Huffman Coding Report

Alan O'Regan

Project Description

This project is an implementation of the Huffman Coding string compression algorithm. This implementation uses the List and Tree abstract data types (with the addtion of the comparable interface) implemented in the **ListArrayBased** and **TreeNode** classes from the Data Structures and Algorithms module along with custom data classes **HuffmanSymbol** and **HuffmanEncodedSymbol** to handle the data. This implementation includes a graphical user interface to interact with the program, developed using methods from the GUI Programming Module.

Part 1: Generating the Huffman Tree

In order to generate the huffman tree, the generateHuffmanTree() method uses the frequency table that was generated from the given LetterCountAscending.txt file and sorted by frequency. - The method iterates through the frequency while there is at least 2 items remaining in the table - On each iteration the top 2 elements are added to a new node as the left and right child nodes with the character as '*' and the frequency value as the sum of the 2 child nodes frequency values. - The top 2 items are then removed from the frequency table and the new node is added - The table is then sorted using the sort method added to the ListArrayBased class based on the bubble sort algorithm. - When the iteration is complete the rootNode can be used to navigate the tree.

Implementation: HuffmanCoding.generateHuffmanTree Line:47

```
private void generateHuffmanTree() {

    TreeNode first;
    TreeNode second;
    HuffmanSymbol newRootSymbol;
    while (frequencyTable.size() > 1) {
        // get the first two items from the table and cast them to a HuffmanSymbol
        first = (TreeNode) frequencyTable.get(1);
        second = (TreeNode) frequencyTable.get(2);

        // create the new root symbol with a '*' character
        // and the sum of the first two items frequency
        newRootSymbol = new HuffmanSymbol(
                '*',
                ((HuffmanSymbol)first.getItem()).getFrequency()
                        + ((HuffmanSymbol)second.getItem()).getFrequency()
        );

        // replace the root node with the new root symbol node
        // and use the first two items as the left and right children
        rootNode = new TreeNode(newRootSymbol);
        rootNode.setLeft(first);
        rootNode.setRight(second);

        // remove the first two items as they are now in a tree
        frequencyTable.remove(1);
        frequencyTable.remove(1);

        frequencyTable.add(frequencyTable.size()+1, rootNode); // add new root symbol to t
        frequencyTable.sort(); // sort in order to place new root symbol in its proper pla
    }
}
```

Part 2: Encode

In order to encode a given String of characters using the lookup table generated from the rootNode using a recursive post order traversal.

Implementation: HuffmanCoding. Line:115

```
/**
 * A recursive post order traversal of the huffman tree
 * @param node the current node (first call should be the root node)
 * @param binary this parameter is used to pass the binary value throughout the tree (firs
 */
private void postOrderTraverse(TreeNode node, String binary) {
        if (node == null) {
            return;
        }

        char character = ((HuffmanSymbol)node.getItem()).getCharacter();
        if (character != '*') {
            lookupTable.add(lookupTable.size()+1, new HuffmanEncodedSymbol(character, binary))
            return;
        }

        postOrderTraverse(node.getLeft(), binary+"0");
        postOrderTraverse(node.getRight(), binary+"1");
    }
```

The encodeCharacters() method uses binary search to search for the binary value of each character in the string and appends it to the StringBuilder which will then be returned as the encoded value.

Implementation: HuffmanCoding.encodeCharacters Line:115

```
    /**
     * encoding method based on binary search.
     * @param characters the string of characters to be encoded
     * @return the encoded value of given characters
     */
    public String encodeCharacters(String characters) {
        StringBuilder sb = new StringBuilder();
        // validate input as only capital letters from A-Z
        characters = characters.toUpperCase(Locale.ROOT).replaceAll("[^A-Z]","");

        for (char ch : characters.toCharArray()) {

            int low = 1, high = lookupTable.size(), mid;
            while (low <= high) {
                mid = (low + high) / 2;
                HuffmanEncodedSymbol midItem = ((HuffmanEncodedSymbol)lookupTable.get(mid));

                if (midItem.letter == ch) {
                    sb.append(midItem.binary);
                    break; // value found exit loop
                }
                if (ch > midItem.letter)
                    low = mid + 1;
                else
                    high = mid - 1;
            }
        }

        return sb.toString();
    }
```

Part 3: Decode

The decode implementation is handled in the decodeCharacters() method using the rootNode of the huffman tree. Each character in the encoded string are iterated through using a enhanced for loop. - the method intialises the current node as the root node - if a character is 0 then the the current node is set as the let child of the current node and the right node for 1. - if the character of the node is the not '*' then the character is added to the stringbuilder and the current node is reset to the root node. - once the loop is finished the contents of the string builder are returned as the decoded value.

Implementation: *HuffmanCoding.decodeCharacters() Line:147*

```
/**
 * Huffman Decoding implementation
 * @param characters the string of characters to be decoded
 * @return the decoded value of given characters
 */
public String decodeCharacters(String characters) {
    StringBuilder sb = new StringBuilder();
    TreeNode currentNode = rootNode;
    // validate characters as only numbers 0 and 1
    characters = characters.toUpperCase(Locale.ROOT).replaceAll("[^01]","");

    for (char ch : characters.toCharArray()) {

        char character = ((HuffmanSymbol) currentNode.getItem()).getCharacter();

        if (character != '*') {
            sb.append(character);
            currentNode = rootNode;
        }
```

```
            currentNode = switch (ch) {
                case '0' -> currentNode.getLeft();
                case '1' -> currentNode.getRight();
                default -> currentNode;
            };
        }
        if (((HuffmanSymbol) currentNode.getItem()).getCharacter() != '*')
            sb.append(((HuffmanSymbol) currentNode.getItem()).getCharacter());

        return sb.toString();
    }
```

Part 4: Program Interface

The program interface features 2 buttons for encoding and decoding, a label for user messages and an input box for encoding and decoding, the results of the encoding and decoding are also placed inside the input box.

Encode button logic

Implementation: *HuffmanCodingGUI.actionPerformed() Line:188*

Compression Ratio Formula:

Number of bits in input = Number of characters after sanitizing * 7

(Number of bits in encoding / Number of bits in input) * 100

```
    if (source == encode) {

        if (!inputText.isBlank())
            result = huffmanCoding.encodeCharacters(inputText);
        else
            resultTextArea.setText("Empty Input");


        if (!result.isBlank()) {
            inputTextArea.setText(result);
            resultTextArea.setText("Encoded!");
```

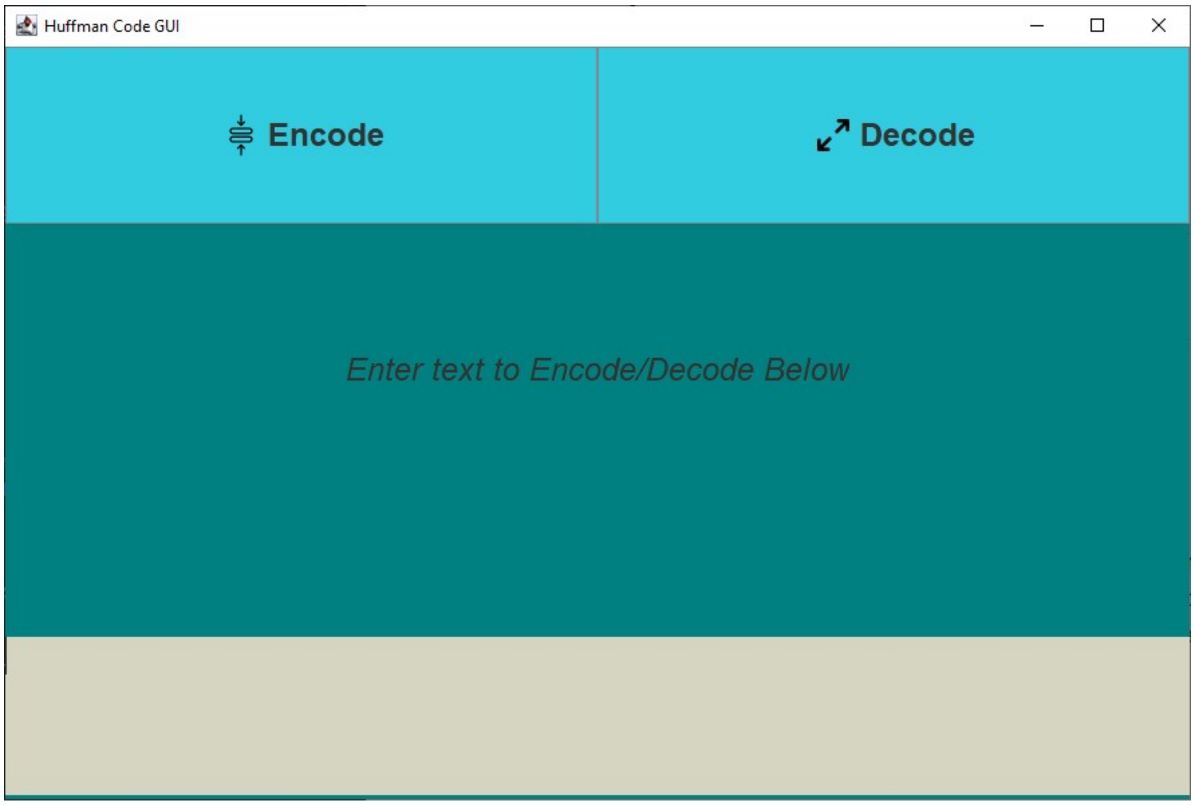


Figure 1: Screenshot of Program Interface

```
            inputText = inputText.toUpperCase(Locale.ROOT).replaceAll("[^A-Z]","");

            if (inputText.length() > 0) {
                resultTextArea.setText(String.format("Encoded! (Compression Ratio: %.2f%%)", (
            }
        }
    }
```

Decode button logic

Implementation: *HuffmanCodingGUI.actionPerformed() Line:108*

```
if (source == decode) {

    if (!inputText.isBlank())
        result = huffmanCoding.decodeCharacters(inputText);
    else
        resultTextArea.setText("Empty Input");


    if (!result.isBlank()) {
        inputTextArea.setText(result);
        resultTextArea.setText("Decoded!");
    }
}
```

Testing

To test the program, A unit test class was created in order to run tests on the critical methods implemented:

- Encoding
- Decoding
- Sorting

Unit Testing

The Unit tests used java assertions which are enabled by giving the `-ea` VM argument when running the test class.

Encoding Test

Encoding Unit Test - iterates through the lookup table and compares the binary value of each letter to the result of encodeCharacters() given the letter.

```java
public static void testEncodeCharacters() {
    for (int i = 1; i < huffmanCoding.getLookupTable().size(); i++) {
        HuffmanEncodedSymbol symbol = (HuffmanEncodedSymbol) huffmanCoding.getLookupTable(
        assert symbol.binary.equals(huffmanCoding.encodeCharacters(Character.toString(symb
        System.out.printf("%c == %-10s\n", symbol.letter, symbol.binary);
    }
}
```

Decoding Test

Decoding Unit Test - iterates through the lookup table and compares the letter of each item to the result of decodeCharacters() given the binary value for that letter.

```java
public static void testDecodeCharacters() {
    for (int i = 1; i < huffmanCoding.getLookupTable().size(); i++) {
        HuffmanEncodedSymbol symbol = (HuffmanEncodedSymbol) huffmanCoding.getLookupTable(
        assert symbol.letter == huffmanCoding.decodeCharacters(symbol.binary).charAt(0) :
        System.out.printf("%-10s == %c\n", symbol.binary, symbol.letter);
    }
}
```

Soring Test

Sorting Unit Test - The sorting unit test creates an unsorted an sorted array of the same items then adds them to a list which is then sorted and compared to the sorted array.

```java
public static void testSortMethod() {
    ListArrayBased unsortedList = new ListArrayBased();
    int[] unsortedArray = {1,4,2,-1};
    int[] sortedArray = {-1,1,2,4};

    for (int i = 0; i < unsortedArray.length; i++) {
        unsortedList.add(i+1, unsortedArray[i]);
    }
```

```
        unsortedList.sort();

        for (int i = 0; i < sortedArray.length; i++) {
            assert (int)unsortedList.get(i+1) == sortedArray[i] : String.format("Array not sor
        }
    }
```

Input Validation Testing

The GUI input and mehod parameters are sanitized using regex.

Validation Regex

```
// GUI and Encoding method sanitisation
inputText = inputText.toUpperCase(Locale.ROOT).replaceAll("[^A-Z]", "");

// Decdoding method sanitisation
characters = characters.toUpperCase(Locale.ROOT).replaceAll("[^01]","");
```

Test Cases

Table 1: Input Validation table

Input	Ouptut	Valid
Alan	111010111111101010	YES
ALaN	111010111111101010	YES
a;L2an	111010111111101010	YES
111010111111101010	ALAN	YES
abc11101011;11113401010	ALAN	YES
111010111111101010 2345	ALAN	YES

Test Case Screenshots

:::: {.columns}

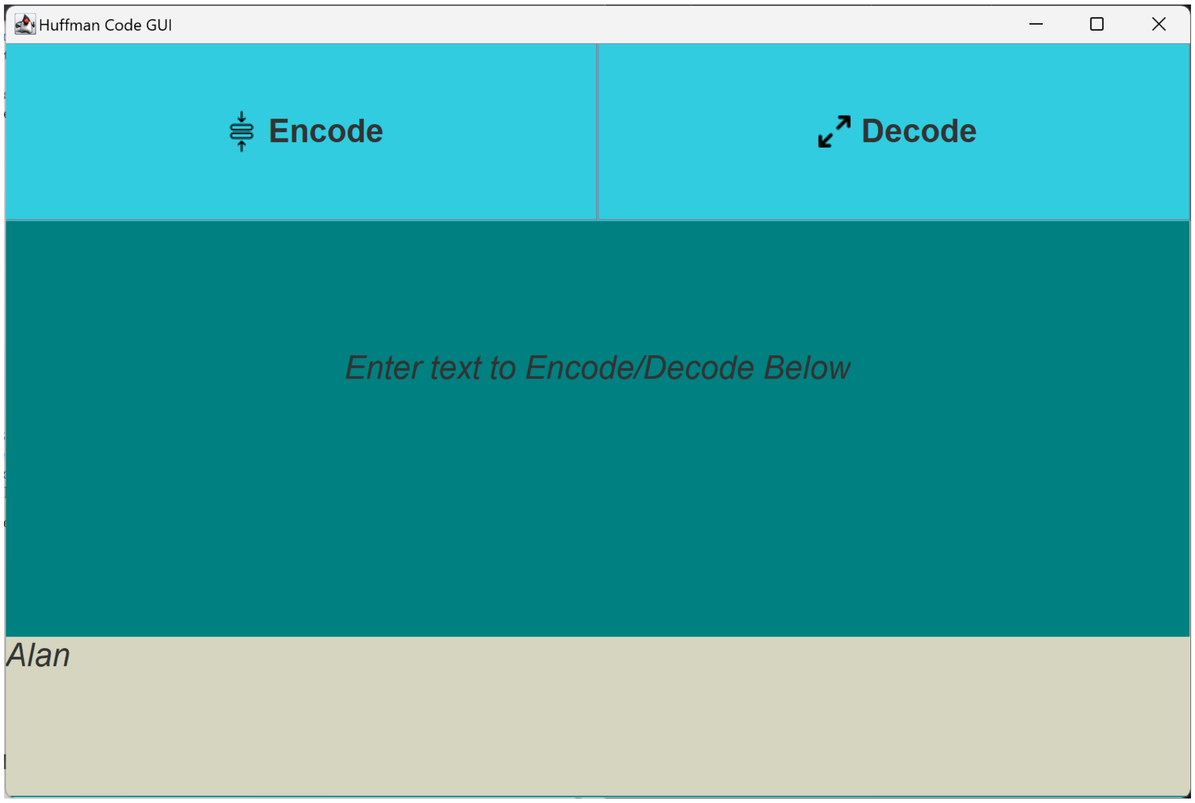
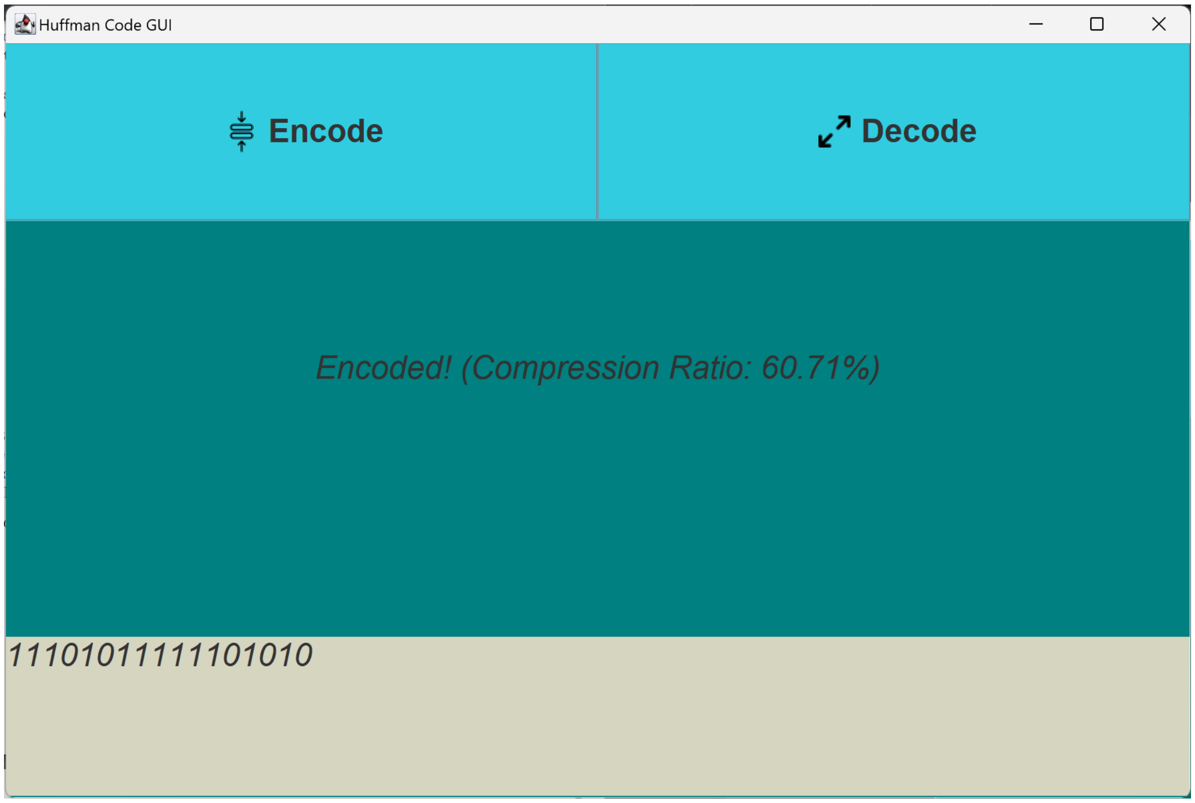


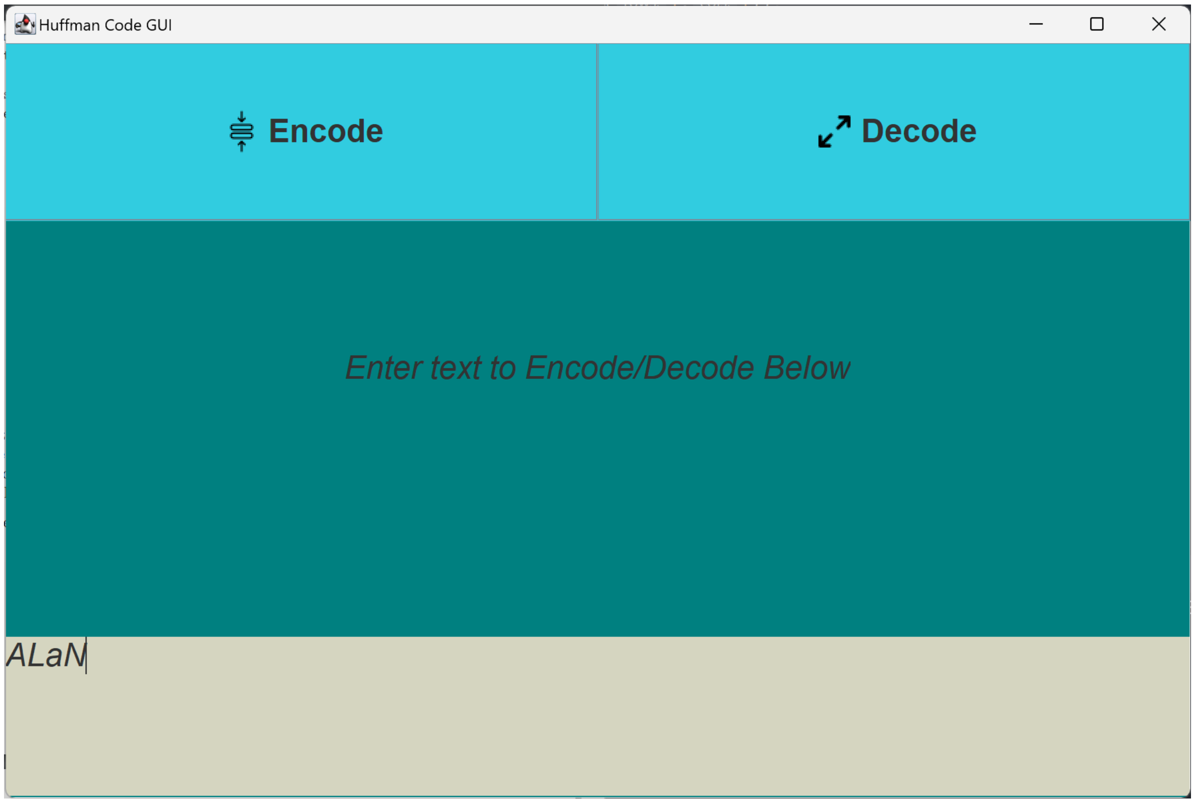
Figure 2: Encode Input test 1



Encoded! (Compression Ratio: 60.71%)

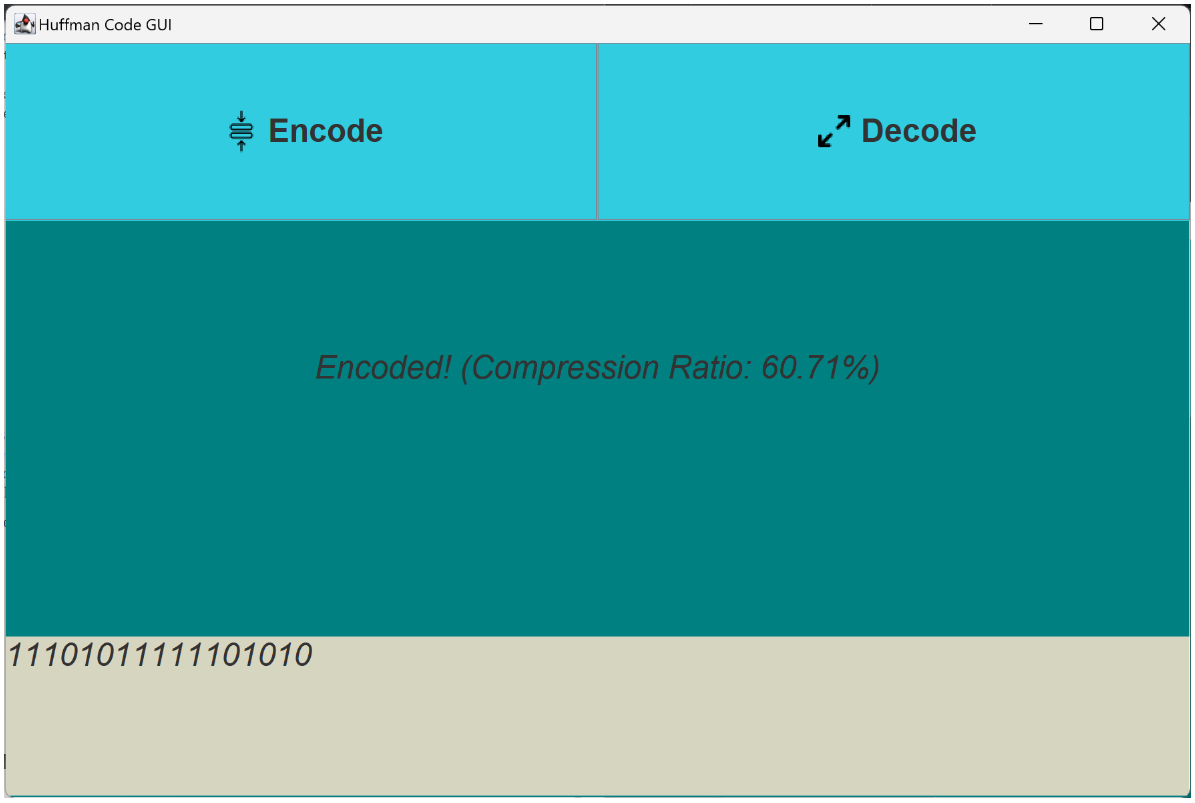
```
11101011111101010
```

Figure 3: Encode Output test 1



ALaN

Figure 4: Encode Input test 2



Encoded! (Compression Ratio: 60.71%)

```
11101011111101010
```

Figure 5: Encode Output test 2

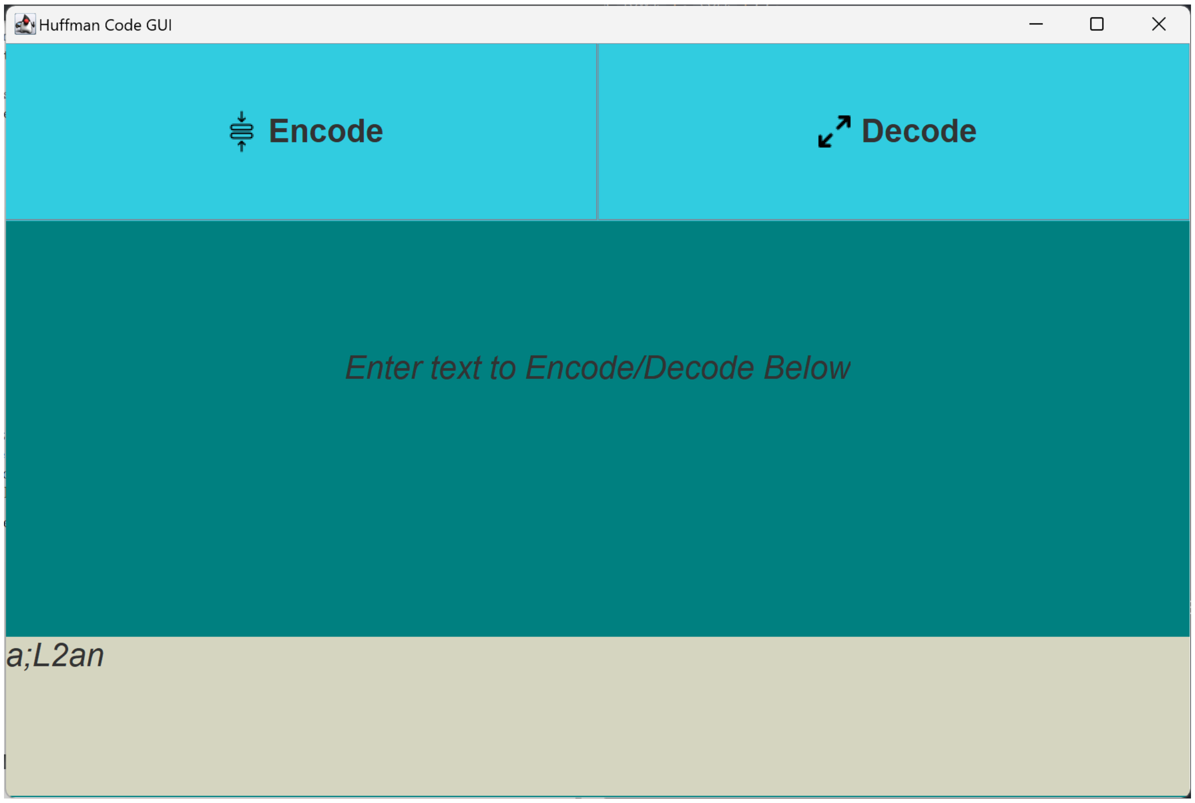
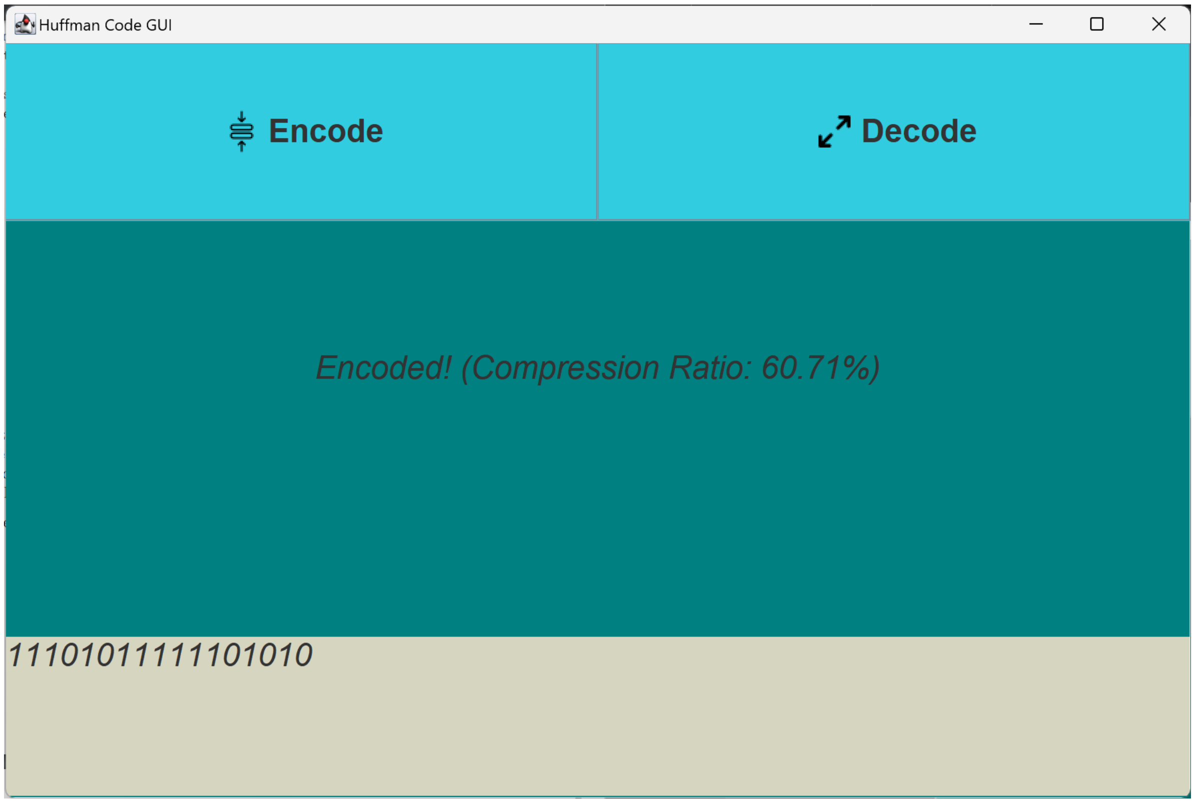


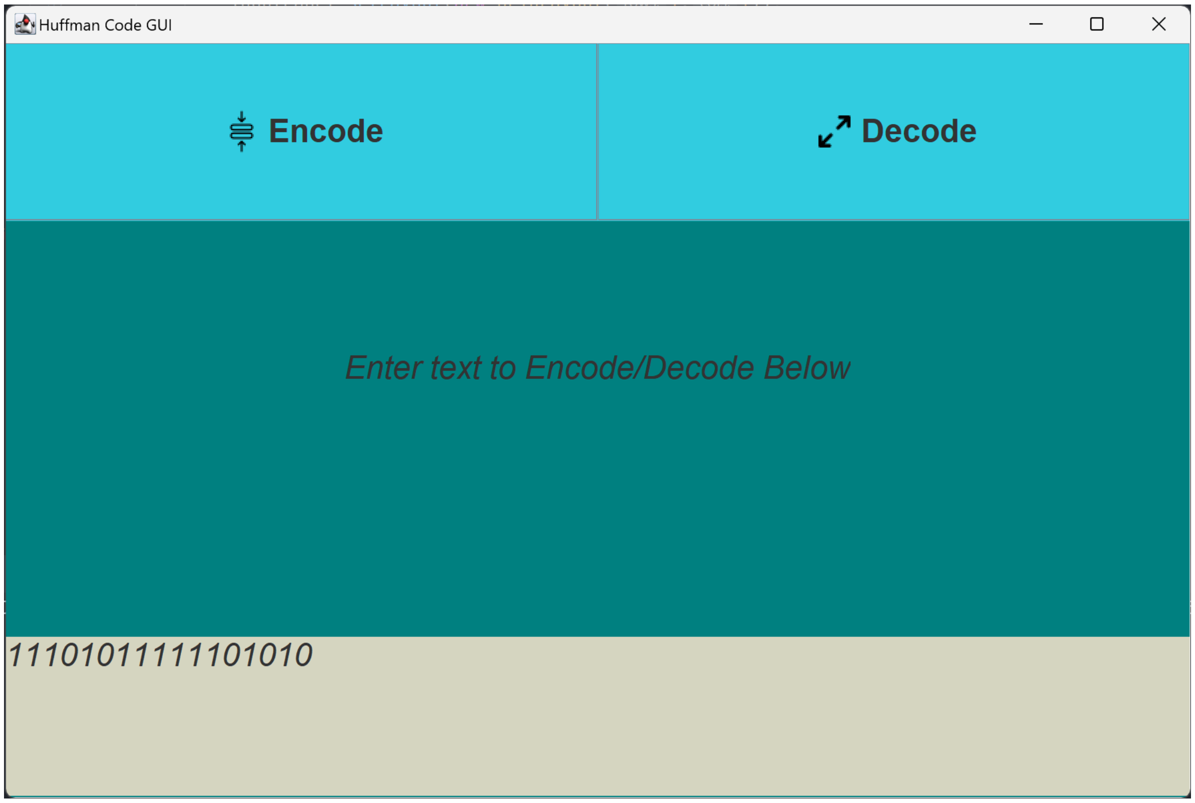
Figure 6: Encode Input test 3



Encoded! (Compression Ratio: 60.71%)

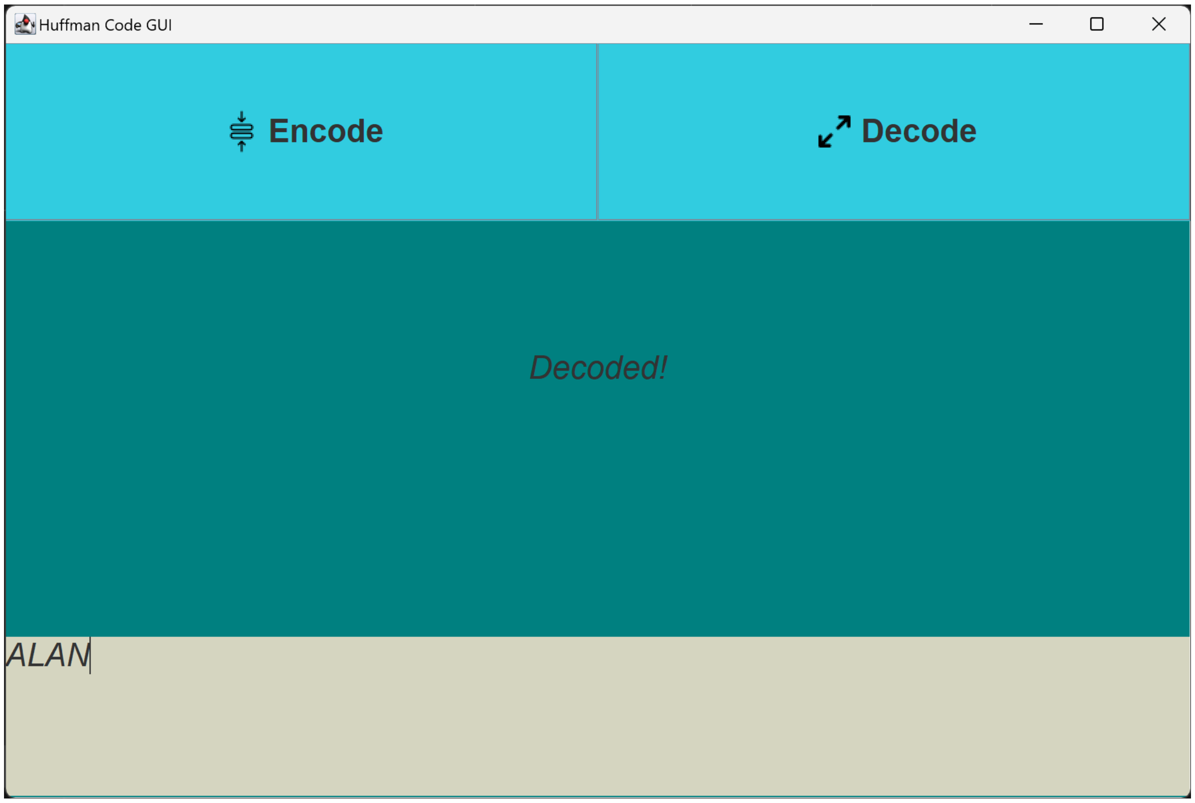
```
11101011111101010
```

Figure 7: Encode Output test 3



11101011111101010

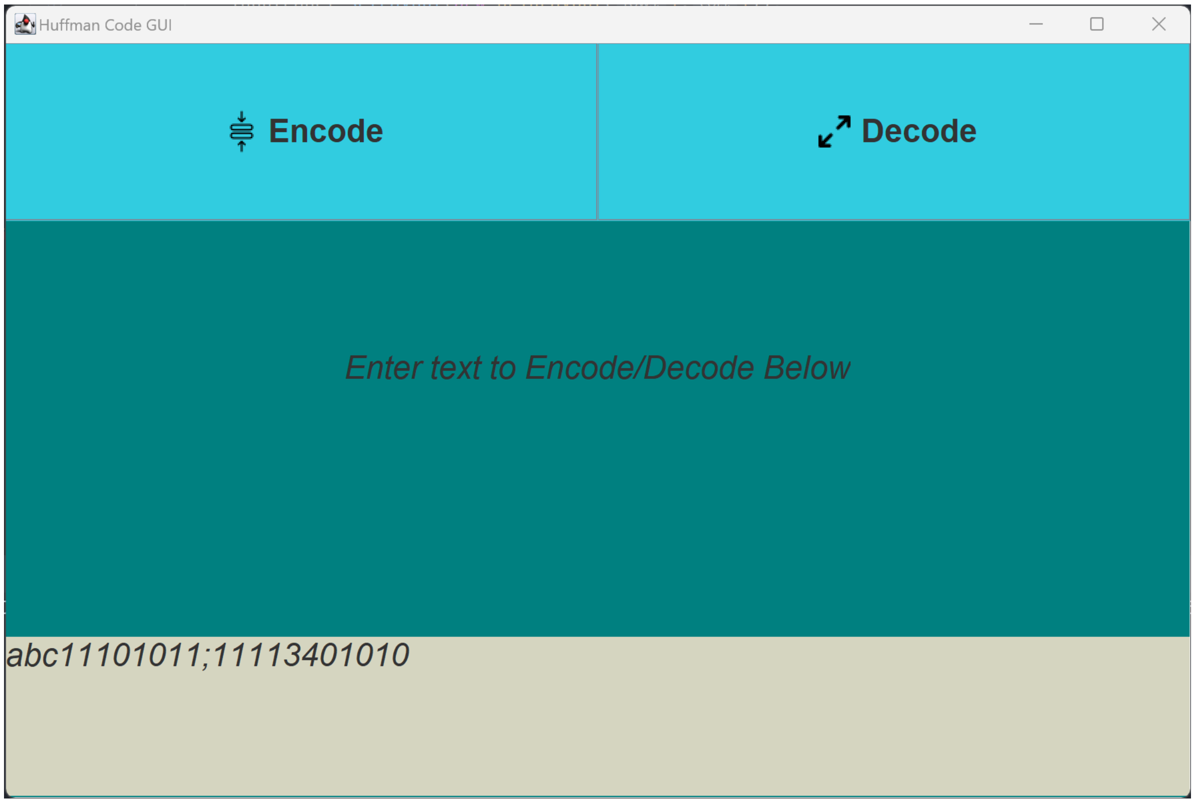
Figure 8: Decode Input test 1



Decoded!

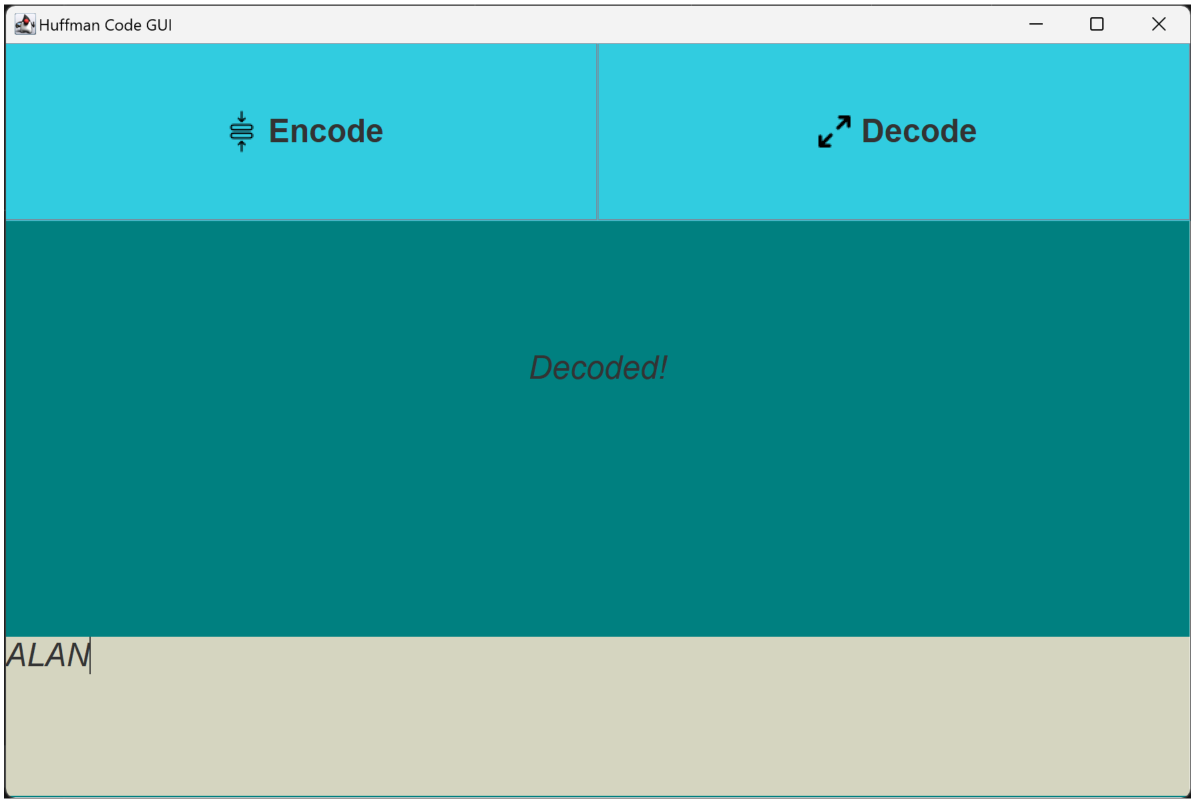
ALAN

Figure 9: Decode Output test 1



abc11101011;11113401010

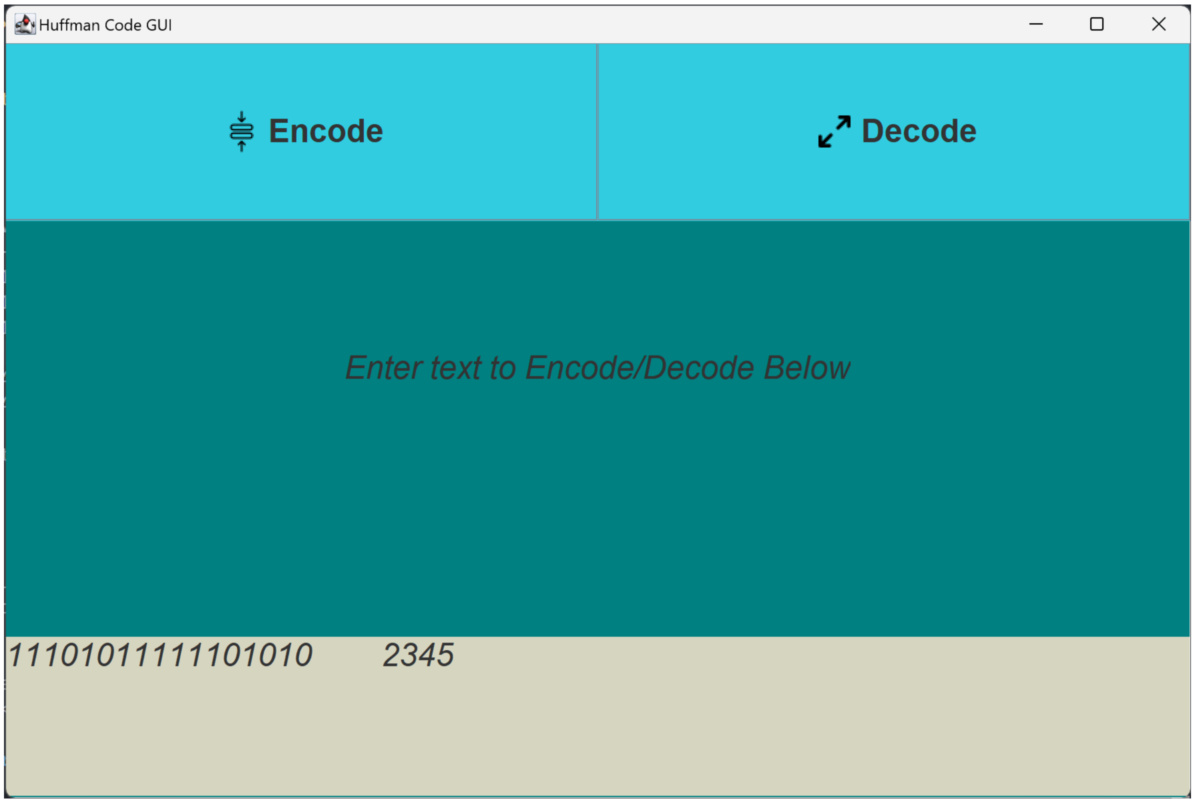
Figure 10: Decode Input test 2



Decoded!

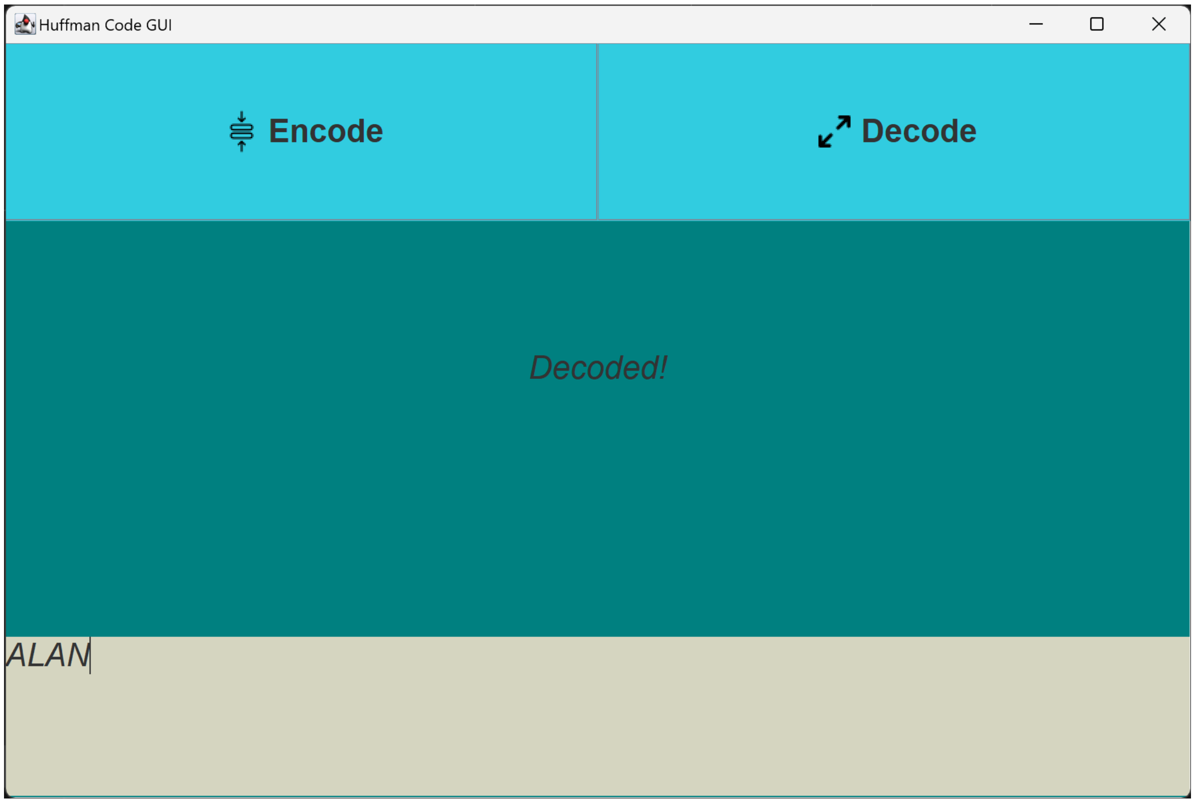
ALAN

Figure 11: Decode Output test 2



```
11101011111101010       2345
```

Figure 12: Decode Input test 3



Decoded!

ALAN

Figure 13: Decode Output test 3

Academic Honesty

Any work that you submit for continuous assessment or assignments must be done by you. Failure to acknowledge the source of a significant idea or approach is considered plagiarism and not allowed. Academic dishonesty will be dealt with severely. At a minimum, you will receive a mark of zero for the assignment.

Signed: Alan O'Regan Date: 30/11/2022

Sources and References

- GUI Button Icons
  - Compress Icon
  - Maximise Icon
- Code References
  - Tree traversal
  - Binary Search