CSC148, Assignment 2
data compression with Huffman codes
due March 24, 10 p.m.

deadline to declare your assignment team: March 20, 10 p.m.

Overview

Data compression involves reducing the amount of space taken up by files. Anyone who has listened to an MP3 file or extracted a file from a `zip` archive has used compression. Reasons for compressing a file include saving disk space and reducing the time required to transfer the file to another computer.

There are two broad classes of compression algorithms: **lossy** compression and **lossless** compression. Lossy compression means that the file gets smaller, but that some information is lost. MP3 is a form of lossy compression: it saves space by throwing away some audio information, and therefore the original audio file can not be recovered. Lossless compression means that the file gets smaller **and** that the original file can be recovered from the compressed file. FLAC is a form of lossless audio compression: it can't save as much space as MP3, but it does let you perfectly recover the original file.

In this assignment, we'll be working with a lossless kind of compression called Huffman coding. When you're finished the assignment, you'll be able to compress a file, and then decompress that file to get back the original file.

Background

Fixed- and Variable-Length Codes

Suppose that we had an alphabet of four letters: **a**, **b**, **c**, and **d**. Computers store only 0s and 1s (each 0 and 1 is known as a **bit**), not letters directly. So, if we want to store these letters in a computer, it is necessary to choose a mapping from these letters to bits. That is, it's necessary to agree on a unique code of bits for each letter.

How should this mapping work? One option is to decide on the length of the codes, and then assign letters to unique codes in whatever way we wish. Choosing a code length of 2, we could assign codes as follows: **a** gets 00, **b** gets 01, **c** gets 10, and **d** gets 11. (Note that we chose a code length of 2 because it is the minimum that supports an alphabet of four letters. Using just one bit gives you only two choices — 0 and 1 — and that isn't sufficient for our four-letter alphabet.) How many bits are required to encode text using this scheme? If we have the text aaaaa, for example, then the encoding takes 10 bits (two per letter).

Another option is to drop the requirement that all codes be the same length, and assign the shortest possible code to each letter. This might give us the following: **a** gets 0, **b** gets 1, **c** gets 10, and **d** gets 11. Notice that the codes for **a** and **b** are shorter than before, and that the codes for **c** and **d** are the same length as before. Using this scheme, we'll very likely use fewer bits to encode text. For example, the text aaaaa now takes only 5 bits, not 10!

Unfortunately, there's a catch. Suppose that someone gives us the encoding 0011. What text does this code represent? It could be aabb if you take one character at a time from the code. But there are other possibilities, too. It could also equally be aad if you break up the code into the pieces 0 (**a**), 0 (**a**), and 11 (**d**). Our encoding is ambiguous! ... Which is really too bad, because it seemed that we had a way to reduce the number of bits required to encode text. Is there some way we can still proceed with the idea of using variable-length codes?

To see how we can proceed, consider the following encoding scheme: **a** gets 1, **b** gets 00, **c** gets 010, and **d** gets 011. The trick here is that no code is a prefix of any other code. This kind of code is called a **prefix code**, and it leads to unambiguous decoding. For example, 0011 now unambiguously decodes to the text baa; there is no other possibility.

What we have here is seemingly the best of both worlds: variable-length codes (not fixed-length codes), and unambiguous decoding. However, note that some of our codes are now longer than before. For example, the code for **c** is now 010. That's a three-bit code, even longer than the 2-bit codes we were getting with the fixed-length encoding scheme. This would be OK if **c** letters didn't show up very often: if **c** letters were rare, then we wouldn't care much that they cause us to use 3 bits. If **c** letters **did** show up often, though, then we'd worry because we'd be spending 3 bits for every occurrence of **c**.

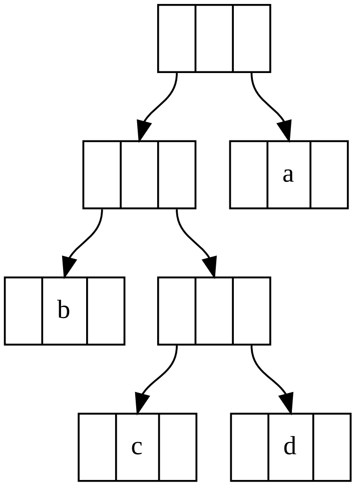
In general, then, we want to have short codes for frequently-occurring letters and longer codes for letters that show up less often. For example, if the frequency of **a** in our text is 10000 and the frequency of **b** in our text is 50, we'd hope that the code for **a** would be much shorter than the code for **b**.

Goal of Algorithm

Our goal is to generate a best prefix code for a given text. The best prefix code depends on the frequencies of the letters in the text. What do we mean by "best" prefix code? It's the one that minimizes the average number of bits per symbol or, alternately, the one that maximizes the amount of compression that we get. If we let C be our alphabet, $f(x)$ denote the frequency of symbol x, and $d(x)$ denote the number of bits used to encode x, then we're seeking to minimize the sum $\sum_{x \in C} f(x)d(x)$. That is, we want to minimize the sum of bits over all symbols, where each symbol contributes its frequency times its length.

Let's go back to our four-letter alphabet: **a**, **b**, **c**, and **d**. Suppose that the frequencies of the letters in our text are as follows: **a** occurs 600 times, **b** occurs 200 times, **c** occurs 100 times, and **d** occurs 100 times. A best prefix code for these frequencies turns out to be the one we gave earlier: **a** gets 1, **b** gets 00, **c** gets 010, and **d** gets 011. Calculate the above sum and you'll get a value of 1600. There are other prefix codes that are worse; for example: **a** gets 110, **b** gets 0, **c** gets 10, and **d** gets 111. Calculate the sum for this and convince yourself that it is worse than 1600!

Huffman's algorithm makes use of binary trees to represent codes. Each leaf is labeled with a symbol from the alphabet. To determine the code for such a symbol, trace a path from the root to the symbol's leaf. Whenever you take a left branch, append a 0 to the code; whenever you take a right branch, append a 1 to the code. The Huffman tree corresponding to the best prefix code above is the following:



Huffman's algorithm generates a tree corresponding to a best prefix code. You'll see proofs of this fact in future courses; for now, you'll implement Huffman's algorithm.

Preliminaries

For this assignment, we're asking you to do some background reading on several topics that are not explicitly taught in the course. (We don't often give students sufficient opportunity to research on their own and build on their available knowledge. This assignment is an attempt to remedy this.) Here's what you'll want to learn:

- Huffman's algorithm. You'll want to understand how Huffman's algorithm works on a conceptual level before working on the implementation.
- Python `bytes` objects. We'll be reading and writing binary files, so we'll be using sequences of bytes (not sequences of characters). `bytes` objects are like strings, except that each element of a `bytes` object holds an integer between 0 and 255 (the minimum and maximum value of a byte, respectively). We will consider each byte in an uncompressed file as a symbol.
- Binary numbers. Background on binary numbers will be useful, especially when debugging your code. Although it's not necessarily crucial for this assignment, you might want to also familiarize yourselves with "endianness" (little endian in particular), to get an idea of how computers store a sequence of bytes in memory.

You're strongly encouraged to find and use online resources to learn this background material. Be sure to cite your sources in your assignment submission; include Python comments giving the locations of resources that you found useful.

Your Task

Your task is to complete the functions in `huffman.py`. Ultimately, you will be able to call the `compress` and `uncompress` functions to compress and uncompress a file, respectively.

The code in huffman.py uses two types of nodes from nodes.py. The first, HuffmanNode, is a node in a Huffman tree. Huffman trees are used to compress and uncompress data. The second, ReadNode, is the representation of a node in a compressed file. It will be necessary to reconstruct the root HuffmanNode of a Huffman tree from the ReadNodes stored in the file.

We now give a suggested order for writing the functions in huffman.py.

Building the Tree

- Implement make_freq_dict. This function generates and returns a frequency dictionary.
- Implement huffman_tree. This function takes a frequency dictionary and returns the root HuffmanNode of the Huffman tree corresponding to a best prefix code. (There's a tricky case to watch for here: you have to think of what to do when the frequency dictionary has only one symbol!)
- Implement get_codes. This function takes a Huffman tree and returns a dictionary that maps each byte in the tree to its code. The dictionary returned by get_codes is very useful for compressing data, as it gives the mapping between a symbol and the encoding for that symbol.
- Implement number_nodes. This assigns a number to each internal node. These numbers will be written when compressing a file to help represent relationships between nodes.
- Now is a good time to implement avg_length. This function returns the average number of bits required to compress a symbol of text.

Compressing Data

Now we know how to generate a Huffman tree, and how to generate codes from that tree. Function generate_compressed uses the given frequency dictionary to generate the compressed form of the provided text.

Every byte comprises 8 bits. For example, 00001010 is a byte. Recall that each symbol in the text is mapped to a sequence of bits, such as 00 or 111. generate_compressed takes each symbol in turn from text, looks-up its code, and puts the code's bits into a byte. Bytes are filled-in from bit 7 (leftmost bit) to bit 0 (rightmost bit). Helper function bits_to_byte will be useful here. You might also find it helpful to use byte_to_bits for debugging; it gives you the bit-representation of a byte.

Note that a code may end up spanning multiple bytes. For example, suppose that you've generated six bits of the current byte, and that your next code is 001. There's room for only two more bits in the current byte, so only the 00 can go there. Then, a new byte is required, and its leftmost bit will be 1.

Implement generate_compressed.

Writing a Compressed File

generate_compressed gives us the compressed version of the text. Suppose that you send someone a file containing that compressed text. How can they decompress that result to get back the original file? Unfortunately, they can't! They won't have the Huffman tree or codes used to compress the text, so they won't know how the bits in the compressed file correspond to text in the original file.

This means that some representation of the tree must also be stored in the compressed file. There are many ways to do this: we have chosen one that is suboptimal but hopefully instructive (Dan Z recalls the technique from authors of old PC games who used Huffman coding to compress a game onto a single disc).

Follow along in function compress throughout this discussion.

Here is what we generate so that someone can later recover the original file from the compressed version:

- (1) The number of internal nodes in the tree
- (2) A representation of the Huffman tree
- (3) The size of the uncompressed file
- (4) The compressed version of the file

Functions are already available (in starter code or written by you) for all steps except (2). Here is what we do for step 2, in function `tree_to_bytes`:

- Internal nodes are written out in postorder.
- Each internal node of the tree takes up four bytes. Leaf nodes are not output, but instead are stored along with their parent nodes.
- For a given internal node n, the four bytes are as follows. The first byte is a 0 if the left subtree of n is a leaf; otherwise, it is 1. The second byte is the symbol of the left child if the left subtree of n is a leaf; otherwise, it is the node number of the left subtree. The third and fourth bytes are analogous for the right subtree of n. That is, the third byte is a 0 if the right subtree of n is a leaf; otherwise, it is 1. The fourth byte is the symbol of the right child if the right subtree of n is a leaf; otherwise, it is the node number of the right subtree.

Implement `tree_to_bytes`.

Decompressing Text

There are two functions that work together to decompress text: a `generate_tree_xxx` function to generate a tree from its representation in a file, and a `generate_uncompressed` function to generate the text itself.

First, the functions to generate a Huffman tree from its file representation. Note that you are being asked to write two **different** `generate_tree_xxx` functions. Once you get one of them working, you can successfully uncompress text. But we're asking for both, so don't forget to do the other one!

- `generate_tree_general` takes a list of nodes representing a tree in the form that it is stored in a file. It reconstructs the Huffman tree and returns its root `HuffmanNode`. This function assumes **nothing** about the order in which the nodes are given in the list. As long as it is given the index of the root node, it will be able to reconstruct the tree. This means that the tree nodes could have been written in postorder (as your assignment does), preorder, inorder, level-order, random-order, whatever. Note that the number of each node in the list corresponds to the index of the node in the list. That is, the node at index 0 in the list has node-number 0, the node at index 1 in the list has node-number 1, etc.
- `generate_tree_postorder` takes a list of nodes representing a tree in the form that it is stored in a file. It reconstructs the Huffman tree and returns its root `HuffmanNode`. This function assumes that the tree is given in the list in postorder (i.e. in the form generated by your compression code). Unlike in `generate_tree_general`, you **are not** allowed to use the node-numbers stored in the `ReadNode`s to reconstruct the tree (in fact, these node-numbers may not even be set correctly — check the docstring for an example). That is, if byte 1 of an internal node is 1 (meaning that the left subtree is not simply a leaf), then you are not allowed to look at byte 2. Similarly, if byte 3 is 1 (meaning that the right subtree is not simply a leaf), then you are not allowed to look at byte 4.

To test these two functions as you write them, notice that `tree_to_bytes`, combined with `bytes_to_nodes`, generates tree representations in exactly the form required by `generate_tree_postorder`.

To test generate_tree_general, things are not so simple. (You have no function that generates a tree in anything except postorder. And generate_tree_general is supposed to work no matter what, as long as you have the index of the root node in the list.) You could, for example, write new number_nodes functions that number the trees in different ways (e.g. preorder, random-order, etc.), and write corresponding tree_to_bytes functions that write out the tree according to the order specified by the numbering.

Next, the function to decompress text using a Huffman tree:

- generate_uncompressed takes a Huffman tree, compressed text, and number of bytes to produce, and returns the decompressed text. There are two possible approaches: one involves generating a dictionary that maps codes to symbols, and the other involves using the compressed text to traverse the tree and discovering a symbol whenever a leaf is reached.

Another Function

For more practice, we are requiring you (as part of your mark) to implement improve_tree. This has nothing to do with the rest of the program, so it won't be called by any of your other functions. It is still a required function for you to write, however.

Suppose that you are given a Huffman tree and a frequency dictionary. We say that the Huffman tree is suboptimal for the given frequencies if it is possible to swap a higher-frequency symbol lower in the tree with a lower-frequency symbol higher in the tree.

improve_tree performs all such swaps, improving the tree as much as possible without changing the tree's shape. Note, of course, that Huffman's algorithm will never give you a suboptimal tree, so you can't hope to improve those trees. But you **can** imagine that someone gives you a suboptimal tree, and you want to improve it as much as possible without changing it's shape.

Testing Your Work

In addition to the doctests provided in huffman.py, you are encouraged to test your functions on small sample inputs. Make sure that your functions are doing the right thing in these small cases. Finding a bug on a huge example tells you that you have a bug but doesn't give you much information about what is causing the bug.

Once you've suitably tested your functions, you can try the compressor and decompressor on some files. You'll be happy to know that you can compress and decompress arbitrary files of any type — although the amount of compression that you get will vary widely!

When you run huffman.py, you have a choice of compressing or uncompressing a file. Choose an option and then type a filename to compress or decompress that file. When you compress file a, a compressed version will be written as a.huf. Similarly, when you decompress file a.huf, a decompressed version will be written as a.huf.orig. (We can't decompress file a.huf back to simply a, because then your original a file would be overwritten!)

We've provided some sample files that you can try to compress:

- book.txt is a public-domain book from Project Gutenberg. Text files like this compress extremely well using Huffman compression.

- dan.bmp is a picture of Dan Z in bitmap (bmp) format. This is an uncompressed image format and so compresses very well.

- music.wav is a few seconds of a song from Mystic Ark (one of Dan Z's favourite game soundtracks). The song is in wav format; this file compresses terribly with Huffman compression.

- music.mp3 is a few seconds of a song from Ys V (another one of Dan Z's favourite game soundtracks). If you thought that the .wav file compressed badly, try compressing this .mp3!

Throw other files at your compressor and see what kind of compression you can get!

Property Tests

In file test_huffman_properties.py, we have provided some **property tests** that you can run on your code as you start completing functions. Please install the hypothesis Python package to use these property tests at home (hypothesis is already installed for you in the labs).

What are property tests? Regular unittests work like this: you provide parameter values and expected return value for each case, and unittest determines whether your function works appropriately in those cases. Property tests, by contrast, work like this: you tell hypothesis about relationships between parameter values and return values, and hypothesis automatically generates tons of examples that try to falsify the correctness of your function. So, with hypothesis, you focus on general behaviour of your functions rather than specific behaviour on prespecified examples.

For a little on the syntax that hypothesis uses, take a look at the first class in test_huffman_properties.py. The @given line specifies that integers between 0 and 255 should be passed to the test_byte_to_bits function as parameter b. We then have two tests (assertXxx statements) that must pass, no matter the value of b: byte_to_bits must return an object where everything in it is a '0' or '1', and it must return something of length 8. Of course, this particular property test is for a starter code function, not one that you write; but later in the file there are some property tests for the functions that you write.

The property tests that we've given you do **not** fully characterize what it means for a function to be correct. They're just a starting point for testing. You're encouraged to write further property tests to help increase your confidence in your functions, though any tests that you write are **not** being marked.

Declaring Your Assignment Team

You may do this assignment alone, with one other student, or with two other students. Your partner(s) may be from any section of the course at UTM. You must declare your team (even if you are working alone) using the MarkUs online system.

Navigate to the MarkUs page for the assignment and find "Group Information". If you are working alone, click "work individually". If you are working in a team:

First: one of you must invite" the other(s) to be partners, providing MarkUs with their utorids.

Second: the invited students must accept the invitation.

Important: there must be **only** one inviter; the other group member(s) accept **after** being invited.

To accept an invitation, find "Group Information" on the Assignment page, find the invitation listed there, and click on "Join".

Submitting Your Work

Submit your huffman.py file on MarkUs (no other .py files will be accepted). To submit the file, click on the "Submissions" tab near the top, click "Add a New File", and type a filename or click "Browse" to find the file. Finally, click “Submit”. You can submit a new version of a file later (before the deadline, of course); look in the “Replace” column.

In addition to `huffman.py`, you are free to submit an `info.txt` file that tells us what is working and not working in your assignment, and anything else that you think we’d be interested in reading about your assignment! For example, what other files did you compress, how good is the compression, how does the compression ratio compare to other tools, etc.

If you are working in a group of 2 or 3, then only one team member should submit the assignment. Everyone in the team will get credit for the work.

Marking Scheme

Here is how your assignment will be marked.

Passing the provided property tests is worth 30%. Passing new property tests that we will run (that fully characterize the correctness of your code) is worth 40%. Passing our traditional unittests (where we hard-code particular parameter and return values) is worth 30%.