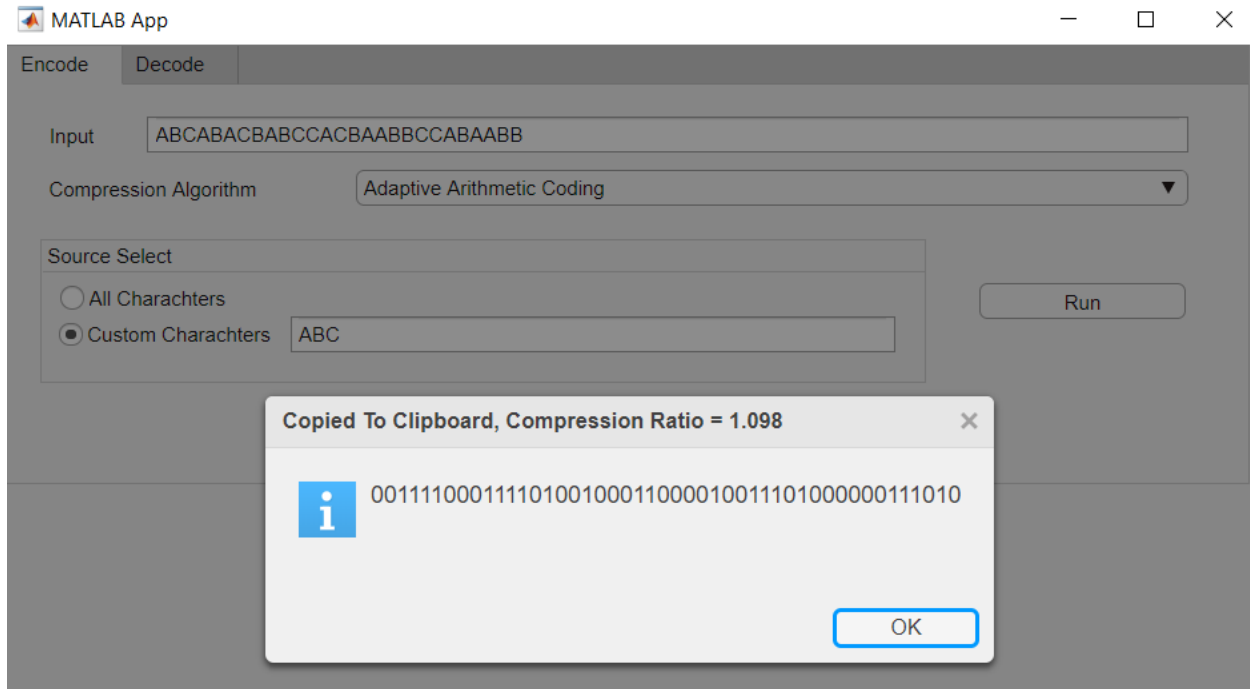
Figure : Test Case #1 | Adaptive Arithmetic

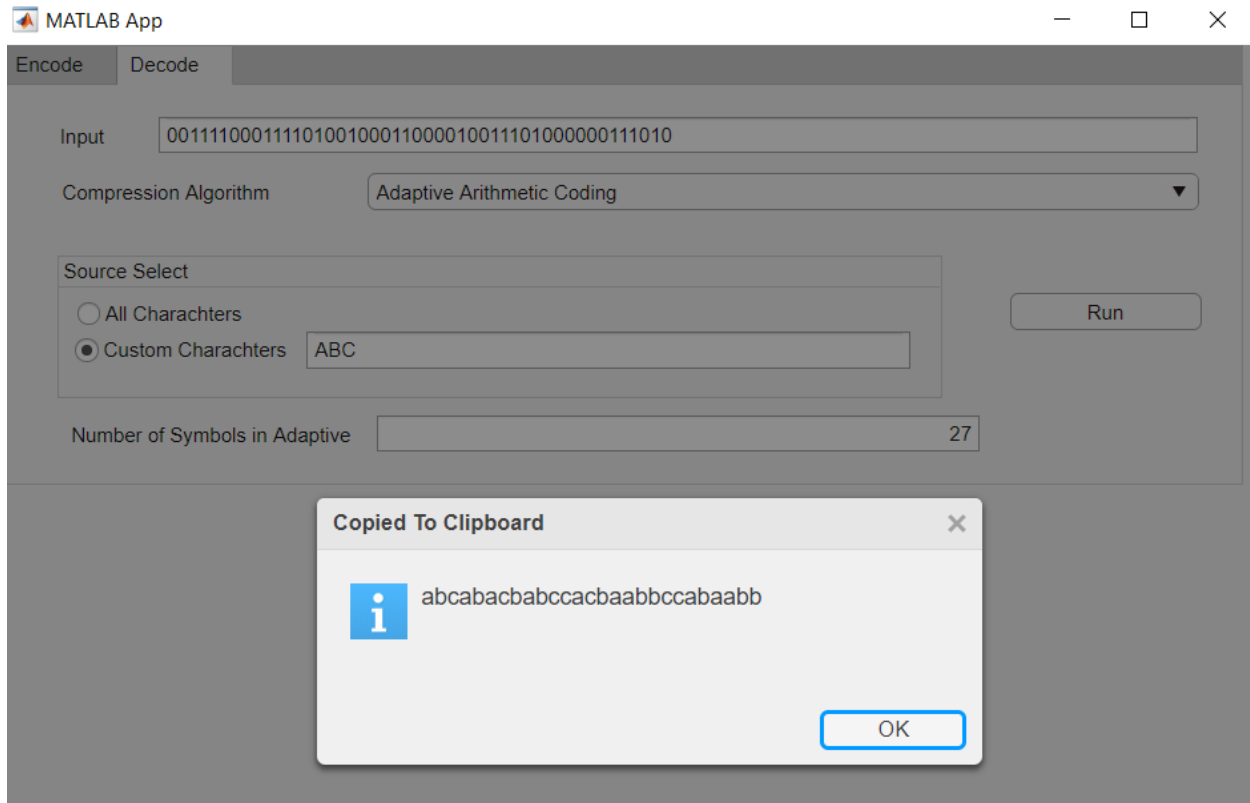




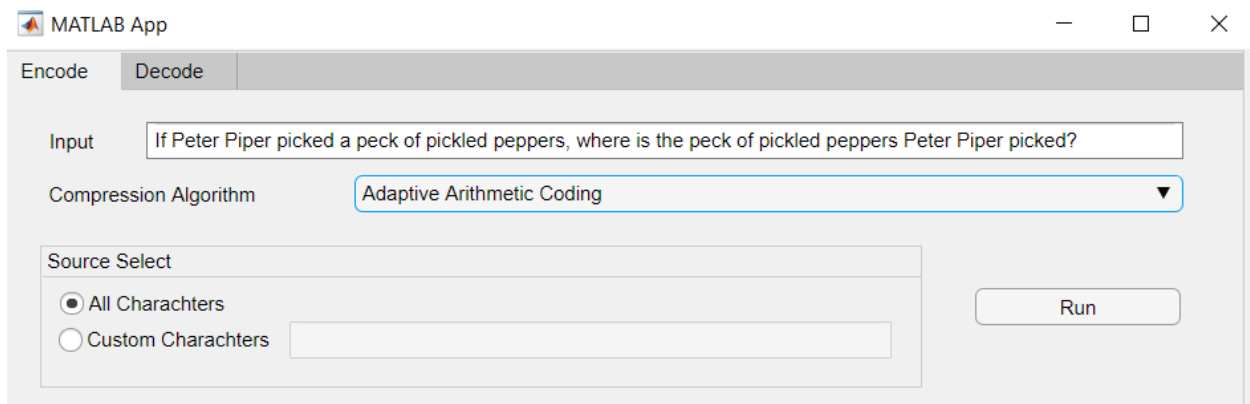
Test Case #2

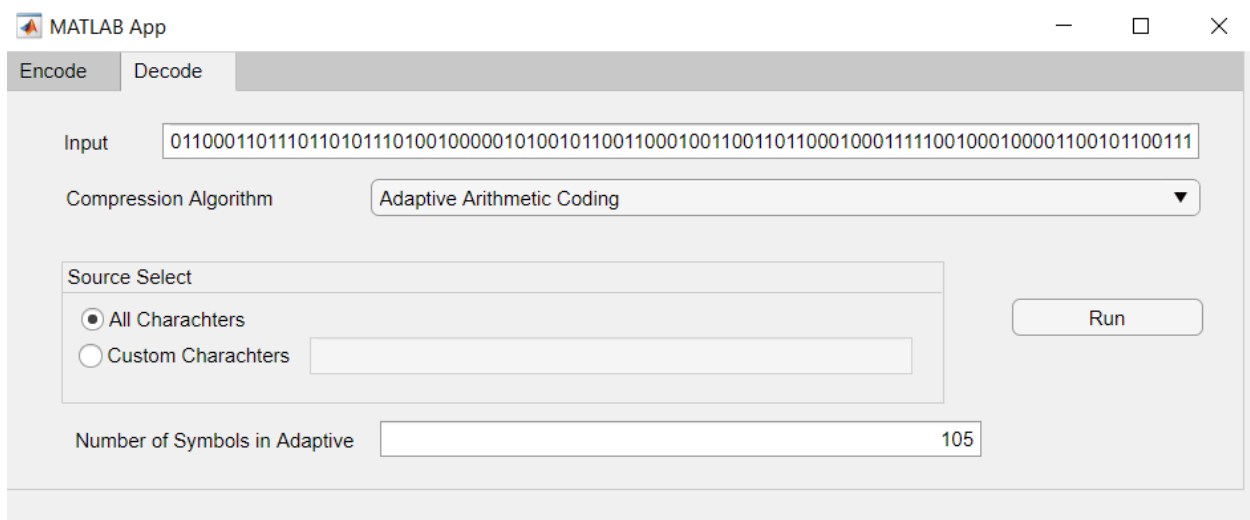
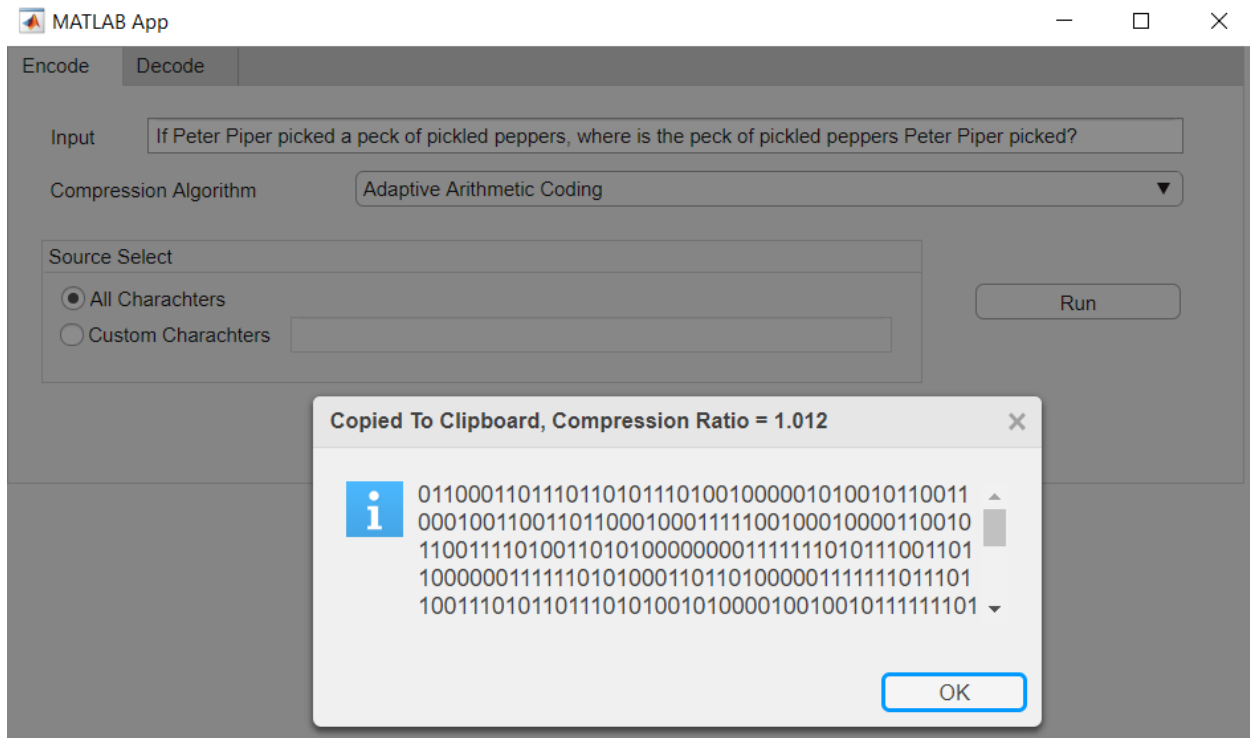
*Figure : Test Case #2 | Adaptive Arithmetic*

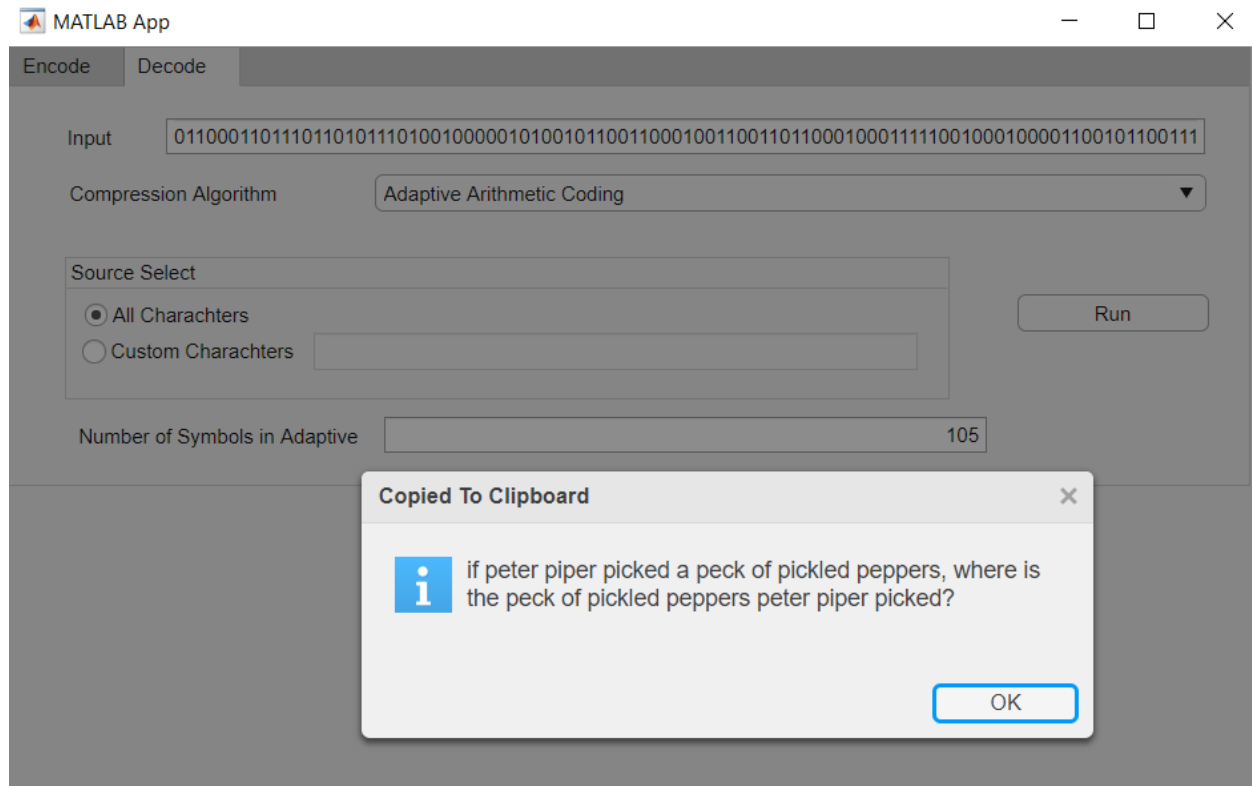




Test Case #3

*Figure : Test Case #3 | Adaptive Arithmetic*





Lempel-Ziv

Test Case #1

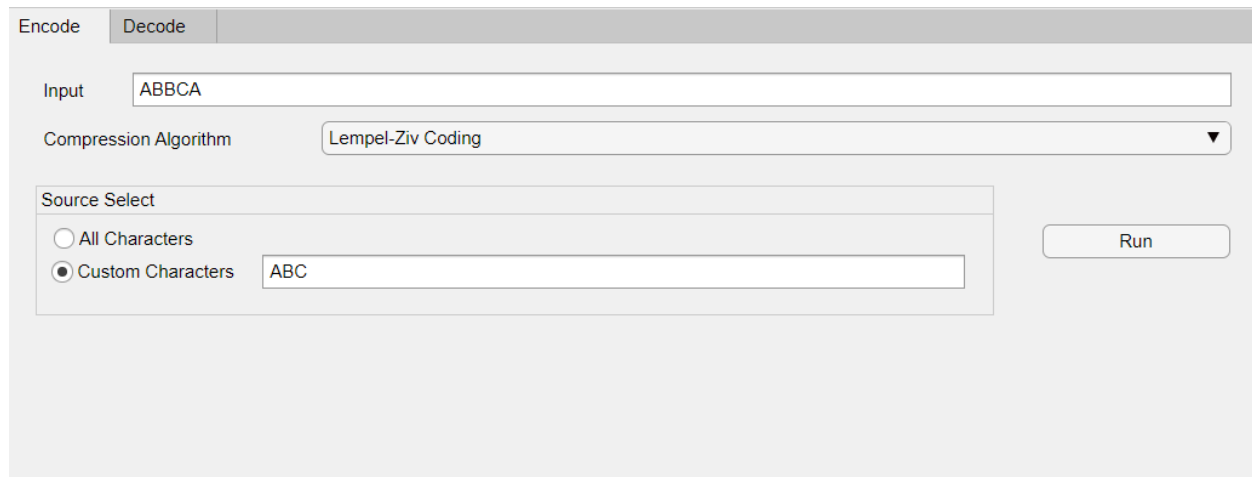
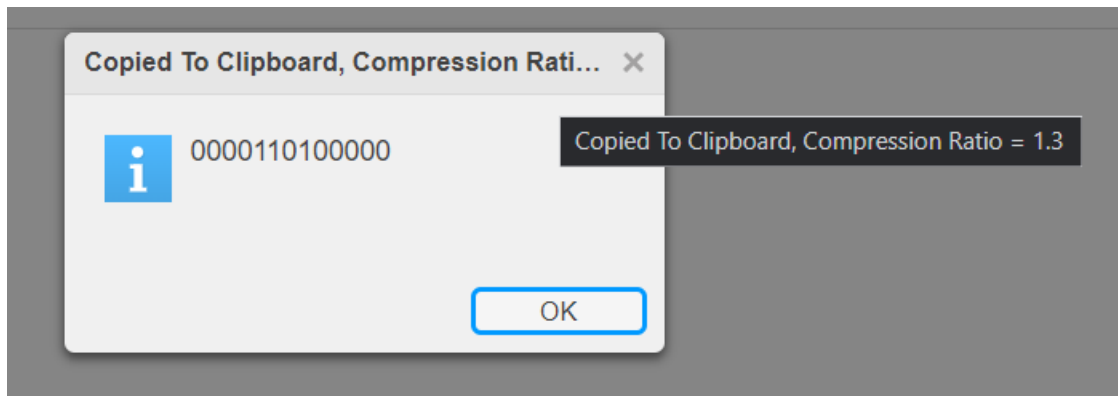


Figure 16 : Test Case #1 | Lempel-Ziv



Encode Decode

Input: 0000110100000

Compression Algorithm: Lempel-Ziv Coding ▼

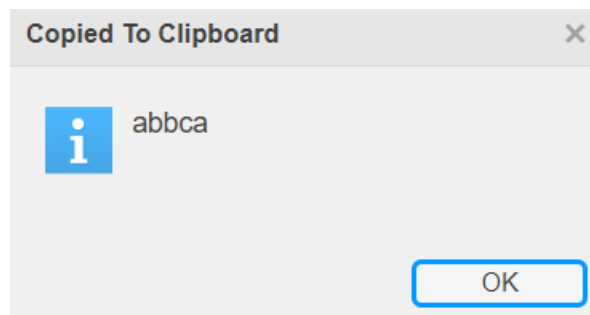
Source Select

☐ All Characters

☒ Custom Characters: ABC

Number of Symbols: 0

Run



Test Case #2

Encode Decode

Input

Compression Algorithm

Source Select

☐ All Characters

☒ Custom Characters

Run

Figure 17 :Test Case #2 | Lempel-Ziv

Encode Decode

Input

Compression Algorithm

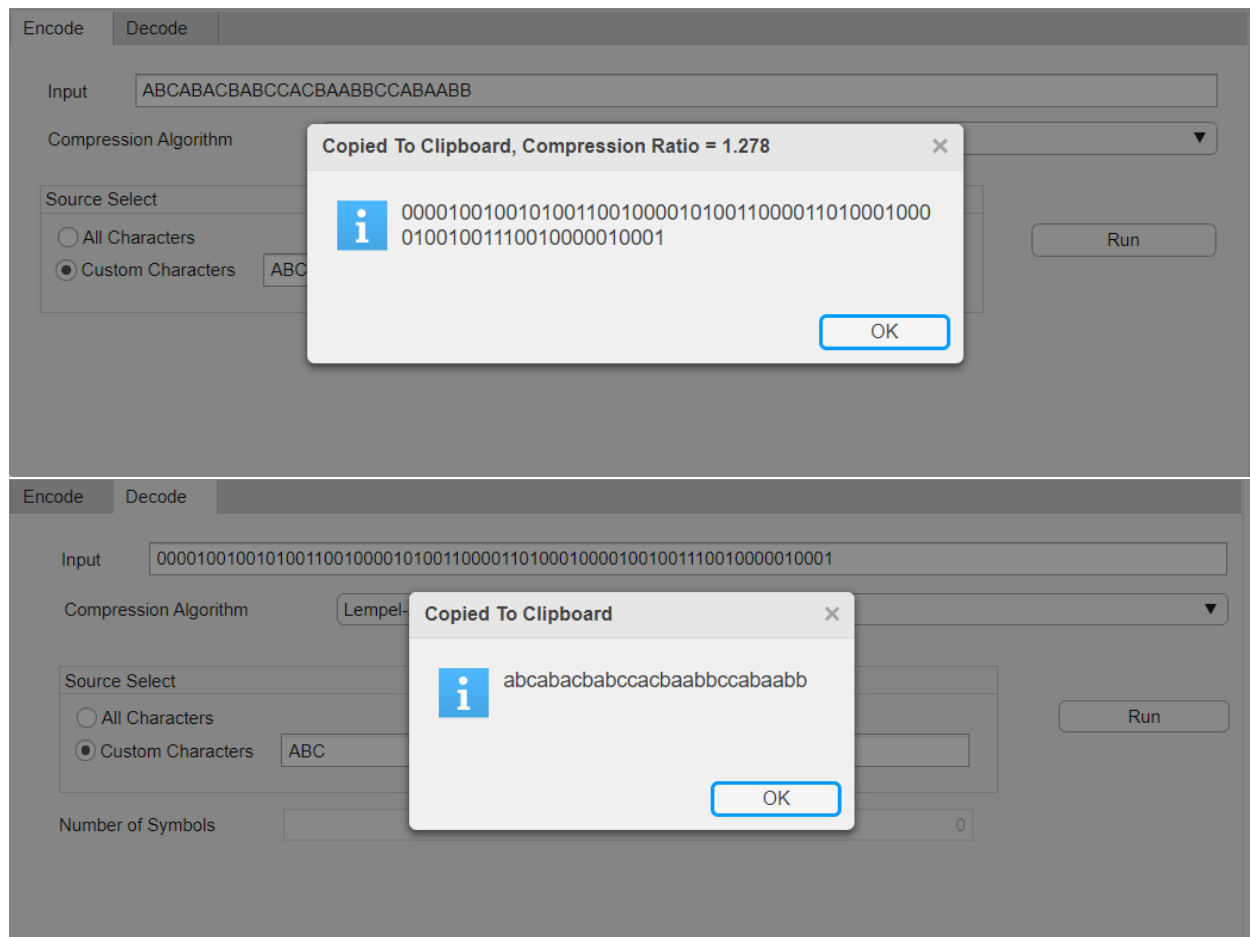
Source Select

☐ All Characters

☒ Custom Characters

Run

Number of Symbols



Test Case #3

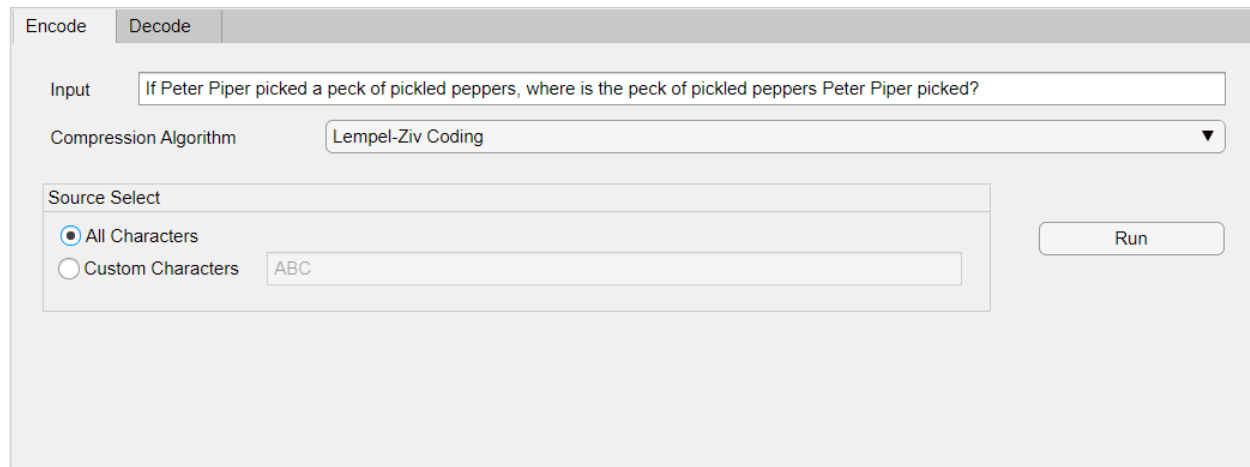
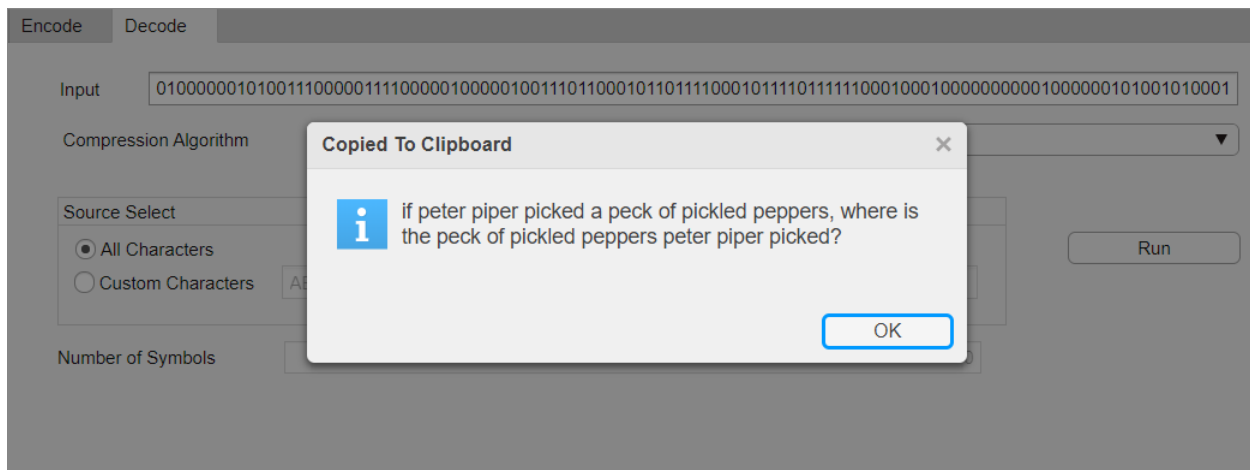
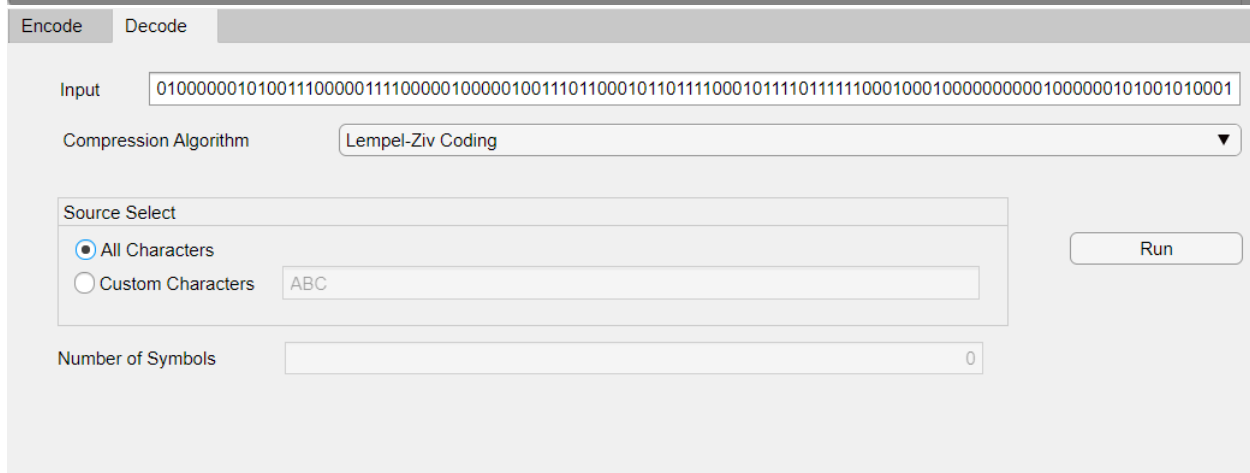
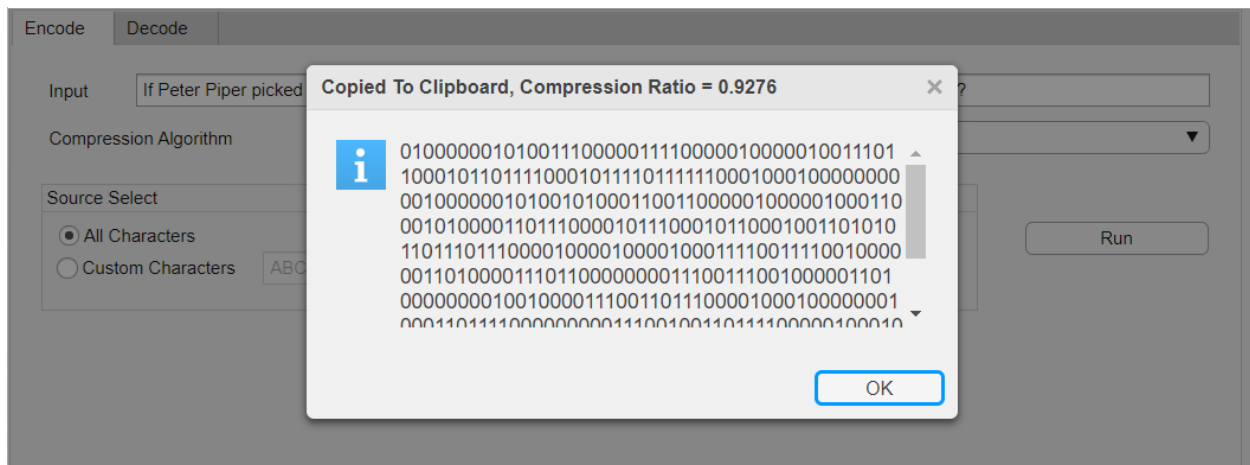


Figure 18 : Test Case #3 | Lempel-Ziv



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:

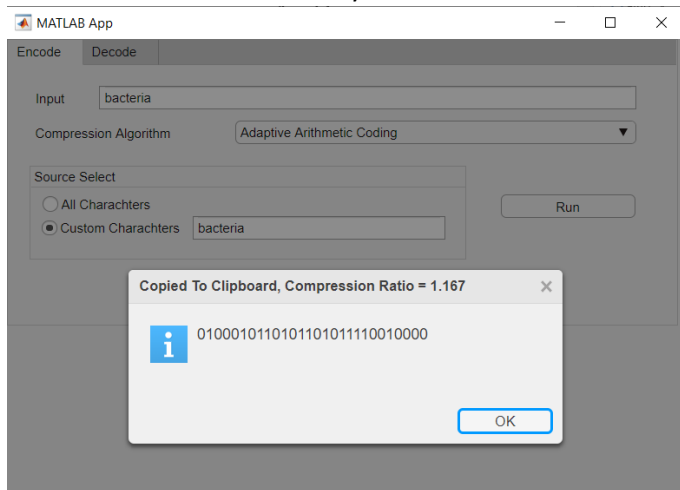
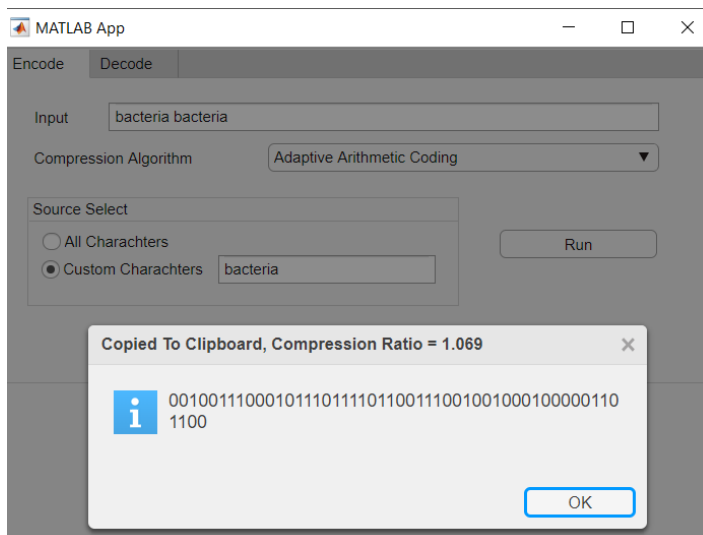


Figure : Low number of characters, Good compression ratio

- 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:

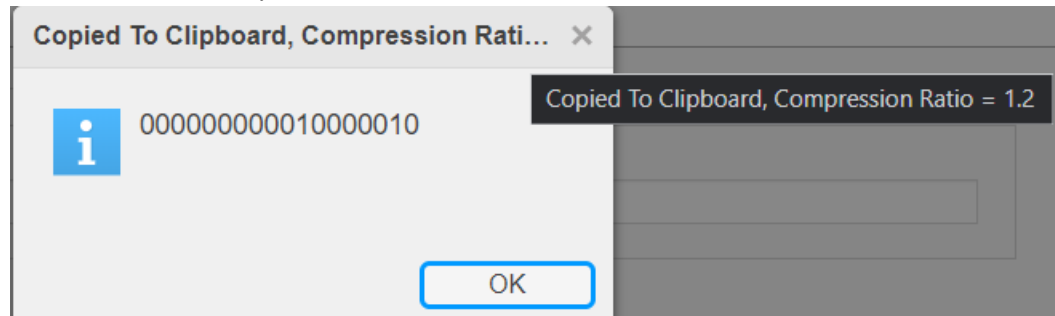
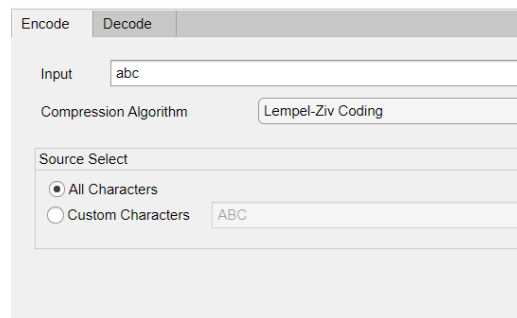
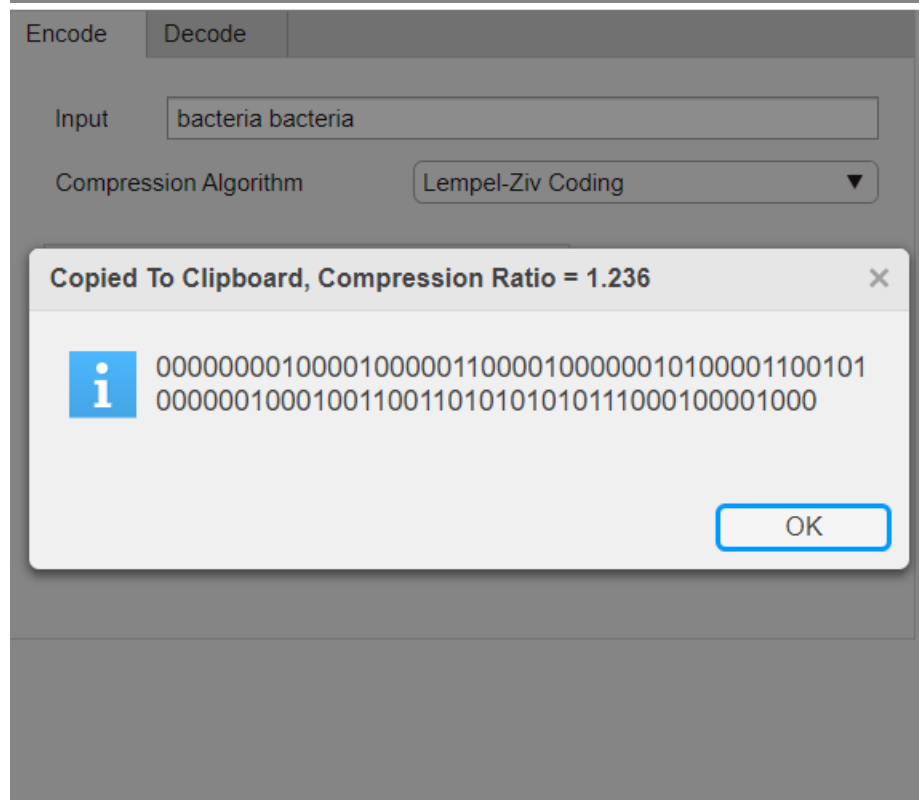
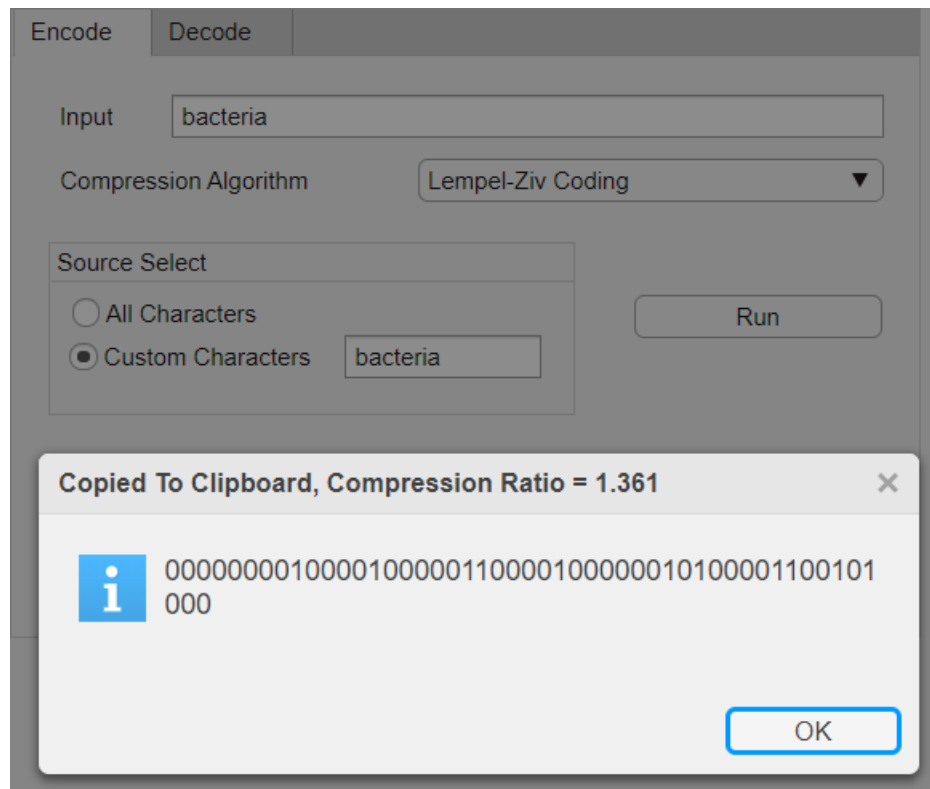
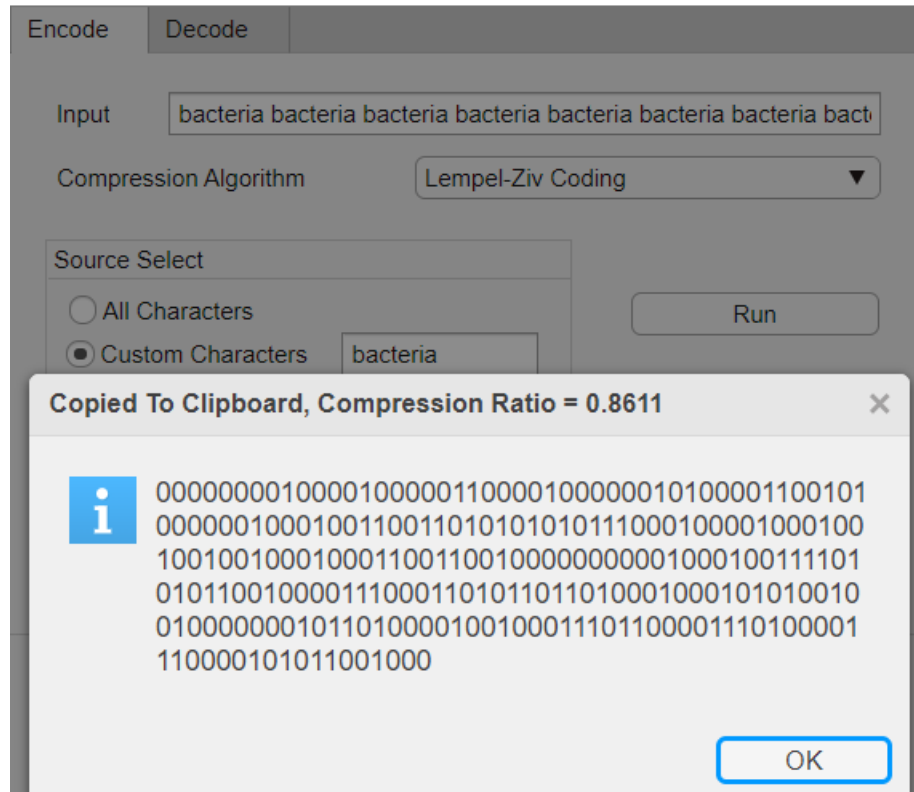


Figure 19 : Low number of characters, bad compression ratio



- 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

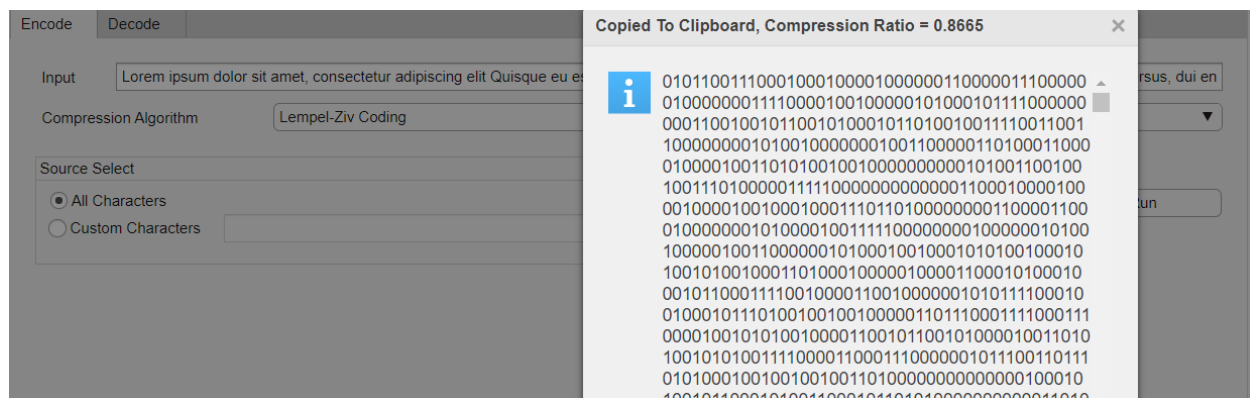


Figure 20 : Five Compressed Paragraphs of Lorem Ipsum

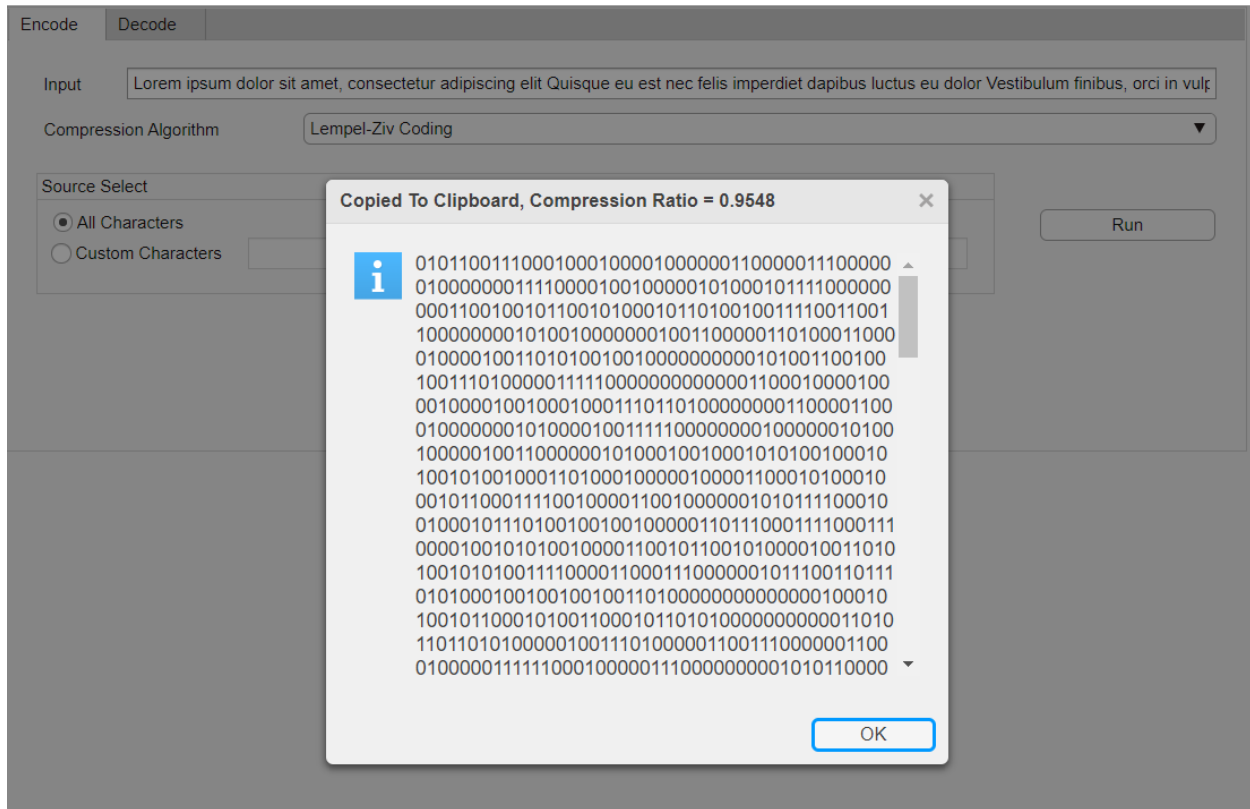


Figure 21 : One Compressed Paragraph of Lorem Ipsum

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.