CS 120 (Spring 22): Introduction to Computer Programming II

Short Project #9 - Huffman Codes

due at 5pm, Thu 24 Mar 2022

REMEMBER: The `itertools` and `copy` libraries in Python are **banned.**

1 Overview

You will write a series of functions, which do most of the parts of a Huffman encoder/decoder. You will declare all of these functions in the file `huffman_tools.py` .

2 What are Bits and Bytes?

All data inside a computer is made of up "bits" - that is, individual ones and zeroes. We organize these bits into groups, called "bytes" - a byte is made of up 8 bits.

For most programs, a byte is the smallest unit of memory we'll ever consider. For instance, a string in memory is (usually) made up of an array of bytes, with each byte representing a single character. When we build larger things, we typically measure them in terms of bytes; for instance, in some languages an integer is stored in 4 bytes (that is, 32 bits) of memory.

From time to time, however, it's useful to break our bytes down into bits, and look at each one and zero in isolation. We'll be doing a very basic version of that for this project.

2.1 Binary Data

When we need to look at individual bits in our data, we often write out the ones and zeroes by hand. For instance, it so happens that the number 7, stored into a byte, is encoded by the following bits:

```
00000111
```

Most of the time, we just think of this variable as "seven," but in fact it is 8 different bits, set in a certain pattern.

In later classes, you will learn how to read and write individual bits inside of a number. But for this project, we're going to do something simpler: we will ask Python to turn our integers into strings - and then we'll read the ones and zeroes as text. This way of doing this is not very efficient, but it's easier to understand. And it will be easy to test with my testcases!

(spec continues on the next page)

3 Huffman Codes

Huffman coding (`https://en.wikipedia.org/wiki/Huffman_coding`) is one of the famous methods that computer scientists use to compress a stream of data - whether that data is text, integers, or just about anything. The central concept of Huffman coding is that we analyze the text to figure out which symbols are common, and which ones are rare; we use short codes to represent the very common symbols, and long codes to represent the rare ones.

This idea isn't just for computers. Morse code does the same thing - the two most common letters (E,T) get one-element codes; rare letters, like Y,Z,Q get four-element codes, with various others in-between. In an ordinary telegram, the total amount of dots and dashes is reduced, because things that happen often are very short.

In computer files, we have one additional requirement. With Morse code, it's OK to have symbols which overlap a bit: for instance, the code for E is "dot," while the code for A is "dot dash." In Morse code, this works because we can insert small pauses between the letters. But if we have a long string of only ones and zeroes (as in a bianry file), we can't insert a "pause" - we can only have another one or zero.

Therefore, Huffman codes are always non-overlapping. In a Huffman code, if you assigned the sequence `0` to the letter E, then it would be illegal to ever have any other sequecnce that started with `0`. On the other hand, if you assigned the sequence `00` to E, then you could have some sequences that started with `01`, `10`, or `11`.

(spec continues on the next page)

3.1 Breaking Up The Stream

Ordinarily, we organize data in our computers into "bytes" - 8 bits of information at a time. When you have a string in memory, for instance, each character is typically represented by a single, 8-bit byte. And when we read and write files, we will always be reading and writing byte-by-byte.

But when we use a Huffman code, we have to break things up differently, because some symbols will be very short (just a couple bits), and others will be long (maybe even more than 8 bits, sometimes). This is easiest to see if we do the encoding step first. Let's imagine that we have a very simple code, which only supports 8 possible symbols in our text:

```
A         01
B         1010
E         00
G         110
R         111
V         10110
<space>   100        # will be ' ' in your actual code
<END>     10111      # will be "<END>" in your actual code
```

Now, let's encode the text `"EAGER BEAVER"` . We turn each symbol into its matching encoding, adding a special `<END>` symbol at the end:

```
 E  A  G   E  R        B   E  A   V    E  R  <END>
00 01 110 00 111 100 1010 00 01 10110 00 111 10111
```

We then organize it into 8-bit groups. (Note that the last group didn't have 8 bits. We added zeroes to fill it out. This is the reason why having an `<END>` mark at the end of the string can be very important, sometimes.)

```
 E  A  G   E  R        B   E  A   V    E  R  <END>
00 01 110 00 111 100 1010 00 01 10110 00 111 10111
00 01 110 0
          0 111 100 1
                    010 00 01 1
                              0110 00 11
                                       1 10111 00
```

Pay attention. How many bytes does that take up? We could encode the entire string in only 5 bytes - even though the string has 12 characters.

Of course, in practical applications, we would have to support more than 8 symbols, and so the symbols would get somewhat longer - but we can still (sometimes) get an excellent compression ratio.

(spec continues on the next page)

3.2 Decoding

Can we decode the stream? Of course! Since no two symbols can overlap, we can take a stream of bytes, break them into bits, and then find the symbols inside them. If we start with these bytes:

```
00011100 01111001 01000011 01100011 11011100
```

we search the beginning of the string to find the first match (`00`), which is the letter E:

```
00        E
  011100 01111001 01000011 01100011 11011100
```

Next, we find an A, since the next match is `01`

```
00        E
  01      A
    1100 01111001 01000011 01100011 11011100
```

We find G (`110`), but notice that the next symbol (`00`) crosses over from one byte to the next:

```
00            E
  01          A
    110       G
       0 0    E
          1111001 01000011 01100011 11011100
```

3.3 Trees

One last detail: how exactly do we compare the stream of bits against our codes, in order to figure out what symbols are there? We use a **tree!**

A binary tree works well for this because we can traverse the tree from the root to find the proper symbol. When we read a 0 from the bit stream, we go left; when we read a 1, we go right. When we finally hit a leaf node, that node is the character that we are encoding. We put that character into the output buffer, and then start over at the root again.

Our tree will be a **little bit different** than what you've seen so far. These tree nodes will never have data inside them - so, although I'm going to use our standard `TreeNode` class, I'm going to set all of the `val` fields to `None`.

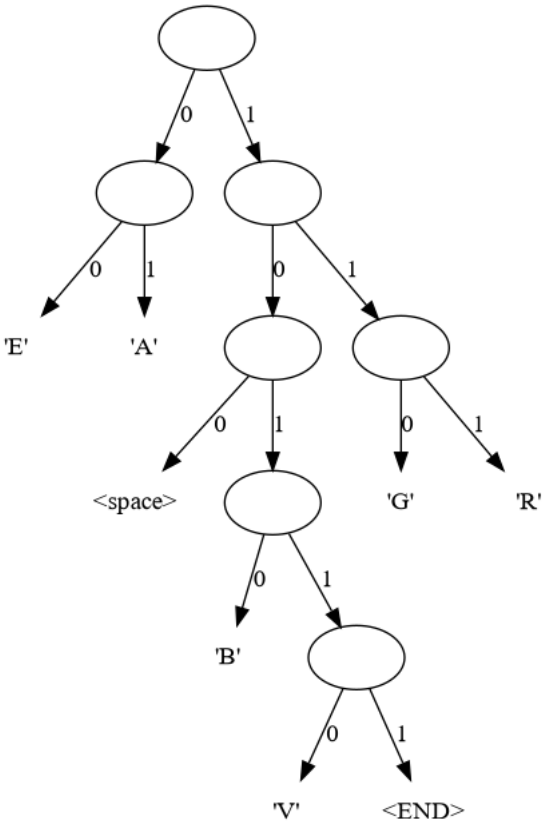
Second (and this is important!) I will use **actual strings** for our leaf nodes. This means that any time you follow a `left` or `right` pointer, you must check the type of the value with the `type()` function. If the `type()` is `str`, then you are on a leaf.

To see how this works, let's remind ourselves of the example encoding we used above:

(spec continues on the next page)

```
A         01
B         1010
E         00
G         110
R         111
V         10110
<space>   100        # will be ' ' in your actual code
<END>     10111      # will be "<END>" in your actual code
```

Here's how we would draw that tree:



REMEMBER:

- All leaves of the tree will be **strings,** not `TreeNode`
- Although I drew it `<space>` in the diagram above, the space character, in your actual code, will be represented by `' '`
- The END symbol, in your actual code, will be represented by `"<END>"`

See the next page for examples.

Here's how I would build the tree above in code:

```
root              = TreeNode(None)

root.left         = TreeNode(None)
root.right        = TreeNode(None)

root.left .left   = 'E'
root.left .right  = 'A'
root.right.left   = TreeNode(None)
root.right.right  = TreeNode(None)

root.right.left .left   = ' '
root.right.left .right  = TreeNode(None)
root.right.right.left   = 'G'
root.right.right.right  = 'R'

root.right.left .right.left   = 'B'
root.right.left .right.right  = TreeNode(None)

root.right.left .right.right.left   = 'V'
root.right.left .right.right.right  = "<END>"
```

4 Required Functions

You are going to write most of the steps for doing Huffman coding and decoding. I won't have you build the tree (because it's a little more complex), but if you're interested, the Wikipedia page shows you how to find the optimal Huffman code for a given string.

Other than that, you will do just about everything else involved in coding a string:

- Turning a string into a set of codes, using a mapping that I provide
- Taking the bits from those codes and organizing them into 8-bit chunks
- Converting 8-bit binary strings to integers, and printing out the result

as well as everything involved in decoding a string:

- Given an array of integers, converting it to a stream of bits
- Given a stream of bits and a tree representing a Huffman code, converting the bits back into letters

4.1 `str_to_codes(text, mapping)`

This function converts a string into a set of codes, using the mapping provided. The input `text` is a `str`, while `mapping` is a dictionary, which maps single characters to binary strings (all ones and zeroes). The mapping will also include a single special key `"<END>"`, which likewise maps to a binary string.

The function returns an array of strings, where each entry in the array is one of the binary strings from the mapping. The function **must** add the code for the `"<END>"` symbol, as the last element of the array.

The function **must not** re-combine the codes into 8-bit chunks; the next function will handle that.

You may assume that none of the strings overlap, as discussed in the overview above.

4.2 `codes_to_chunks(codes)`

This function groups codes into 8-bit chunks. It takes an array of binary strings (as returned by the previous function) and re-groups them into 8-bit chunks, by appending the strings together and then dividing them up as required.

If the total number of bits in the input is not a multiple of 8, then append 0's to the last string until it is exactly 8 bits long.

4.3 `print_chunks_as_decimal(chunks)`

This function takes the chunks, as returned by the previous function, and prints them out as decimal values. Print them all on a line, separated by spaces, like this:

```
123 253 10 3 72
```

HINT: You already know that the `int()` function can convert a string to an integer. If you pass a second parameter to `int()`, you can tell Python what "base" to expect. Decimal (the default) is base-10; binary is base-2.

(spec continues on the next page)

4.4 `ints_to_bits(vals)`

This function takes an array of integers, and converts it to a single gigantic binary string, turning each integer into exactly 8 bits. Do not place any spaces or any characters between the bits.

For example, if your input is

```
[10, 75, 102, 17]
```

the binary version of each number are:

```
00001010                        - 10
        01001011                - 75
                01100110        - 102
                        00010001  - 17
```

and you must return

```
"00001010010010110110011000010001"
```

HINT: You should investigate Python's built-in `bin()` function.

4.5 `bits_to_str(bits, root)`

Given a bit string (as returned by the previous function) and a Huffman code encoded as a tree (see the description above), decode the bits into a string.

Terminate the encoding when you read the `"<END>"` symbol. You may assume that it will exist somewhere in the input data.

REMINDER:
The leaves of your tree will be type `str`, not `TreeNode`.

DEBUG HINT:
You are going to need to debug this function, and so, even though the input will definitely include `"<END>"`, you're going to miss it sometimes. If you end up consuming all of the input bits without finding that symbol, print out some debug data (you can choose what) and return gracefully from the function. **Defensive coding will make your life easier in the long run.**

5 Turning in Your Solution

You must turn in your code using GradeScope.