Project 3a — Huffman Encoding

Please read the entire document carefully! I would suggest reading the entire document *before* writing any code.

1 Introduction

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 *code* consisting of `'0'`s and `'1'`s (i.e., binary digits or bits). 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'`. 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 (i.e., should be higher in the tree) compared to an "infrequent" character (which should be lower in the tree).

The key problem is therefore to construct such a binary tree based on the frequency of occurrence of characters. For a detailed example of how to construct such a tree, see the Huffman Example document on Canvas.

BE SURE that you understand the example in the document *before* attempting to implement any of the following. If you do not understand what the algorithm should be doing, you will not be able to write code for it.

2 Specification

In the provided `huffman.py`, you will implement the following:

2.1 Count Frequencies

You will implement a function called `count_frequencies` that takes as an argument the name of a text file and counts the number of occurrences of all the characters within that file.

Use the builtin Python `list` data structure of size 256 for counting the occurrences of characters. The indices into the list will be the ASCII values of the characters (use the Python builtin `ord` to get this value for a given character) and the value at a given index will be the frequency with which that character occurred.

For example, suppose that the file to be encoded contained `"aaabbbbc"`. Then this function will return a list containing mostly zeros, *except* indices 97–99 will be `[3, 4, 1]`.

For an empty file, this function should return a list containing 256 zeros.

2.2 Data Definition for Huffman Tree

Most of a `HuffmanNode` class is provided. These represent nodes in your Huffman Tree. You will need to implement the `__eq__` and `__lt__` methods so that these nodes may be places in your `OrderedList` (from Lab 4). See the example for details of when one tree should be considered "less than" another tree.

2.3 Building the Huffman Tree

You will implement a function `build_huffman_tree` that takes as an argument a list of frequencies and builds and returns the resulting Huffman Tree. You will do this by:

- Creating an `OrderedList` (from Lab 4) of individual Huffman Trees each consisting of a single `HuffmanNode` containing the character and its frequency. We should only include characters with non-zero frequency here.
- While the list contains more than one tree, you will remove the two least frequent trees, join them together into one tree, and put the resulting tree back into the ordered list.
  - In our implementation, when joining two trees, the "lesser" of the trees *must* go on the left. See the example document for details.
- Once the list only has a single tree, this is our Huffman Tree!

Note that when connecting two trees, this new larger tree does not represent a single character, but rather jointly represents all the characters in the tree. As such, the new node should have the sum of the frequencies of its children. For our implementation, we will also store the smaller of the characters in this new node to continue to enable tie breaking. See the example document for details.

Some examples to consider:

- If the frequency list was all zeros, your function should return an empty tree.
- If there was only one character with non-zero frequency, the resulting tree will only have a single node.

2.4 Creating the Codes

You will implement a function `create_codes` that takes as an argument a Huffman Tree and determines the code for each character in the tree.

Use the builtin Python `list` data structure of size 256 for storing the codes. The indices into the list will be the ASCII values of the characters and the value at a given index will be the code for that character. Any character not in the tree should have an empty code (i.e., `''`).

You should accomplish this by traversing the tree, building the codes on the way down. Each time you traverse to the left, you will add a `'0'` and each time you traverse to the right, you will add a `'1'`. You may find it useful to use the builtin `'+'` operator to concatenate strings.

2.5 Creating the Header

When we decode a file (coming in Project 3b), we need to have the Huffman Tree available (see the example document for details). This means that we have to store enough information into the encoded file to reconstruct the tree. For our implementation we will accomplish this by storing the frequencies at the top of the file.

You will implement a function `create_header` that takes as an argument a list of frequencies and returns a string containing the frequency data to be stored at the top of the encoded file.

This will be a string of the ASCII values and their associated frequencies, separated by spaces. You should only include characters with non-zero frequency, and they should be ordered by ASCII value. For example, if the original file had contained `"aaacbbbb"`, then this function would return the string `"97 3 98 4 99 1"`

2.6 Huffman Encoding

You will implement a function `huffman_encode` that takes as arguments two file names (for the input and output), encodes the input file, and stores the resulting encoded data in the output file.

This encoded data has two pieces:

- A header on the first line of the file (see Section 2.5 for details), ending with a newline character.
- The encoded data, where each character is replaced with its Huffman Code (see the example document for details).

You may notice after doing this that the resulting file is larger than the original! What lousy compression! This is because although we encoded our input text characters in sequences of `0`s and `1`s *representing* bits, we wrote them to the file as the text characters `0` (ASCII value 48) and `1` (ASCII value 49). So, the result is roughly 8 times larger than it should have been. Writing the "true" binary data to the file is a bit of a hassle, so we will forgo it.

2.7 More Notes

When writing your own text files, or copying and pasting text around, take into account that most text editors will add a new line character (`'\n'`, ASCII value 10) to the end of the file. This is now a character in your file that will be encoded along with everything else. So, it will end up in your Huffman Tree, and it's encoded bits will end up in your resulting file. This isn't *wrong*, but it makes testing a little harder if the file you're encoding for testing isn't what you thought it was.

Additionally, the exact behavior of newlines is different for Windows vs. every other operating system in the world. On a Windows machine, a newline is actually two characters `"\r\n"` (ASCII values 13 and 10).

If you're seeing your Huffman Trees getting nodes with values 10 and 13, this isn't necessarily wrong, it may just mean that the files you're compressing have newlines.

3 Testing

Writing tests for functions that operate on files is difficult. So I've provided you with an example (along with some basic tests of the rest of your functions).

When testing, always consider *edge cases* like the following cases:

- If the input file consists only of some number of a single character, say `"aaaaa"`, what do you think the result should look like? What is the "code" for the letter `'a'`?
- If the input file is empty (as in 0 bytes large), what should the result look like?

In your `huffman_tests.py`, you should *only* test the functions that are required (for which stubs were provided). I will be running your tests against my code for this projects (similar to project 1) and if you're testing a helper function, it would crash my tests (because I most likely won't have written the exact same helper functions).

That said, you are still *required* to achieve 100% test coverage. This means that every line of code in your functions *must* be executed at some point in at least one of your tests. If you would like to write tests for your helper functions directly, you may create a file `huffman_helper_tests.py` for your helper function tests.

As discussed in the syllabus, when I grade your code, you will ordinarily receive feedback regarding any tests that fail, *unless* you do not have 100% test coverage. In the event you do not have 100% test coverage, the only feedback you will receive is that you need to do more testing. I don't want you using my grading script to do your testing in the last day.

4 GitHub Submission

Push your finished code back to GitHub. Refer to Lab 0, as needed, to remember how to push your local code.

The files you need to have submitted are:

- `ordered_list.py`
  - Your correct implementation of the `OrderedList` from Lab 4.
- `huffman.py`
  - Contains the functions specified above, and any helper functions that you find necessary.
- `huffman_tests.py`
  - Contains tests for all your functions specified by the assignment. These should run on *any* correct implementation of this project. As such, you should not include any tests for helper functions.
- `huffman_helper_tests.py`
  - Optional file containing tests for all your helper functions. These tests will be used in conjunction with your `huffman_tests.py` to check for 100% coverage.
- Any text files you use for testing.
  - If you use a text file for testing, you'll need to submit it so that your tests run on my end.