Huffman Coding Implementation in Java

Introduction:

Huffman coding is a lossless data compression algorithm that assigns variable-length codes to input characters based on their frequencies. The most frequent characters get shorter codes, reducing the overall size of the encoded data.

This document describes a Java implementation of Huffman coding that includes functionalities for encoding and decoding text, calculating entropy, and managing code tables.

Class Structure:

1. `HuffmanNode`

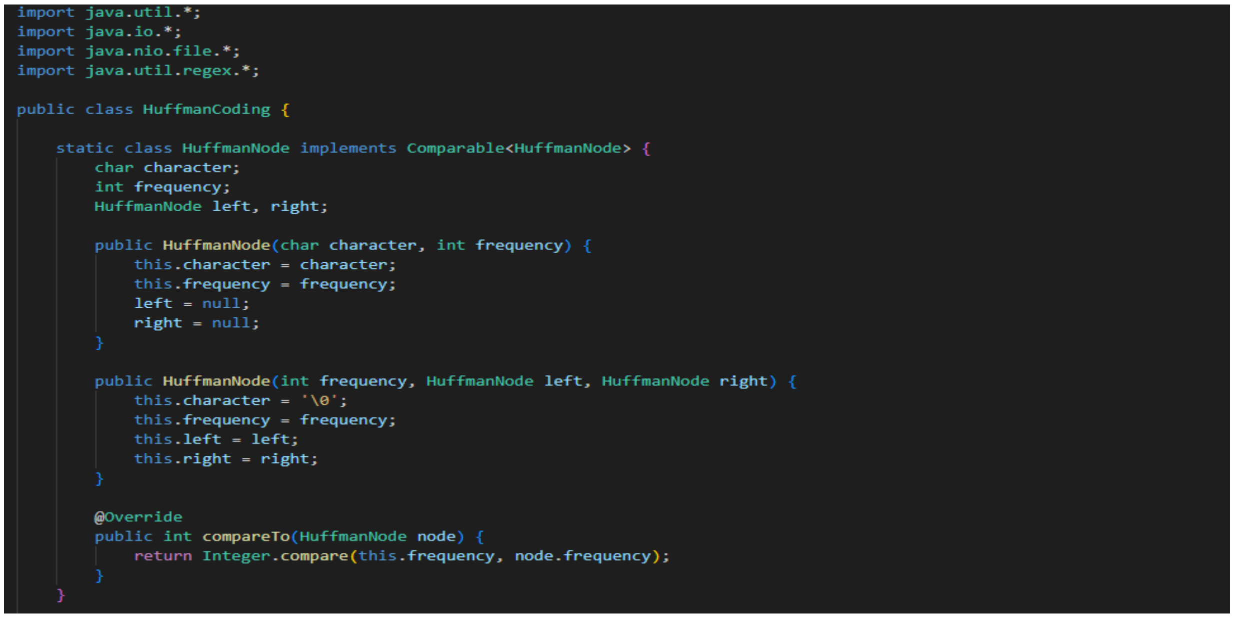
A node class that represents characters and their frequencies in the Huffman tree.

- `character`: The character being represented.
- `frequency`: The frequency of the character.
- `left`, `right`: Pointers to child nodes.

Implements `Comparable<HuffmanNode>` to allow sorting by frequency.

2. `HuffmanCoding`

The main class contains methods for encoding, decoding, and handling Huffman trees.



```java
import java.util.*;
import java.io.*;
import java.nio.file.*;
import java.util.regex.*;

public class HuffmanCoding {

    static class HuffmanNode implements Comparable<HuffmanNode> {
        char character;
        int frequency;
        HuffmanNode left, right;

        public HuffmanNode(char character, int frequency) {
            this.character = character;
            this.frequency = frequency;
            left = null;
            right = null;
        }

        public HuffmanNode(int frequency, HuffmanNode left, HuffmanNode right) {
            this.character = '\0';
            this.frequency = frequency;
            this.left = left;
            this.right = right;
        }

        @Override
        public int compareTo(HuffmanNode node) {
            return Integer.compare(this.frequency, node.frequency);
        }
    }
```

Core Methods

1. `calculateEntropy(Map<Character, Integer> frequencyMap, int totalCharacters)`

Computes the entropy of the given text based on character frequencies.

2. `buildHuffmanTree(Map<Character, Integer> frequencyMap)`

Builds the Huffman tree using a priority queue.

3. `generateHuffmanCodes(HuffmanNode root, String code, Map<Character, String> huffmanCodes)`

Recursively generates Huffman codes for characters in the tree.

4. `encodeText(String text, Map<Character, String> huffmanCodes)`

Encodes the input text using the generated Huffman codes.

5. `decodeHuffman(HuffmanNode root, String encodedText)`

Decodes the Huffman-encoded binary string back into text.

6. `rebuildHuffmanTree(Map<String, Character> codeMap)`

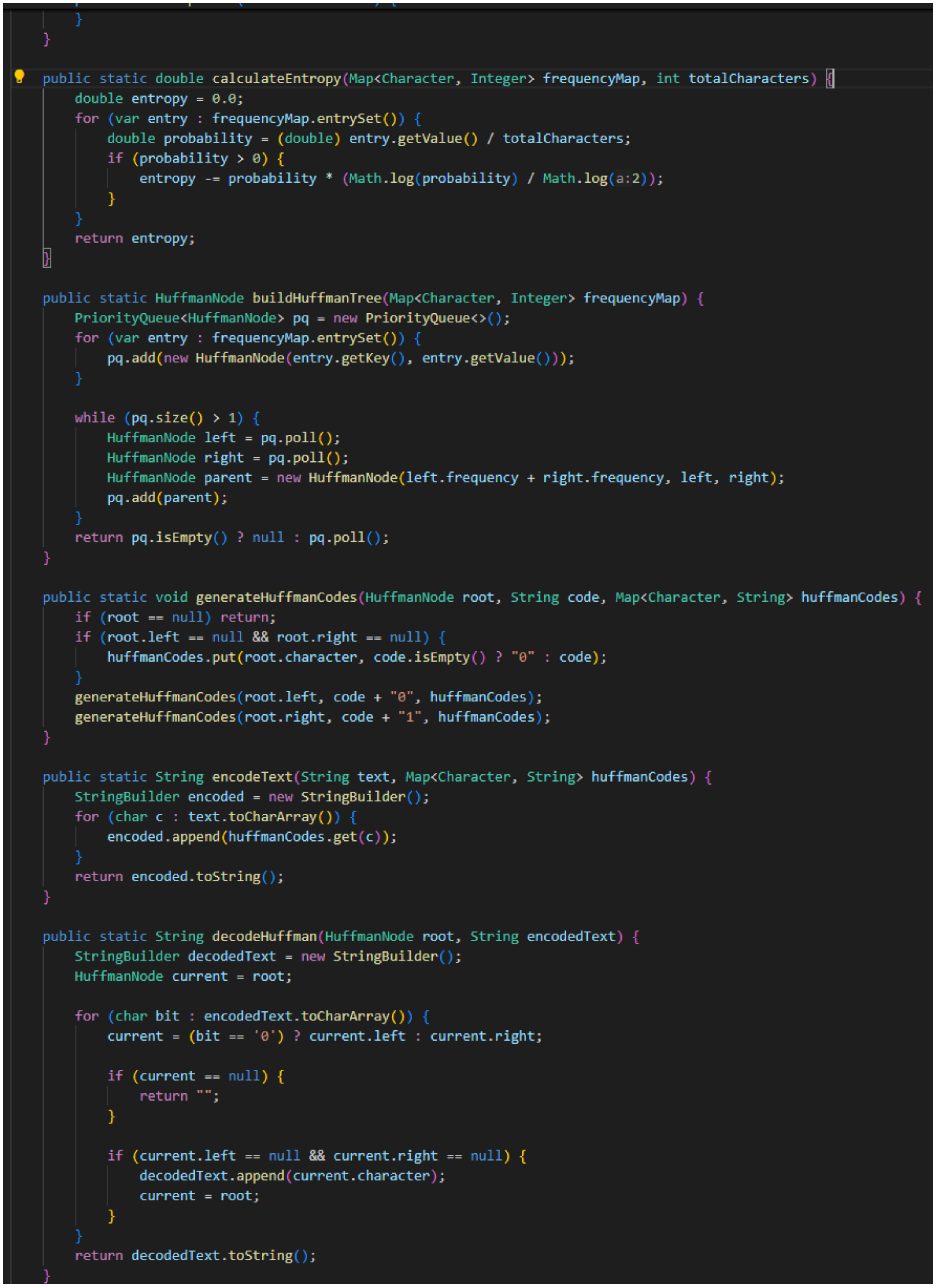
Reconstructs the Huffman tree from the code table.

7. `writeCodeTable(String filename, Map<Character, String> huffmanCodes, int bitLength, int padding)`

Writes the generated Huffman codes to a file.

8. `readCodeTable(String filename)`

Reads a saved code table and reconstructs the mapping of codes to characters.



```java
        }
    }

    public static double calculateEntropy(Map<Character, Integer> frequencyMap, int totalCharacters) {
        double entropy = 0.0;
        for (var entry : frequencyMap.entrySet()) {
            double probability = (double) entry.getValue() / totalCharacters;
            if (probability > 0) {
                entropy -= probability * (Math.log(probability) / Math.log(a:2));
            }
        }
        return entropy;
    }

    public static HuffmanNode buildHuffmanTree(Map<Character, Integer> frequencyMap) {
        PriorityQueue<HuffmanNode> pq = new PriorityQueue<>();
        for (var entry : frequencyMap.entrySet()) {
            pq.add(new HuffmanNode(entry.getKey(), entry.getValue()));
        }

        while (pq.size() > 1) {
            HuffmanNode left = pq.poll();
            HuffmanNode right = pq.poll();
            HuffmanNode parent = new HuffmanNode(left.frequency + right.frequency, left, right);
            pq.add(parent);
        }
        return pq.isEmpty() ? null : pq.poll();
    }

    public static void generateHuffmanCodes(HuffmanNode root, String code, Map<Character, String> huffmanCodes) {
        if (root == null) return;
        if (root.left == null && root.right == null) {
            huffmanCodes.put(root.character, code.isEmpty() ? "0" : code);
        }
        generateHuffmanCodes(root.left, code + "0", huffmanCodes);
        generateHuffmanCodes(root.right, code + "1", huffmanCodes);
    }

    public static String encodeText(String text, Map<Character, String> huffmanCodes) {
        StringBuilder encoded = new StringBuilder();
        for (char c : text.toCharArray()) {
            encoded.append(huffmanCodes.get(c));
        }
        return encoded.toString();
    }

    public static String decodeHuffman(HuffmanNode root, String encodedText) {
        StringBuilder decodedText = new StringBuilder();
        HuffmanNode current = root;

        for (char bit : encodedText.toCharArray()) {
            current = (bit == '0') ? current.left : current.right;

            if (current == null) {
                return "";
            }

            if (current.left == null && current.right == null) {
                decodedText.append(current.character);
                current = root;
            }
        }
        return decodedText.toString();
    }
```

Encoding Process

1. **Read Input File**: Read the text content from `input.txt`.
2. **Compute Frequency**: Count occurrences of each character.
3. **Calculate Entropy**: Compute entropy based on character probabilities.
4. **Build Huffman Tree**: Construct the Huffman tree using a priority queue.
5. **Generate Huffman Codes**: Traverse the tree to assign binary codes to characters.
6. **Encode Text**: Replace characters in the text with their respective Huffman codes.
7. **Handle Binary Output**: Convert encoded binary string into bytes and write to `encoded.bin`.
8. **Save Code Table**: Write the Huffman codes to `code_table.txt`.
9. **Store Calculations**: Save entropy and frequency details to `calculations.txt`.

Decoding Process

1. **Read Code Table**: Load mappings from `code_table.txt`.
2. **Rebuild Huffman Tree**: Construct the tree based on stored mappings.
3. **Read Encoded Data**: Load the binary-encoded content from `encoded.bin`.
4. **Extract Binary String**: Convert bytes back into a binary string.
5. **Trim Padding**: Remove extra padding bits.
6. **Decode Text**: Traverse the Huffman tree to reconstruct the original text.
7. **Save Output**: Write the decoded text to `output.txt`.

File Management

- `input.txt`: Contains the original text.
- `encoded.bin`: Stores the compressed binary output.
- `code_table.txt`: Stores the Huffman code mappings.
- `output.txt`: Stores the decoded text.
- `calculations.txt`: Stores entropy calculations and frequency data.

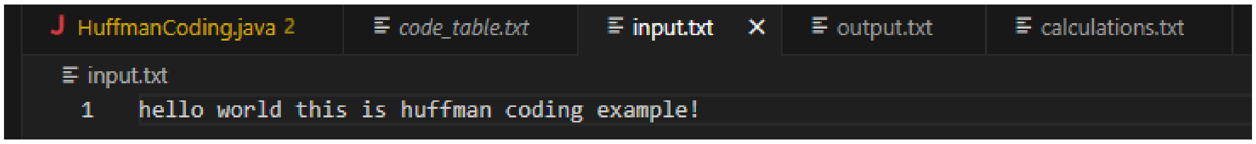
User Interface

A simple console-based interface allows users to:

1. Encode text from a file.
2. Decode the binary-encoded text.
3. Exit the program.

Steps & test cases:

Step1:

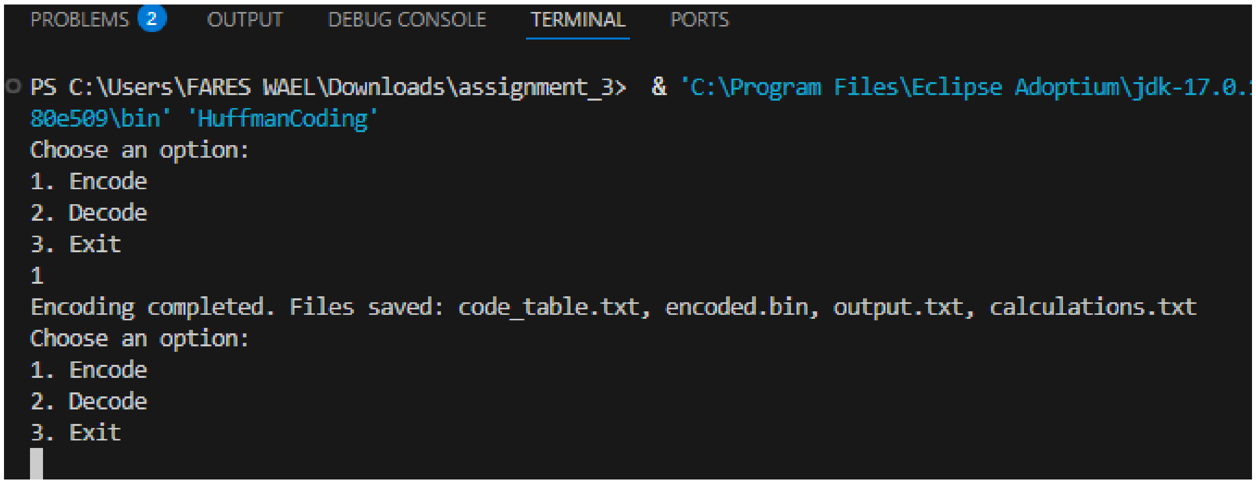


HuffmanCoding.java 2 | code_table.txt | input.txt | output.txt | calculations.txt

input.txt

```
hello world this is huffman coding example!
```

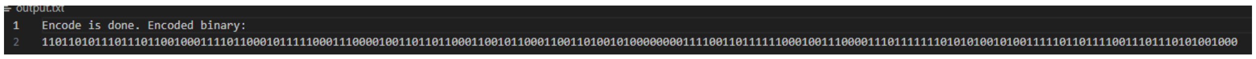
Step 2:



PROBLEMS 2 | OUTPUT | DEBUG CONSOLE | TERMINAL | PORTS

```
PS C:\Users\FARES WAEL\Downloads\assignment_3>  & 'C:\Program Files\Eclipse Adoptium\jdk-17.0.
80e509\bin' 'HuffmanCoding'
Choose an option:
1. Encode
2. Decode
3. Exit
1
Encoding completed. Files saved: code_table.txt, encoded.bin, output.txt, calculations.txt
Choose an option:
1. Encode
2. Decode
3. Exit
```

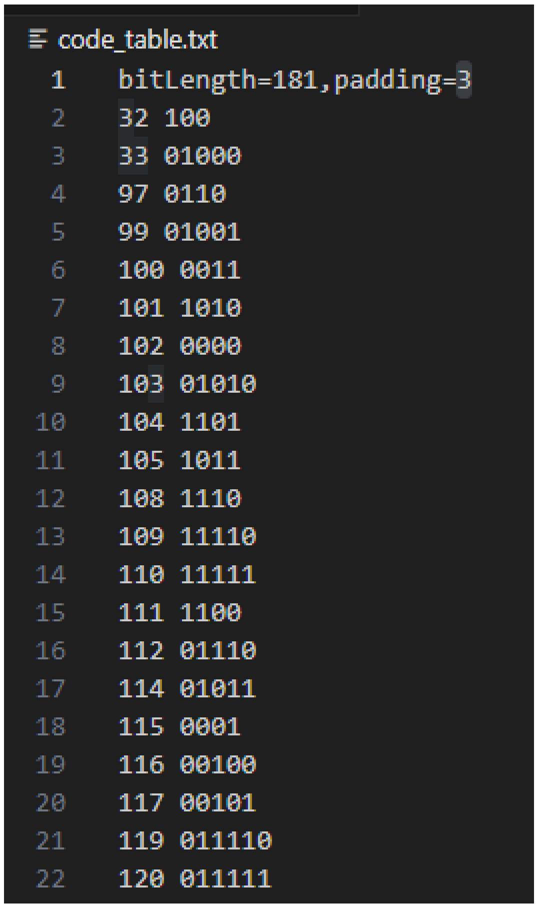
Step 3:



output.txt

```
Encode is done. Encoded binary:
110110101110111011001000111101100010111110001110000100110110110001100101100011001101001010000000001111001101111110001001110000111011111110101010010100111110110111100111011101010 01000
```

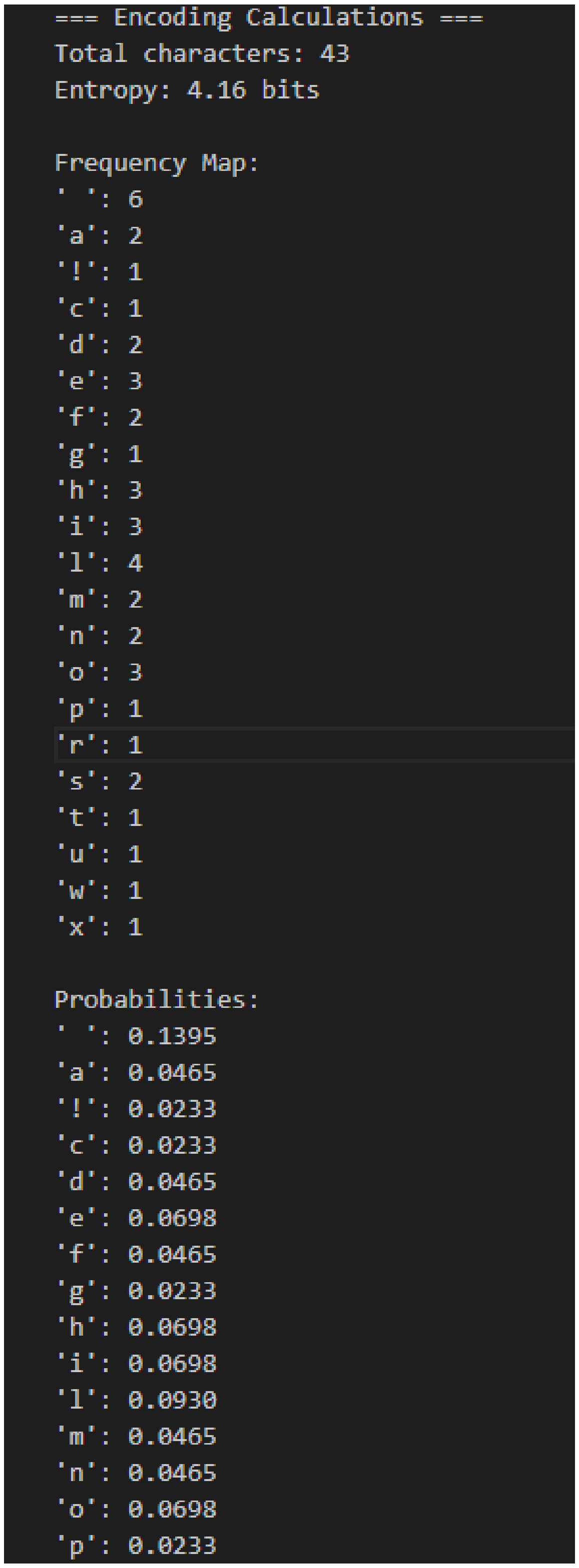
Step 4:



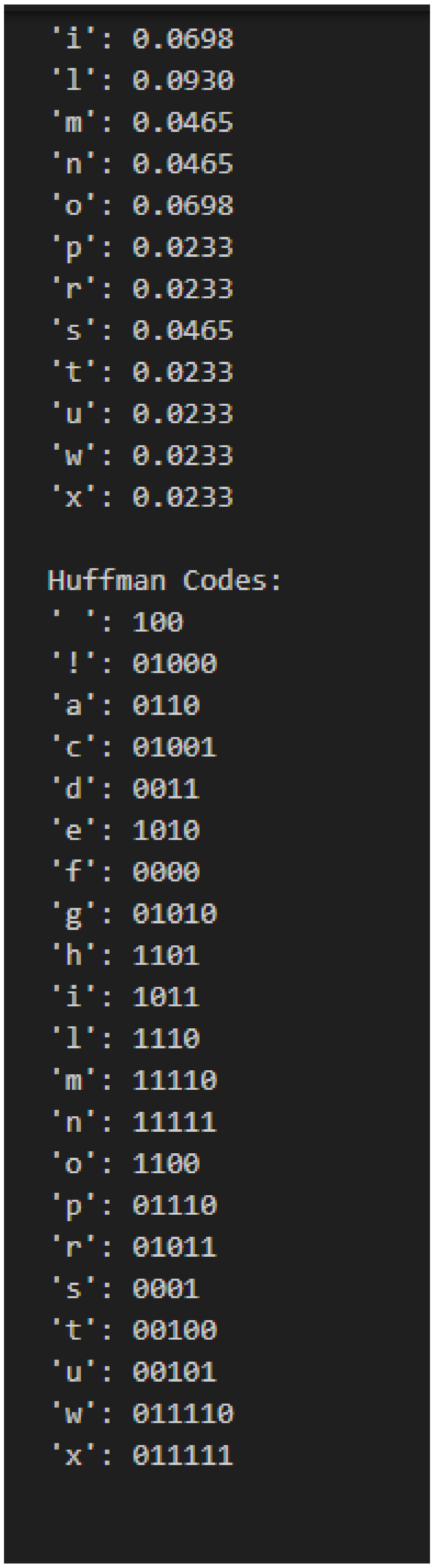
code_table.txt

```
bitLength=181,padding=3
32 100
33 01000
97 0110
99 01001
100 0011
101 1010
102 0000
103 01010
104 1101
105 1011
108 1110
109 11110
110 11111
111 1100
112 01110
114 01011
115 0001
116 00100
117 00101
119 011110
120 011111
```

Step 5:

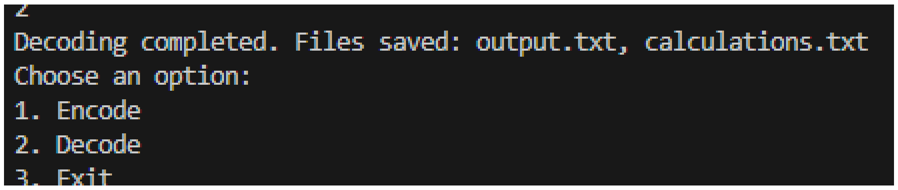


```
=== Encoding Calculations ===
Total characters: 43
Entropy: 4.16 bits

Frequency Map:
' ': 6
'a': 2
'!': 1
'c': 1
'd': 2
'e': 3
'f': 2
'g': 1
'h': 3
'i': 3
'l': 4
'm': 2
'n': 2
'o': 3
'p': 1
'r': 1
's': 2
't': 1
'u': 1
'w': 1
'x': 1

Probabilities:
' ': 0.1395
'a': 0.0465
'!': 0.0233
'c': 0.0233
'd': 0.0465
'e': 0.0698
'f': 0.0465
'g': 0.0233
'h': 0.0698
'i': 0.0698
'l': 0.0930
'm': 0.0465
'n': 0.0465
'o': 0.0698
'p': 0.0233
```



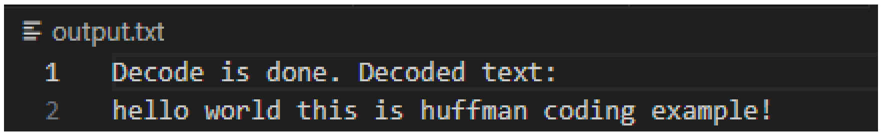
```
'i': 0.0698
'l': 0.0930
'm': 0.0465
'n': 0.0465
'o': 0.0698
'p': 0.0233
'r': 0.0233
's': 0.0465
't': 0.0233
'u': 0.0233
'w': 0.0233
'x': 0.0233

Huffman Codes:
' ': 100
'!': 01000
'a': 0110
'c': 01001
'd': 0011
'e': 1010
'f': 0000
'g': 01010
'h': 1101
'i': 1011
'l': 1110
'm': 11110
'n': 11111
'o': 1100
'p': 01110
'r': 01011
's': 0001
't': 00100
'u': 00101
'w': 011110
'x': 011111
```

Step 6:



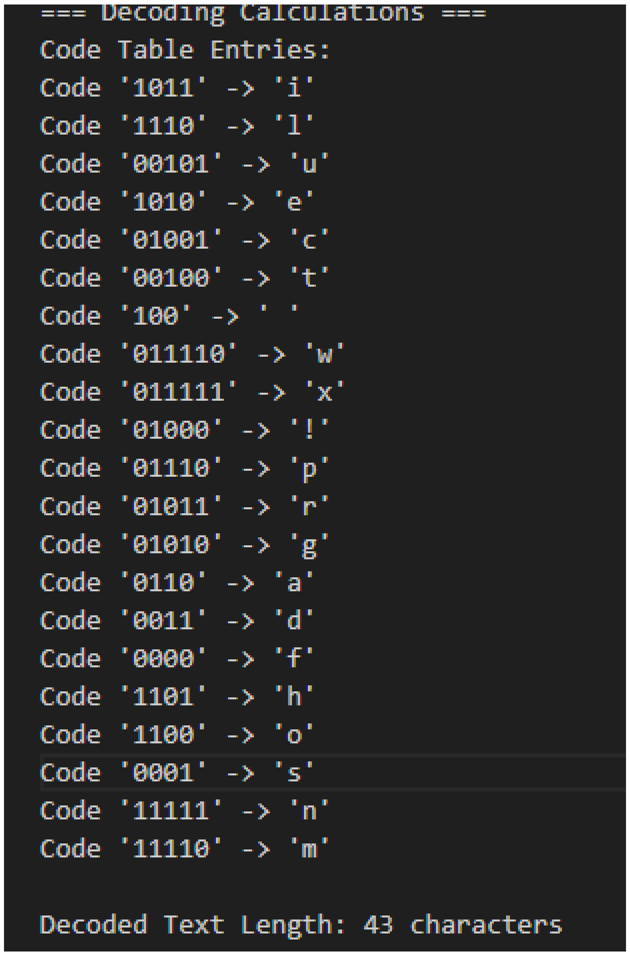
```
2
Decoding completed. Files saved: output.txt, calculations.txt
Choose an option:
1. Encode
2. Decode
3. Exit
```

Step 7:



```
output.txt
Decode is done. Decoded text:
hello world this is huffman coding example!
```

Step 8:



```
=== Decoding Calculations ===
Code Table Entries:
Code '1011' -> 'i'
Code '1110' -> 'l'
Code '00101' -> 'u'
Code '1010' -> 'e'
Code '01001' -> 'c'
Code '00100' -> 't'
Code '100' -> ' '
Code '011110' -> 'w'
Code '011111' -> 'x'
Code '01000' -> '!'
Code '01110' -> 'p'
Code '01011' -> 'r'
Code '01010' -> 'g'
Code '0110' -> 'a'
Code '0011' -> 'd'
Code '0000' -> 'f'
Code '1101' -> 'h'
Code '1100' -> 'o'
Code '0001' -> 's'
Code '11111' -> 'n'
Code '11110' -> 'm'

Decoded Text Length: 43 characters
```

Conclusion

This Huffman coding implementation effectively compresses text files and provides a structured way to manage Huffman trees and code tables. It ensures lossless compression and efficient storage using binary encoding.