Overview

To create a GUI Java application that allowed experimentation of Huffman and Arithmetic encodings of user input information sources.

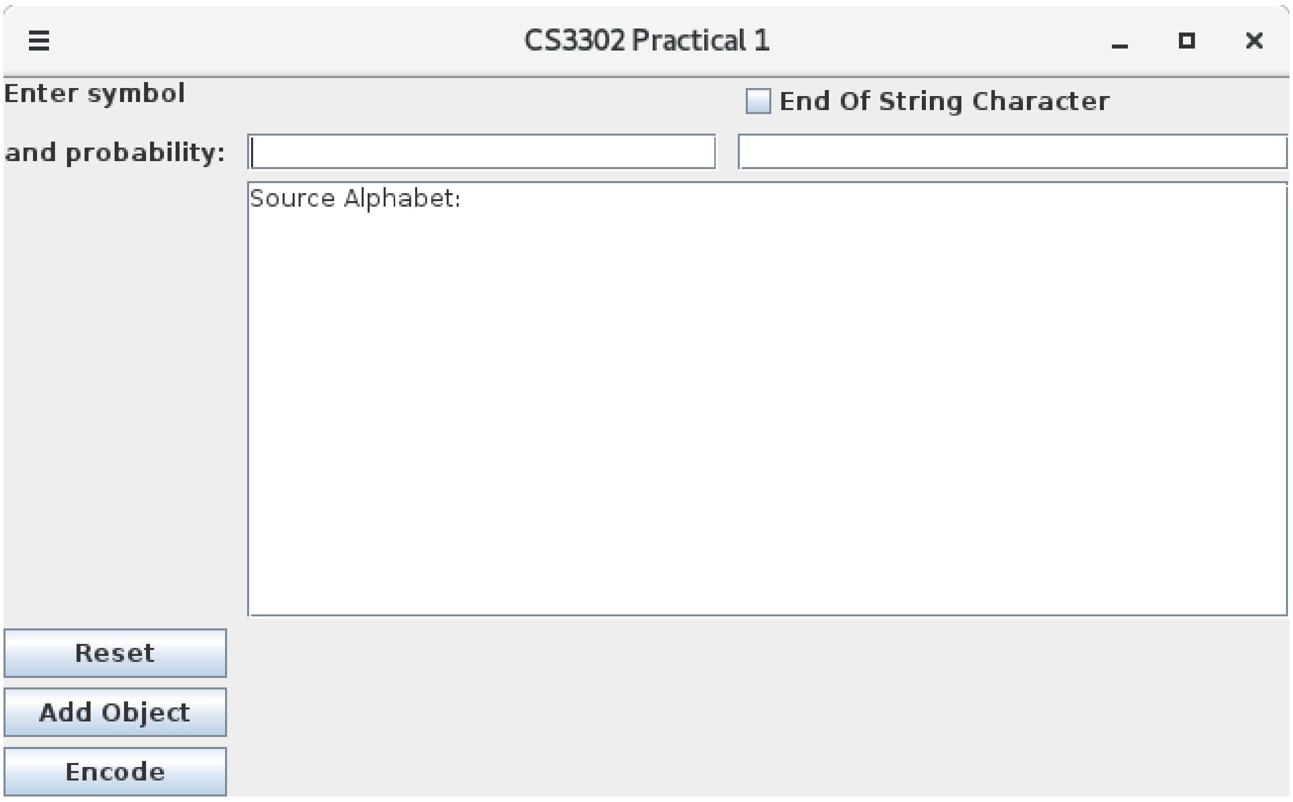
Build Instructions

To run the program, simply run these commands in the terminal:

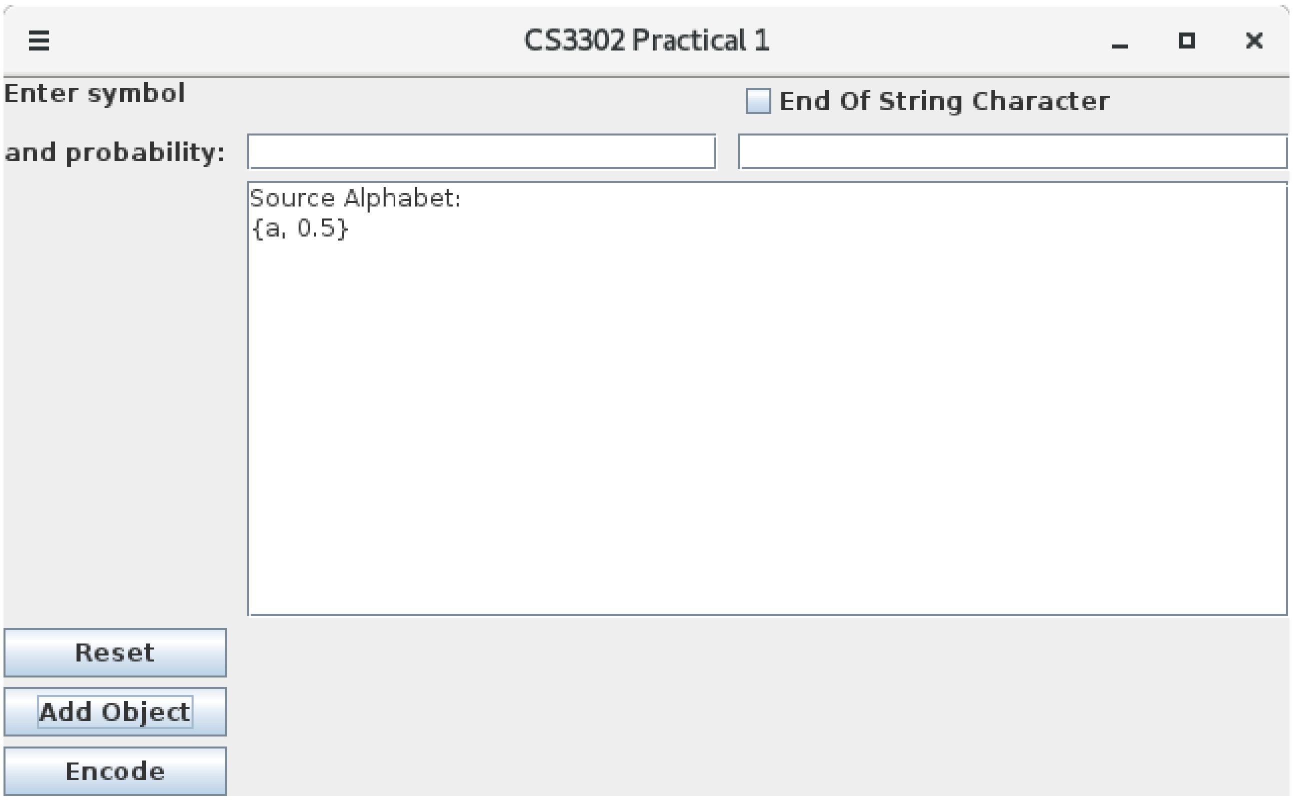
cd out/artifacts/Practical1_jar

java -jar Practical1.jar

Design and Implementation

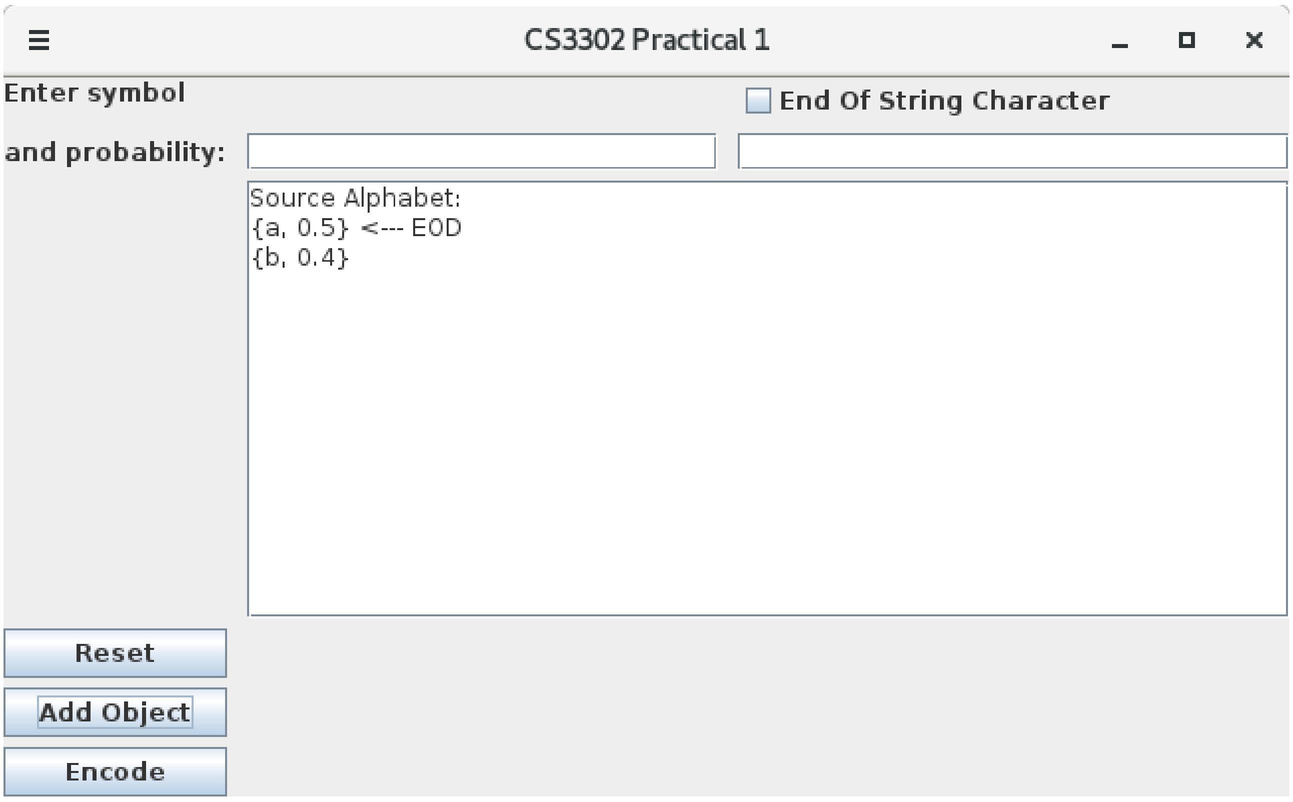


This is what the application looks like upon start-up. The two white boxes on the top are where the user inputs the information source. The left box takes the chosen symbol, the right box takes the respective probability. The large white box underneath is where data and responses are output.



```
Source Alphabet:
{a, 0.5}
```

For example, user has input “a” as the symbol, and the probability as “0.5”. Provided this input passes all the input validation, this is then added to the source alphabet.



```
Source Alphabet:
{a, 0.5} <--- EOD
{b, 0.4}
```

If the data is entered with the “End Of String Character” box selected, this symbol is marked as the EOD character, as can be seen here.

A symbol input is valid if it hasn’t already been entered into the source alphabet and it is only one character long. Suitable exceptions are thrown if either of these checks fail, displaying appropriate error messages.

A probability input is also validated. They are checked to ensure that the input variable is indeed a decimal and that its value is less than 1. The sum of all entered probabilities is also tracked, ensuring that the sum does not exceed 1.

The input values are read when the user clicks the “Add Object” button. This will add the values to the information sources of the encoders.

When the “Encode” button is pressed, a random string is generated based on the source alphabet (excluding the chosen EOD character, this is appended onto the end of the random string). If no EOD character has been selected at the point of encoding, the system assigns the most improbable character as the EOD character. To see the benefit of “tertiary” huffman encoding compared to binary huffman encoding, the minimum size of the source alphabet is set to three.

If at any point the user screws up, a “Reset” button is provided, allowing the system to be set back to the beginning, clearing the source alphabet and resetting the encoders.

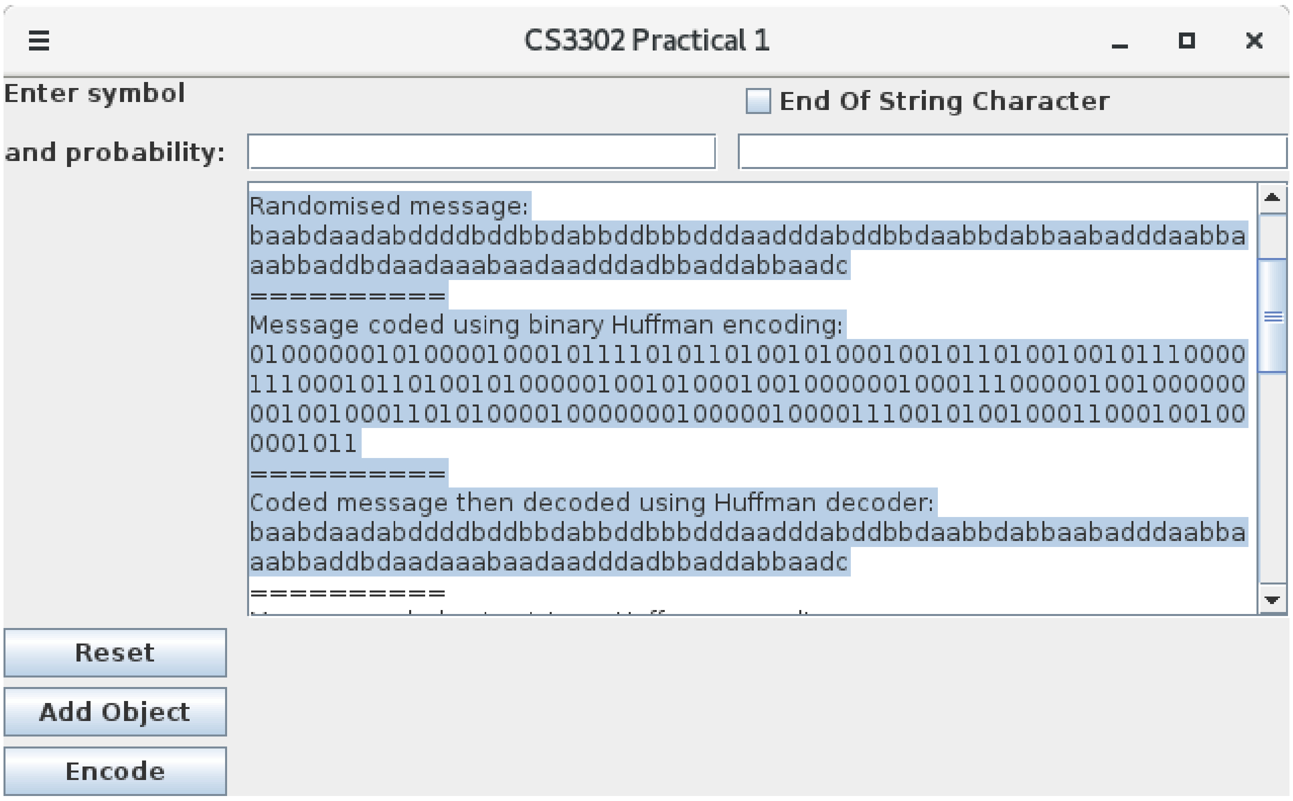
Huffman

Originally, the huffman encoding was performed via the use of a binary tree, with each node of the code tree being represented by an instance of the HuffmanObject class. This would store the symbol, probability, coded value, left child and right child. The left and right child would be the two previous nodes that were concatenated together to build the new object (this is how the tree is originally built).

However, when it came to extending the huffman encoder to using coding alphabets larger than 2, it was clear that the amount of children a node was going to have would vary, relative to the specified size of the code alphabet. Thus, the two children attributes were replaced with a list that would store each child node used to build the parent. This provided the functionality to build and explore a multi-branched tree.

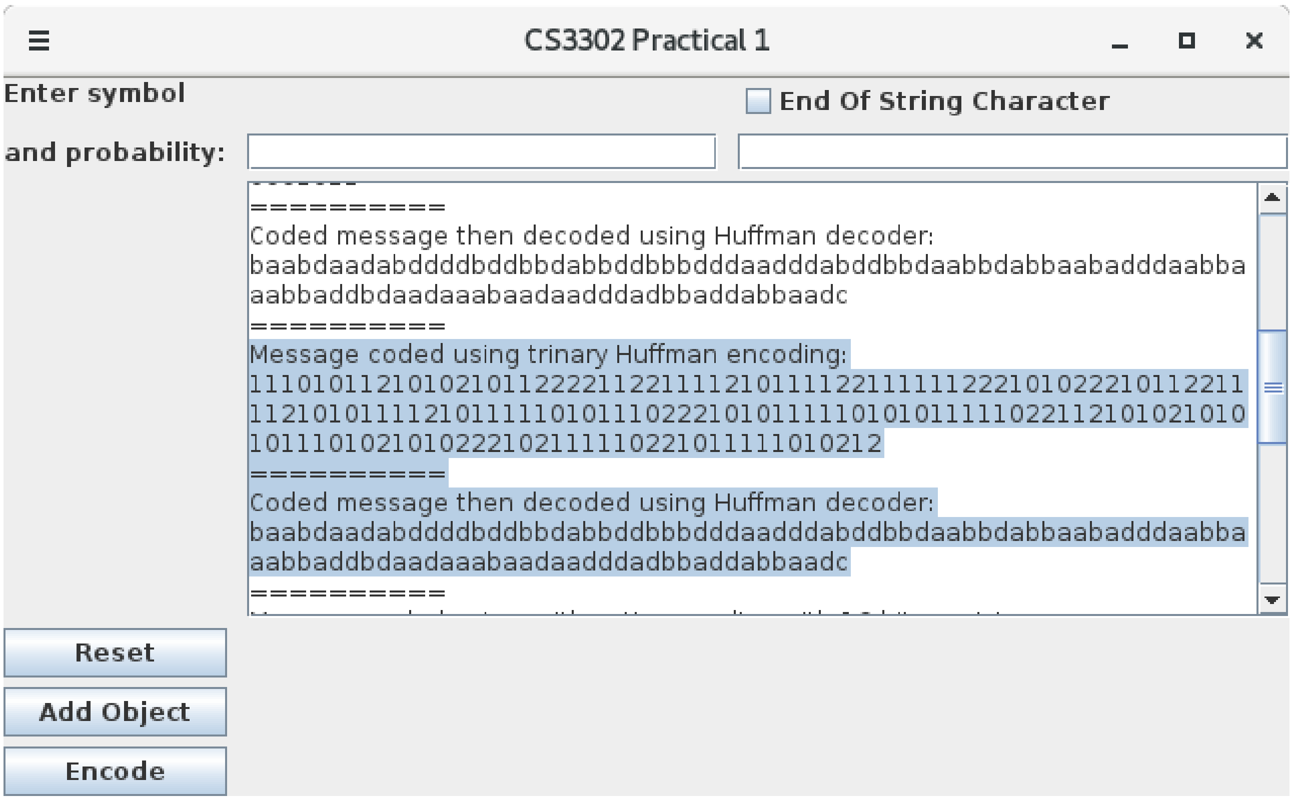
The encoder itself uses recursive methods to build the code tree, inspecting each child node and appending a code digit as it goes, until it finds a leaf node. The leaf nodes represent the original source alphabet, and so the encoding for that symbol is now complete (the base case checks whether or not a node has any children. If no children are present, you’ve found a leaf).

Recursive methods are also used search the tree. These are used in actually encoding the given message (retrieving and substituting the coded value of the current character in the string) and in decoding the encoded string. Since the huffman code produced by the program is instantaneous, the decoder can simply take the first character of the encoded string, search the tree to see if any leaves have that code value, and if not simply append the next character and try again. Eventually, you’ll find a matching symbol for your code value.



```
Randomised message:
baabdaadabddddbddbbdabbddbbbdddaadddabddbbdaabbdabbaabadddaabba
aabbaddbdaadaaabaadaadddadbbaddabbaadc
===========
Message coded using binary Huffman encoding:
0100000010100001000101111010110100101000100101101001001011110000
1110001011010010100000100101000100100000010001110000010010000000
0010010001101010000100000001000001000011100101001000110001001000
0001011
===========
Coded message then decoded using Huffman decoder:
baabdaadabddddbddbbdabbddbbbdddaadddabddbbdaabbdabbaabadddaabba
aabbaddbdaadaaabaadaadddadbbaddabbaadc
===========
```

This shows the generated message, the binary huffman encoding and decoding. “c” had the lowest probability in the source alphabet, so it was automatically assigned as the EOD character, as can be seen in the random string.



```
===========
Coded message then decoded using Huffman decoder:
baabdaadabddddbddbbdabbddbbbdddaadddabddbbdaabbdabbaabadddaabba
aabbaddbdaadaaabaadaadddadbbaddabbaadc
===========
Message coded using trinary Huffman encoding:
1110101121010210112222112211112101111221111112221010222101122 11
1121010111121011111010111022210101111101010111110221121010210 10
10111010210102221021111102210111111010212
===========
Coded message then decoded using Huffman decoder:
baabdaadabddddbddbbdabbddbbbdddaadddabddbbdaabbdabbaabadddaabba
aabbaddbdaadaaabaadaadddadbbaddabbaadc
===========
```

This is the same message being encoded and decoded with the tertiary huffman encoding.

Arithmetic

Some methods within the arithmetic encoder were also required for the decoding (mainly the case checks for rescaling). With this in mind, I decided to create an abstract class “Arithmetic” that implements the rescale conditionals, as well as abstract methods for the actual rescaling methods (as the rescale methods are slightly different for encoding and decoding). Since encoding and decoding use different implementations of the same methods, the encoding and decoding have to be done within different classes.

I tried to use BigDecimal as much as possible to ensure not over or underflow occurred whilst doing arithmetic with double types.

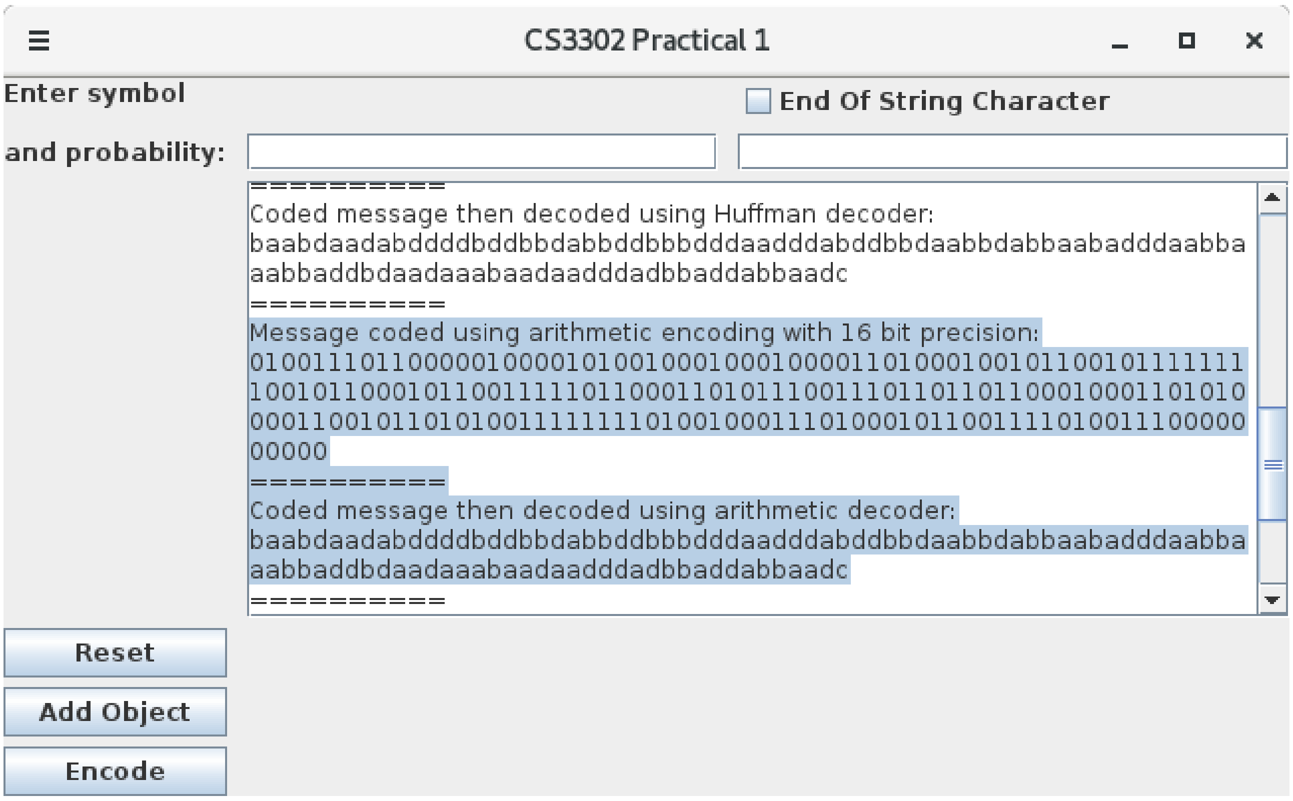
For encoding, the encoder simply iterates through each character of the given string, updating the upper and lower boundary values as required, before checking whether rescaling was necessary. When assigning the values of the upper and lower boundaries, I cast the double value of the BigDecimal as an int, causing the double to be floored (always being rounded down). This method of rounding provides the correct boundary values.

These boundary values are then converted to strings of bits. If and when rescaling occurs, these bit strings are then manipulated as stated by the rescaling algorithm, and then the new bit strings are parsed back into integer values, providing the new upper and lower boundaries.

At the end of the encoding process, the tag contains the fully encoded message. This can then be passed to the decoder.

When decoding, I decided to calculate all proportional boundary values beforehand, allowing the system to then quickly compare the buffer value and find the correct partition, thus decoding the correct character. This decoded character is then appended onto the rest of the decoded message, and then checked to see if it is equal to the EOD character. If so, decoding has been completed, and the decoded message can be returned. If not, the boundary values can be updated accordingly and rescaling can occur if necessary.

A constructor method exists for the arithmetic encoder that allows you to choose the precision of the buffer and boundary values. Since the random string that is generated is 80 characters long, I chose to set the encoder precision to be 16. This will allow you to enter a substantially large source alphabet without any boundaries overlapping.



```
===========
Coded message then decoded using Huffman decoder:
baabdaadabddddbddbbdabbddbbbdddaadddabddbbdaabbdabbaabadddaabba
aabbaddbdaadaaabaadaadddadbbaddabbaadc
===========
Message coded using arithmetic encoding with 16 bit precision:
0100111011000001000010100100010001000011010001001011001011111111
100101100010110011111011000110101110011101101101100010001101010
0001100101101010011111111010010001110100010110011110100111000000
00000
===========
Coded message then decoded using arithmetic decoder:
baabdaadabddddbddbbdabbddbbbdddaadddabddbbdaabbdabbaabadddaabba
aabbaddbdaadaaabaadaadddadbbaddabbaadc
===========
```

This is the same message as the binary and tertiary huffman encoders used, encoded and decoded using the arithmetic encoder and decoder.

Statistics

The average code length and compression ratio is calculated for each encoder, as well as the length of the encoded string is given.

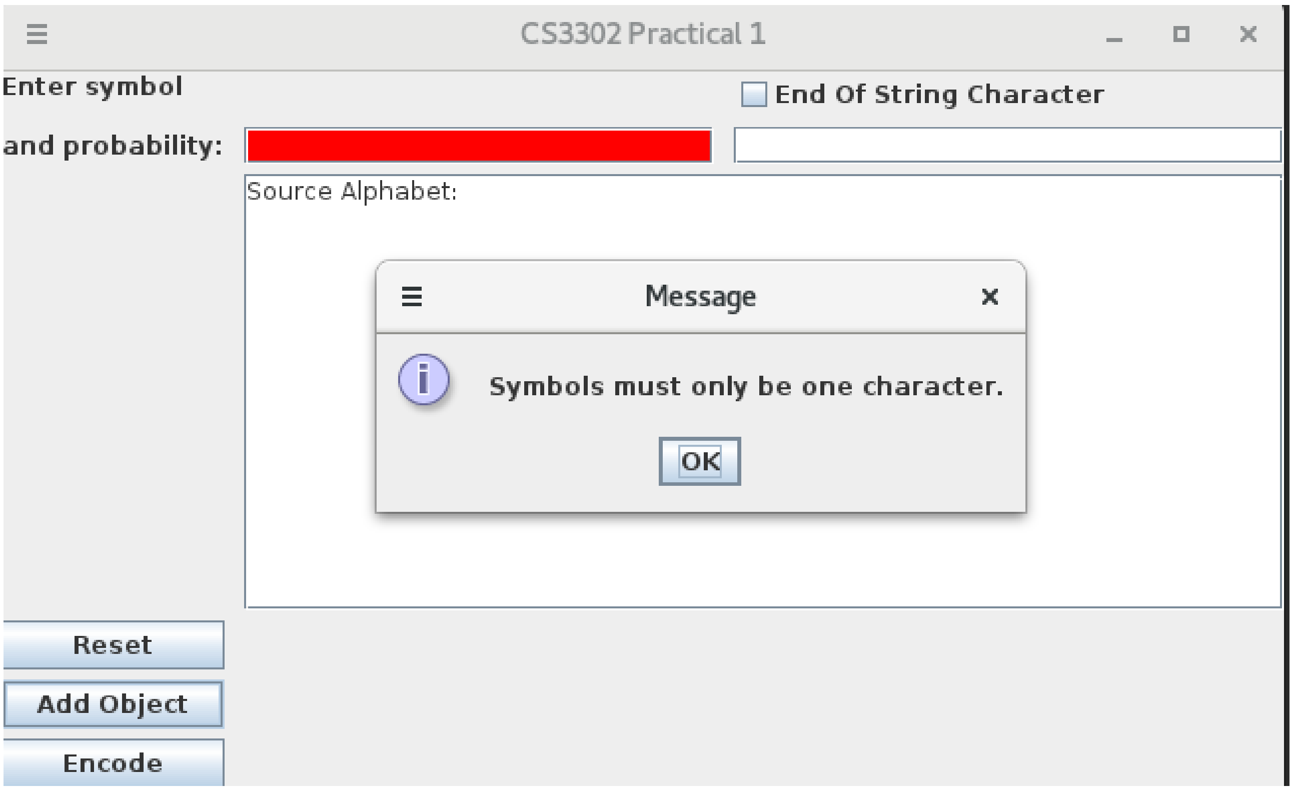
The average code length represents the average amount of bits used to encode a single symbol.

The compression ratio shows how well the encoding has compressed the original message. This assumes the original message was originally written using 7 bit ASCII characters (if you assume it used UTF-8, the compression could be substantially greater, depending on whether you're encoding crazy accents or symbols). The higher the compression ratio, the better the compression. It essentially tells you how many of the encoded strings you can fit in the original message (encoded in ASCII).

The entropy of the information source is also calculated and displayed. Since the random string isn't generated based on the source probabilities, there's actually an equal probability for each symbol to appear. This means the entropy of the random strings probability distribution will almost always be different to the user input information source.

Testing

Inputting blank symbol


CS3302 Practical 1

Enter symbol and probability:

End Of String Character

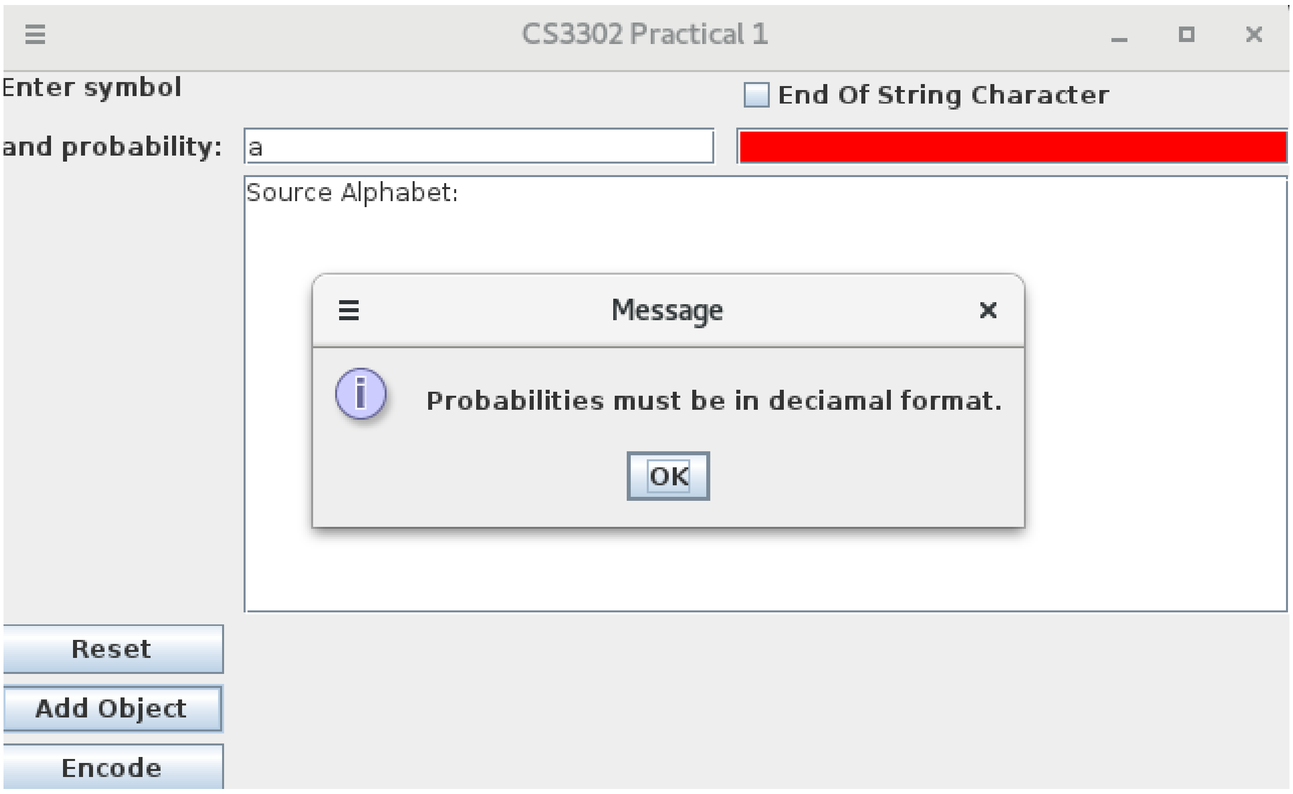
Source Alphabet:

Message

Symbols must only be one character.

OK

Inputting blank probability


CS3302 Practical 1

Enter symbol and probability: a

End Of String Character

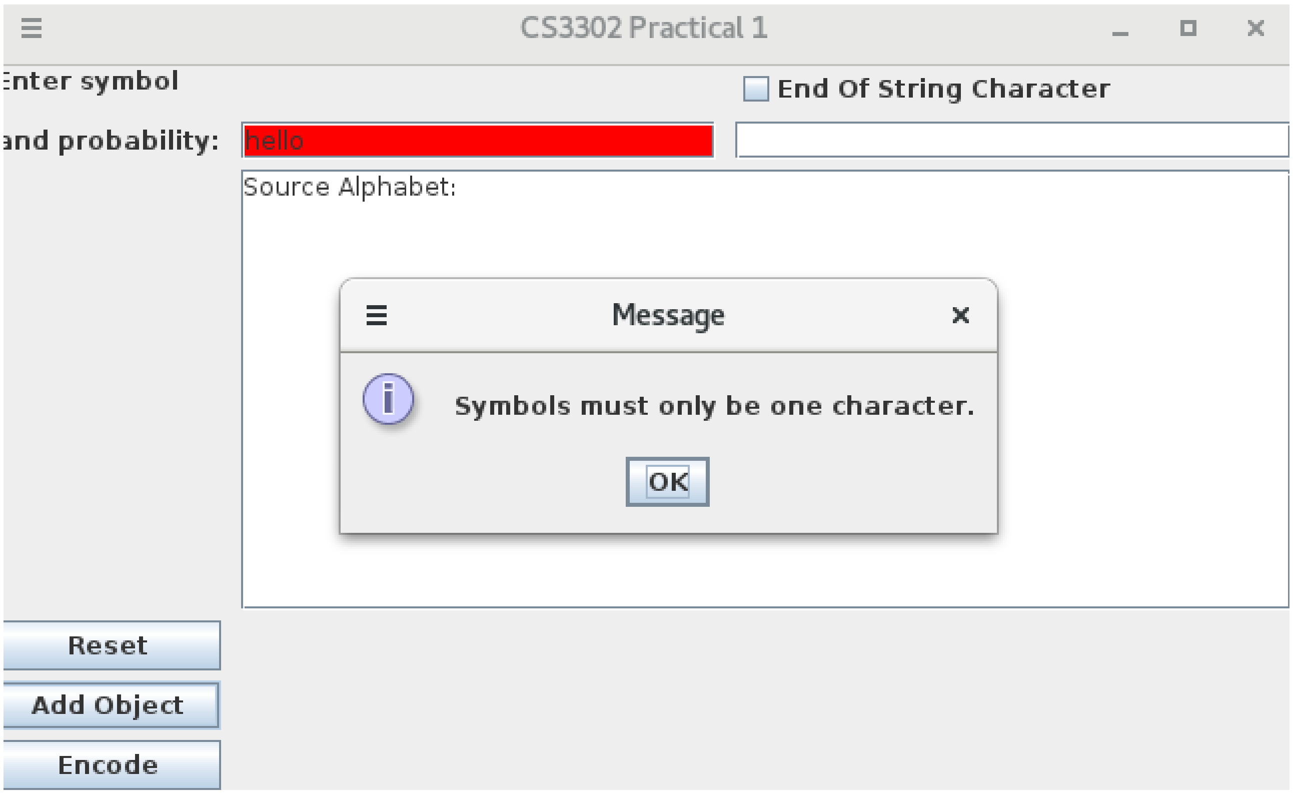
Source Alphabet:

Message

Probabilities must be in deciamal format.

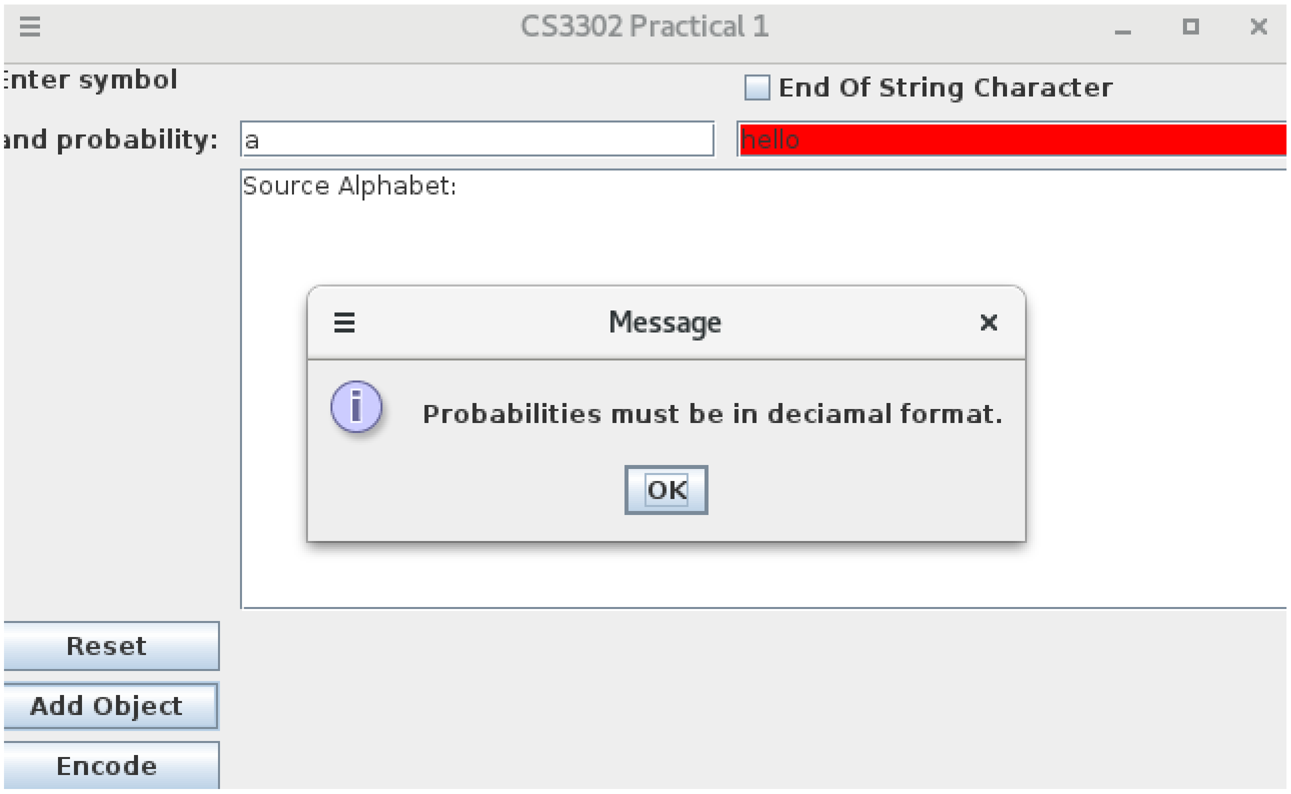
OK

Inputting string as symbol



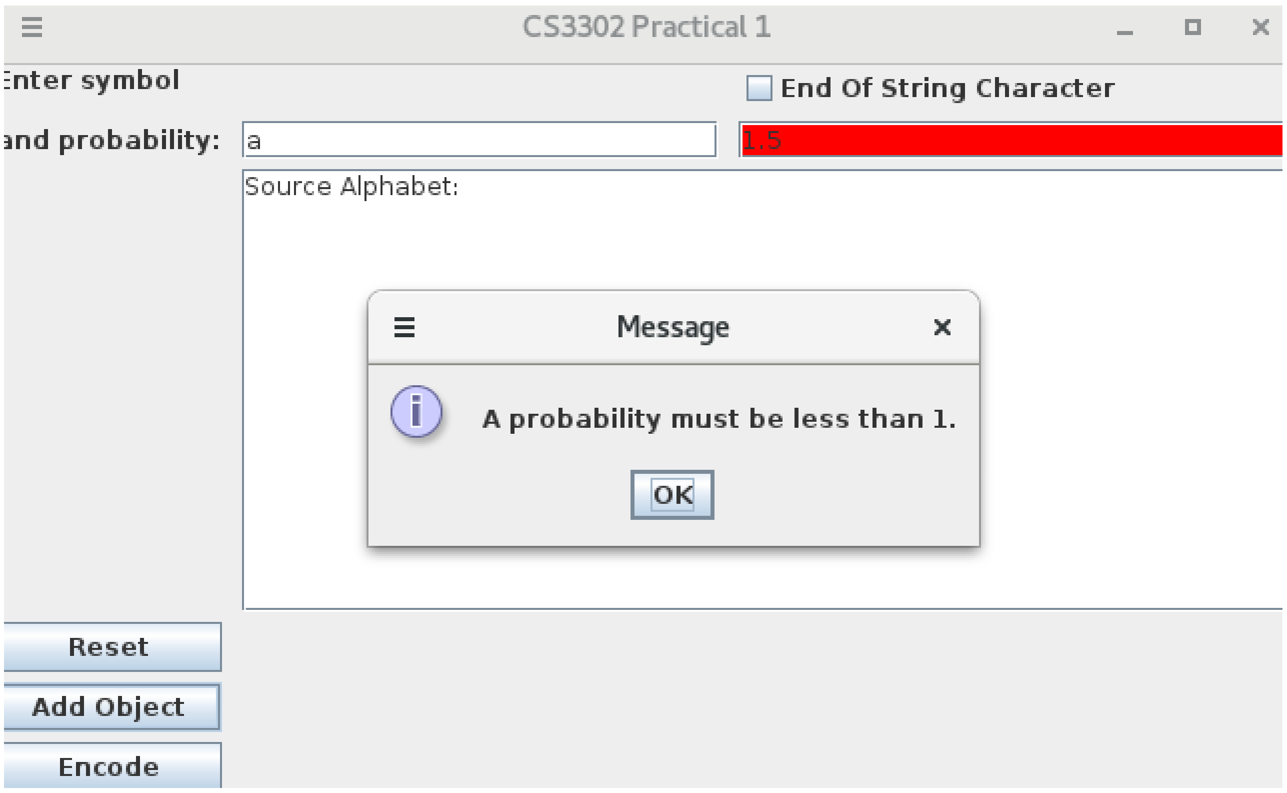
Symbols must only be one character.

Inputting string as probability



Probabilities must be in deciamal format.

Inputting probability greater than 1



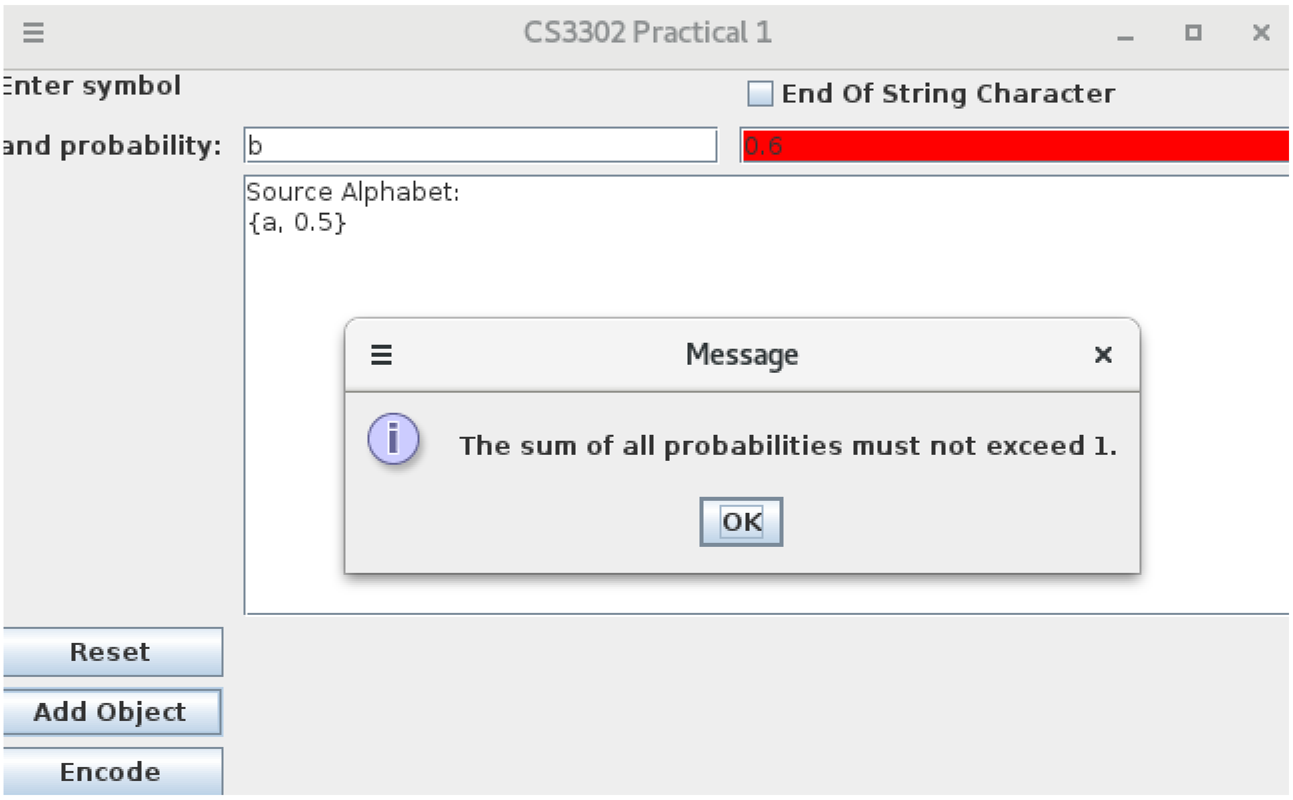
a

1.5

Source Alphabet:

A probability must be less than 1.

Trying to exceed probability sum of 1



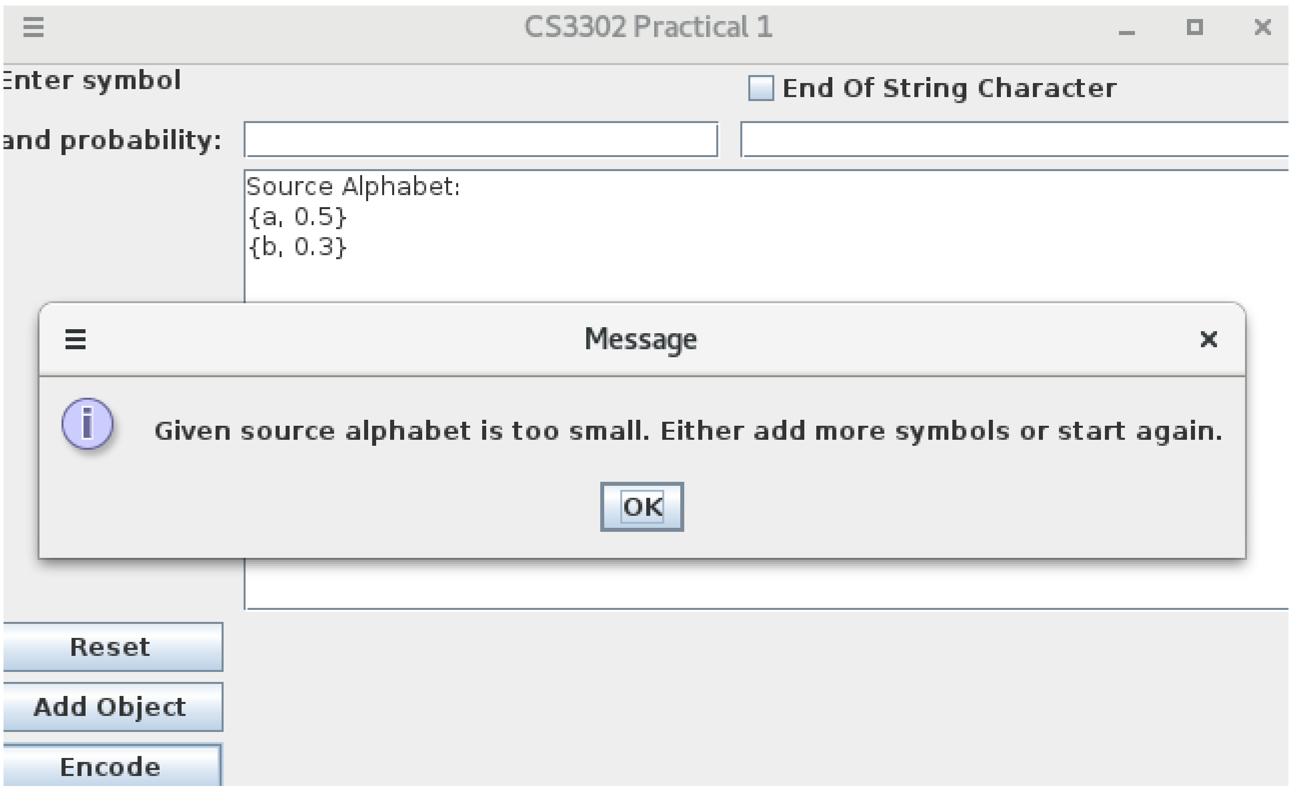
b

0.6

Source Alphabet:
{a, 0.5}

The sum of all probabilities must not exceed 1.

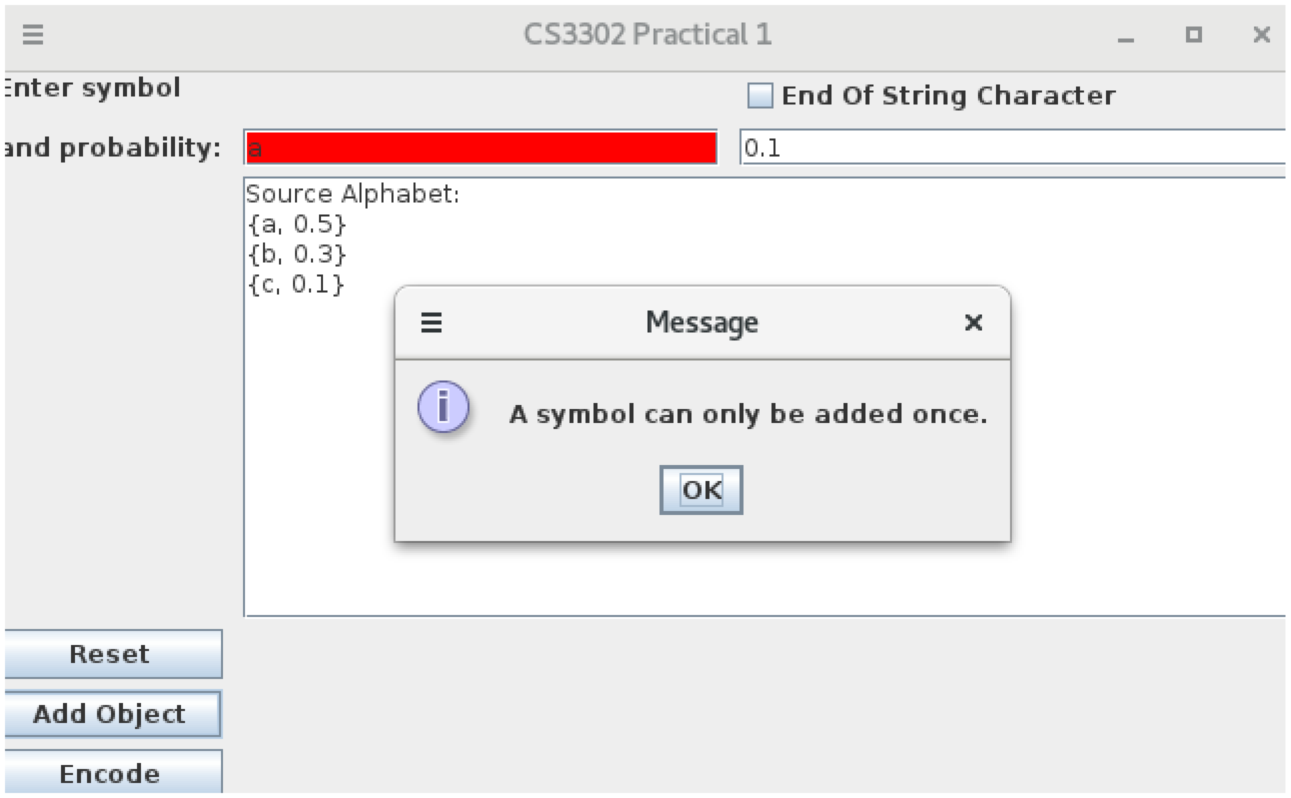
Trying to submit a source alphabet that's too small



```
Source Alphabet:
{a, 0.5}
{b, 0.3}
```

Given source alphabet is too small. Either add more symbols or start again.

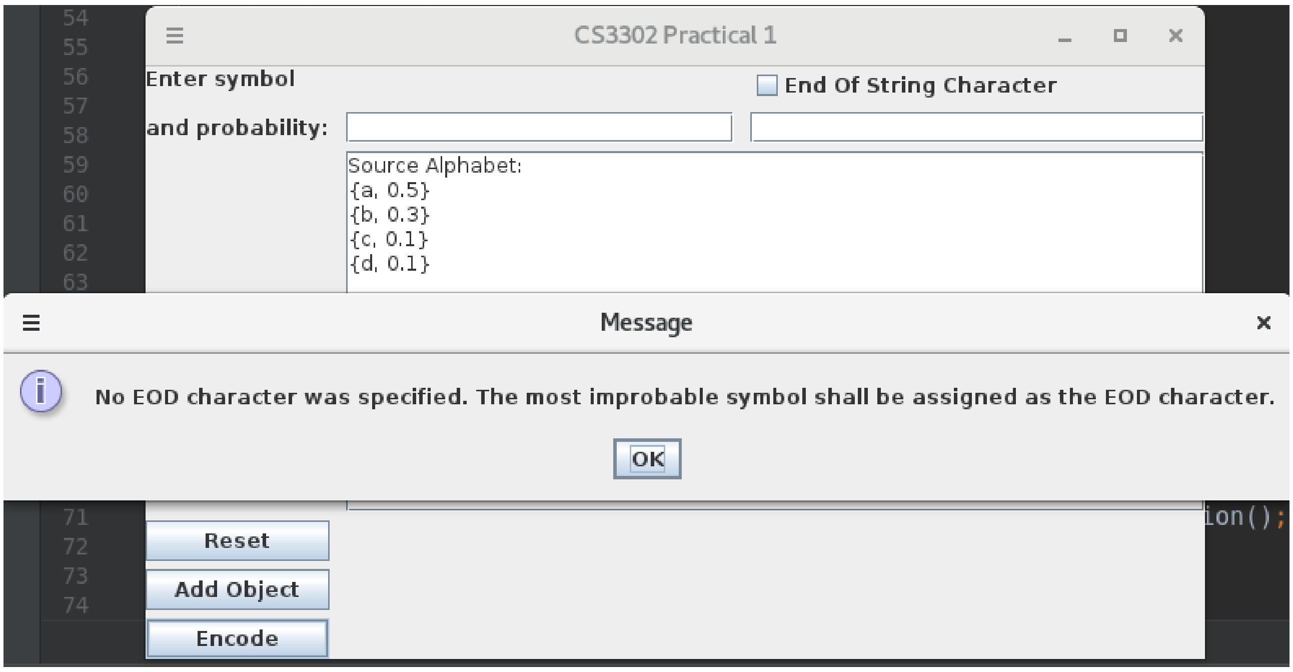
Inputting symbol that already exists



```
Source Alphabet:
{a, 0.5}
{b, 0.3}
{c, 0.1}
```

A symbol can only be added once.

Attempting to encode without specifying EOD characters


CS3302 Practical 1

Enter symbol and probability:

End Of String Character

Source Alphabet:
{a, 0.5}
{b, 0.3}
{c, 0.1}
{d, 0.1}

Message

No EOD character was specified. The most improbable symbol shall be assigned as the EOD character.

Evaluation

I feel I have met all the basic requirements of this practical to a high standard, as well as implementing an extension for the huffman encoder to code in a tertiary coding alphabet. Although the possibility isn't explored, the functionality exists for any code alphabet size to be used.

I f I were to improve my application, I would have the random string be dictated by the probabilities given within the information source, allowing your coded lengths to be directly compared to the strings entropy value.