1. draw_tree(node, x, y, dx): This function recursively draws the Huffman tree on a Tkinter canvas. The time complexity depends on the number of nodes in the tree, which is O(N), where N is the number of nodes in the tree.

2. printCodes(root, code_str): This function recursively prints the Huffman codes for each character in the tree. The time complexity is O(N), where N is the number of nodes in the tree.

3. storeCodes(root, code_str): This function recursively stores the Huffman codes for each character in the codes dictionary. The time complexity is O(N), where N is the number of nodes in the tree.

4. HuffmanCodes(size): This function builds the Huffman tree and calls storeCodes to store the Huffman codes. The time complexity is O(N log N), where N is the number of unique characters in the input string.
5. calcFreq(input_str, n): This function calculates the frequency of each character in the input string. The time complexity is O(n), where n is the length of the input string.

6. decode_file(root, s): This function decodes a Huffman-encoded string using the provided Huffman tree. The time complexity depends on the length of the input string, which is O(m), where m is the length of the encoded string.

7. compare_with_ascii(input_str): This function compares the Huffman codes with the ASCII codes of the characters in the input string. The time complexity is O(n), where n is the length of the input string.

8. visualize_frequency_distribution(input_str): This function calculates the frequency distribution of characters in the input string and visualizes it using a bar chart. The time complexity is O(n log n), where n is the length of the input string.

9. efficiency_comparison(input_str): This function compares the compression efficiency of Huffman coding with other compression techniques. The time complexity depends on the compression techniques used, but it is O(n) for calculating the Huffman codes.

10. compress_file(file_path): This function compresses a file using Huffman coding and writes the compressed data to a new file. The time complexity depends on the size of the input file but is O(n log n), where n is the size of the input file.

11. calculate_compression_ratio(input_str, encoded_str): This function calculates the compression ratio between the original input string and the encoded string. The time complexity is O(1).

Code Documentation:
Please find the code documentation below.
Title: Huffman Coding Implementation

Author: AmirHossein Sabry, Kimia Keivanloo

1. Introduction:

2. The Huffman Coding Implementation is a Python program that demonstrates the compression and decompression of data using the Huffman coding algorithm. It provides various functionalities such as building the Huffman tree, encoding and decoding data, visualizing frequency distribution, comparing with ASCII codes, and analyzing compression efficiency.

3. Dependencies:

    - heapq: This module provides an implementation of the heap queue algorithm, which is used to build the Huffman tree.
    - collections.defaultdict: This module provides a defaultdict class that allows the default values for keys that have not been set in a dictionary.
    - tkinter: This module provides the Tkinter library for creating a graphical user interface to visualize the Huffman tree.
    - colorama: This module provides support for colored terminal output.
    - art: This module provides ASCII art fonts for displaying the program title.

4. Functions:

- draw_tree(node, x, y, dx): Draws the Huffman tree on a Tkinter canvas.
- printCodes(root, code_str): Prints the Huffman codes for each character in the tree.
- storeCodes(root, code_str): Stores the Huffman codes for each character in the codes dictionary.
- HuffmanCodes(size): Builds the Huffman tree and stores the Huffman codes.
- calcFreq(input_str, n): Calculates the frequency of each character in the input string.
- decode_file(root, s): Decodes a Huffman-encoded string using the provided Huffman tree.
- compare_with_ascii(input_str): Compares the Huffman codes with the ASCII codes of the characters in the input string.
- visualize_frequency_distribution(input_str): Calculates the frequency distribution of characters in the input string and visualizes it using a bar chart.
- efficiency_comparison(input_str): Compares the compression efficiency of Huffman coding with other compression techniques.
- compress_file(file_path): Compresses a file using Huffman coding and writes the compressed data to a new file.
- calculate_compression_ratio(input_str, encoded_str): Calculates the compression ratio between the original input string and the encoded string.

5. Time Complexities:

    - draw_tree(node, x, y, dx): O(N), where N is the number of nodes in the tree.
    - printCodes(root, code_str): O(N), where N is the number of nodes in the tree.
    - storeCodes(root, code_str): O(N), where N is the number of nodes in the tree.
    - HuffmanCodes(size): O(N log N), where N is the number of unique characters in the input string.
    - calcFreq(input_str, n): O(n), where n is the length of the input string.
    - decode_file(root, s): O(m), where m is the length of the encoded string.
    - compare_with_ascii(input_str): O(n), where n is the length of the input string.
    - visualize_frequency_distribution(input_str): O(n log n), where n is the length of the input string.
    - efficiency_comparison(input_str): O(n) for calculating Huffman codes.
    - compress_file(file_path): O(n log n), where n is the size of the input file.
    - calculate_compression_ratio(input_str, encoded_str): O(1).
    - In summary, the overall time complexity of the code depends on the specific function being executed, ranging from O(n) to O(n log n).

6. Usage:

    - Modify the input_str variable to provide the input string for Huffman coding.
    - Run the script to see the results in the console and the graphical user interface.

7. Limitations and Future Enhancements:

    - The current implementation only supports text-based input.
    - The graphical user interface is simple and may not scale well for large trees.
    - The program can be enhanced to handle binary files and provide more advanced compression techniques.
    - Error handling and validation can be improved to handle edge cases and input errors.

8. Conclusion:

9. The Huffman Coding Implementation provides a practical example of Huffman coding for data compression. It demonstrates the construction of Huffman trees, encoding and decoding of data, comparison with ASCII codes, visualization of frequency distribution, and efficiency comparison with other compression techniques. The program can be extended and customized based on specific requirements and use cases.