Assignment 3a

Huffman Encoding

For this project, you will implement a program to encode text using Huffman coding. Huffman coding is a method of lossless (the original can be reconstructed perfectly) data compression. Each character is assigned a variable length *code* consisting of ’0’s and ’1’s. The length of the code is based on how frequently the character occurs; more frequent characters are assigned shorter codes.

The basic idea is to use a binary tree, where each **leaf node** represents a character and frequency count. The Huffman code of a character is then determined by the unique path from the root of the binary tree to that leaf node, where each ’left’ accounts for a ’0’ and a ’right’ accounts for a ’1’ as a bit. Since the number of bits needed to encode a character is the path length from the root to the character, a character occurring frequently should have a shorter path from the root to their node than an “infrequent” character, i.e. nodes representing frequent characters are positioned less deep in the tree than nodes for infrequent characters.

The key problem is therefore to construct such a binary tree based on the frequency of occurrence of characters. A detailed example of how to construct such a Huffman tree is provided here: Huffman_Example.pdf

Note:

- You must provide test cases for all functions (Only test the provided functions, with the exact name provided)
- Use descriptive names for data structures and helper functions.

2 Functions

The following bullet points provide a guide to implement some of the data structures and individual functions of your program. Start by creating a file **huffman.py** and add functions and data definitions to this file, if not otherwise stated. You should develop incrementally, building test cases for each function as it is implemented.

2.1 Count Occurrences: cnt_freq(filename)

- Implement a function called **cnt_freq(filename)** that opens a text file with a given file name (passed as a string) and counts the frequency of occurrences of all the characters within that file. Use the built-in Python List data structure of size 256 for counting the occurrences of characters. This will provide efficient access to a given position in the list. (In non-Python terminology you want an array.) You can assume that in the input text file there are **only 8-bit characters** resulting in a total of 256 possible character values. This function should return the 256 item list with the counts of occurrences.
    - Suppose the file to be encoded contained: ddddddddddddddddccccccccbbbbaaff
    - Numbers in positions of freq counts [96:104] = [0, 2, 4, 8, 16, 0, 2, 0]
    - For an empty file, this function should return a list of size 256 filled with all 0s.

2.2 Data Definition for Huffman Tree

A Huffman Tree is a binary tree of HuffmanNodes.

- A **HuffmanNode** class represents either a leaf or an internal node (including the root node) of a Huffman tree. A HuffmanNode contains a character (stored as an ASCII value) an occurrence count for that character, as well as references to left and right Huffman subtrees, each of which is a binary tree of HuffmanNodes. The character value and occurrence count of an internal node are assigned as described below. You may add fields in your HuffmanNode definition if you feel it is necessary for your implementation. **Do not change the names of the fields specified in the huffman.py starter file given to you via GitHub.**

2.3 Build a Huffman Tree

Since the code depends on the order of the left and right branches take in the path from the root to the leaf (character node), it is crucial to follow a specific convention about how the tree is constructed. To do this we need an ordering on the Huffman nodes.

- Start by defining the **__lt__ (self, other)** method for HuffmanNode objects that returns true if **self** should come before **other** when added to an OrderedList. A HuffmanNode '**a**' should come before HuffmanNode '**b**' if the occurrence count of '**a**' is smaller than that of '**b**'. In case of equal occurrence counts, break the tie by using the ASCII value of the character to determine the order. If, for example, the characters 'd' and 'k' appear exactly the same number of times in your file, then 'd' comes before 'k', since the ASCII value of character 'd' is less than that of 'k'.
- Write a function that builds a Huffman tree from a given list of the number of occurrences of characters returned by **cnt_freq()** and returns the root node of that tree.

Call this function **create_huff_tree(list_of_freqs)**.

- Start by creating an OrderedList (your implementation from Lab 4) of individual Huffman trees each consisting of a single HuffmanNode containing the character and its occurrence counts. Building the actual tree involves removing the two nodes with the lowest frequency count from the sorted list and connecting them to the left and right field of a new created Huffman Node as in the example provided. The node that comes before the other node should go in the left field.
- Note that when connecting two HuffmanNodes to the left and right field of a new parent node, that this new node is also a HuffmanNode, but does not contain an actual character to encode. Instead this new parent node should contain an occurrence count that is the sum of the left and right child occurrence counts as well as the **minimum** of the left and right character representation in order to resolve ties in the **__lt__** method.
- Once a new parent node has been created from the two nodes with the lowest occurrence count as described above, that parent node is inserted into the list of sorted nodes.
- This process of connecting nodes from the front of the sorted list is continued until there is a single node left in the list, which is the root node of the Huffman tree. **create_huff_tree(list_of_freqs)** then returns this node.
- If there are no entries in the list of occurrences passed to this function, it should return None.
- If there is only one entry in the list of occurrences passed to this function, the tree returned by this function will consist of a single node.

2.4 Build an Array for the Character Codes

- We have completed our Huffman tree, but we are still lacking a way to get our Huffman codes. Implement a function named **create_code(root_node)** that traverses the Huffman tree that was passed as an argument and returns an array (using a Python list) of 256 strings. Use the character's respective integer ASCII representation as the index into the arrary, with the resulting Huffman code for that character stored at that location. Traverse the tree from the root to each leaf node and adding a '0' when we go 'left' and a '1' when we go 'right' constructing a string of 0's and 1's. You may want to:
    - use the built-in '+' operator to concatenate strings of '0's and '1's here.
    - You may want to initialize a Python list of strings that is initially consists of 256 empty strings in **create_code**. When **create_code** completes, this list will store for each character (using the character's respective integer ASCII representation as the index into the list) the resulting Huffman code for the character. The code will be represented by a sequence of '0's and '1's in a string. Note that many entries in this list may still be **the empty string**. Return this list.

2.5 Huffman Encoding

- Write a function called **huffman_encode(in_file, out_file)** (use that exact name) that reads an input text file and writes to an output file the following:
    - A header (see below for format) on the first line in the file (should end with a newline)
    - Using the Huffman code, the encoded text into an output file.
- Write a function called **create_header(list_of_freqs)** that takes as parameter the list of freqs previously determined from cnt_freq(filename). The create_header function returns a string of the ASCII values and their associated frequencies from the input file text, separated by one space. For example, create_header(list_of_freqs) would return "97 3 98 4 99 2" for the text "aaabbbbcc"
- The **huffman_encode** function accepts two file names in that order: input file name and output file name, represented as strings. If the specified output file already exists, its old contents will be erased. See example files in the test cases provided to see the format.
- Note: Writing the generated code as a string for each character into a file will actually enlarge the file size instead compress it. The explanation is simple: although we encoded our input text characters in sequences of '0's and '1's, representing actual single bits, we write them as individual '0' and '1' characters, i.e. 8 bits.
- To actually obtain a compressed the file you will also write the '0' and '1' characters as individual bits. Do this by taking the **out_file** string and:
    - Add _compressed to the name of the file before the .txt file. (Ex: if out_file is named "output.txt", you will create the string "output_compressed.txt")
    - Use the filename created above to create a HuffmanBitWriter object using the huffman_bit_writer module.
    - Use the methods in the HuffmanBitWriter object to write the header and individual bits of the character codes to the compressed file.

3 Tests

- Write sufficient tests using unittest to ensure full functionality and correctness of your program. Start by using the supplied **huffman_tests.py**, and the various text files (e.g. **file1.txt** and **file1_soln.txt)** to begin testing your programs. You may want to initially comment out some of the tests, or ignore failures when testing functionality that hasn't been fully implemented.
- When testing, always consider *edge conditions* like the following cases:
    - If the input file consists only of some number of a single character, say "aaaaa", it should write just that character followed by a space followed by the number of occurrences with a newline: "97 5\n"
    - In case of an *empty* input text file, your program should also produce an *empty* file.
    - If an input file does not exist, your program should raise a FileNotFoundError.
- Your tests are only given a few seconds to run by the automated grader. **DO NOT test large files like War and Peace** in the tests you submit for grading.

4 Some Notes

- When writing your own test files or using copy and paste from an editor, take into account that most text editors will add a newline character to the end of the file. If you end up having an additional newline character '\n' in your text file, that wasn't there before, then this '\n' character will also be added to the Huffman tree and will therefore appear in your generated string from the Huffman tree. Different operating systems have different codes for "newline". Windows uses '\r\n' = 0x0d 0x0a, while Unix and Mac use '\n' = 0x0a. It is always useful to use a hex editor to verify why certain files that appear identical in a regular text editor are actually not identical.

5 Submission

You must submit (commit and push) the following files:

- **huffman.py**, containing the functions specified and any helper functions necessary
    - **cnt_freq(filename):** returns a Python list of 256 integers representing the frequencies of characters in file (indexed by ASCII value of characters)
    - **create_huff_tree(list_of_freqs):** returns the root node of a Huffman Tree, a Huffman node object
    - **create_code(root_node):** returns a Python list of 256 strings representing the code for each character (indexed by ASCII value of the character). Note: Use an **empty string** to represent the code for ASCII characters that do not appear in the file being compressed.
    - **create_header(list_of_freqs):** returns a header for the output file with ascii values and their associated counts, space separated.
    - **huffman_encode(in_file, out_file):** encodes in_file and writes the it to out_file

- **huffman_tests.py**, containing testcases for the functions specified by the assignment
    - This file should contain tests only for the functions specified above and should only test functionality required by the specification. These tests must run properly on **any valid solution** and will be tested to see if they can catch bugs in incorrect solutions.

- **huffman_helper_tests.py**, containing all testcases used in developing your solution
    - This file should contain all tests for your solution, including tests for any helper functions that you may have used. These tests will be used to verify that you have 100% coverage of your solution. Your solution must pass these tests.

- **ordered_list.py**, your correct implementation of the OrderedList class from Lab 4.

- **huffman_bit_writer.py**, unmodified, as provided to you.