Black Problem 2: Huffman Compression [75 points]

Copied from: https://www.cs.hmc.edu/twiki/bin/view/CS5/HuffmanBlack on 3/15/2017

Due: 11:59 PM on November 14, 2016

Starter files

First, here is a set of placeholder files to start with:

hw9pr2.zip

Next, the Millisoft back story!

You've been hired by Millisoft. Your boss, Gill Bates, has asked you to implement a Huffman compression scheme as part of their new operating system, Portholes 8. Specifically, you will implement two programs: `compress.py` and `uncompress.py`. The `compress.py` program is the "interesting" part, implementing Huffman compression.
The `uncompress.py` program is quite short, implementing decompression. You will also submit two additional documents: `design.txt` and `tests.txt`; more on these below.

When you submit, be sure to place your files:

- `compress.py`
- `uncompress.py`
- `design.txt`
- `tests.txt`
- `huff.txt` (already there)

into a folder named `hw9pr2` and zip up that folder into a file named `hw9pr2.zip`.

Then, submit your `hw9pr2.zip` file in the usual way at the submission system.

Design First

Like most software companies, Millisoft requires its software engineers to submit a software design document before embarking on the implementation. So the first thing that you must do is write a document, called `design.txt`, that lists the functions that you plan to implement for each of your `compress.py` and `uncompress.py` programs. For each function, you should give the full "signature", i.e., the name of the function and the list of its arguments, but **not** the actual Python code that implements the function. Also, for each function you should provide a docstring that clearly describes what those arguments are and what the function will return as a result. Remember that each function should perform a clearly delineated task (e.g. building the tree from the frequencies, etc.).

The Functionality of Your Programs

Your `compress.py` program should have the following functionality:

- The program starts automatically when you run the file (using the "main" trick described in class and below).
- The program asks the user for the name of a file.
- The program reads the file.
- The program counts the number of occurrences of each symbol in the file, and computes the frequencies.
- The program builds the Huffman encoding tree for this set of symbols.
- The program builds the prefix code from the tree.
- The program encodes the text in the file using the prefix code.
- The program now has a long binary string of symbols. It then packs (or "chunks") these bits, 8 at a time, into characters. Remember the built-in functions `chr` and `ord`! (`chr` accepts a number between 0 and 255 and returns the corresponding character, while `ord` accepts a single character in quotes and returns the corresponding numerical representation between 0 and 255.)
- This string of characters is now written to an output file whose name is the same as the input file except that the suffix `.HUFFMAN` is added to the end. For example, if the input file had the name `myfile` then the output file would be called `myfile.HUFFMAN`. This encoded file contains the prefix code (the code for each of the symbols—also known as "the dictionary") followed by the encoding.
- The program should report the following statistics:
    - The number of different characters in the input file.
    - The total number of bytes in the input file.

- The number of bytes required to store the code in the compressed file (this is called the "dictionary overhead").
- The number of bytes used to store the compressed text (excluding the dictionary overhead).
- The total length of the compressed file in bytes (this is just the sum of the previous two items).
- The compression achieved (length of compressed file divided by length of original file).
- The "asymptotic compression" achieved (length of compressed file except for the dictionary overhead divided by the length of the original file). This is the amount of compression that would be achieved if the file were so long that the dictionary overhead became negligible.

For one test case, you should use a text file (originally from Wikipedia) on Huffman Compression, called `huff.txt`, which is included in the starter folder (See `hw9pr2.zip`, at the top of this page.) For reference, that file is also here. It is a sample text file that the graders (and you) will use in trying your encoding (compress) and decoding (uncompress) programs.

Here is some sample input and output for the `huff.txt` file:

```
Enter name of file to be compressed: huff.txt
Original file:  huff.txt
   Distinct characters: 56
   Total bytes: 2439

Compressed file: huff.txt.HUFFMAN
   Dictionary overhead in bytes:  563
   Compressed text length in bytes:  1377
   Total length in bytes:  1940
   Actual compression ratio:  0.795407954079
   Asymptotic compression ratio:  0.564575645756
```

Notice that your program might achieve slightly better or worse compression due to the way that you save your dictionary (the prefix code). However, it should use exactly the same number of bytes to encode the data (in the example above, 1377 bytes). **Note:**If you use a special character to represent the end of the file, as discussed under "Some Details" below, your compressed text should be 1378 bytes long.

Also notice that the last byte of your compressed output won't necessarily contain 8 bits. However, since the computer operates in bytes, you'll have to

occupy a full byte even if you're storing only one bit. If your compression program claims to use only 1376 bytes, you're rounding incorrectly.

Your `uncompress.py` program will be relatively short. It should have the following functionality:

- It runs automatically using the "main trick". It simply asks the user for the name of the file to decode, and checks to make sure that the file name ends in `.HUFFMAN`. If not, it complains that the file is not of the right type and asks the user for a new file name.
- The text is decoded and placed in a file that has the same name as the input file, but with the ending `.DECODED`. For example, if the original file was `huff.txt` then the encoded file would be called `huff.txt.HUFFMAN` and the decoded file would be called `huff.txt.HUFFMAN.DECODED`. If your encoder and decoder work correctly, then `huff.txt.HUFFMAN.DECODED` is perfectly identical to `huff.txt`.

Here is some sample input and output:

```
Enter name of file to be UNcompressed (must end in .HUFFMAN):
huff.txt.HUFFMAN
Output written to file: huff.txt.HUFFMAN.DECODED
```

An Important Note

Be sure to read the hints below before you start writing your code!!!

The Implementation of Your Programs

Your implementation should follow the design in your `design.txt` document. If you need to make some changes as you implement, that is OK, but you should put comments at the very top of your program explaining the deviations from the original design and why they came about. (You should NOT change `design.txt` to make it match your final implmentation) A small number of deviations from `design.txt` is fine. A large number of deviations will suggest that the design was not as good as was intended. This is very much like a builder making changes to an architect's plans: a few changes are expected, but a larger number of changes suggests that the architect probably wasn't as careful as he or she should have been.

Also, you must have a global `debug` variable. When this variable is set to `True`, your functions will provide you a verbose description of what they are doing (e.g. telling you that the program has just entered or exited a given

function, telling you the value of some important variable in that function, or any other information that might be useful when trying to understand what's going on as the function runs). We'll run your program first with `debug = True` and then with `debug = False`.

The "main" Trick

Traditionally, the "outer" function of a program, the one that controls all of its behavior, is named `main()`. In Python, if you have a `main()` function, the following two lines of code will cause `main()` to be automatically executed when the file is run from the command line (e.g., `python foo.py`):

```
if __name__ == "__main__":
    main()
```

Note that each word above is surrounded by two underscore characters; most browsers will omit space between them so that they look like long horizontal lines.

File Input and Output (I/O)

Before you can access a file, you must first "open" it. You do this with the Python `open` function:

```
handle = open(filename, mode)
```

Here, `filename` is the name of the file you want to access; `mode` is either `"r"` (I want to read the file) or `"w"` (I want to write the file). Writing a file will create it if necessary, and wipe out the previous contents if the file already existed. The "r" or "w" can be followed by a "b" for "binary" (i.e., "rb" or "wb"). Because Python 3 distinguishes text and binary files, you **must** use the "b" suffix for your `.HUFFMAN` file. Somewhat oddly, you must also use it for the text files.

For example, to create the file `huff.txt.HUFFMAN` in your current directory, you would do:

```
huffman = open("huff.txt.HUFFMAN", "wb")
```

Open returns a *handle*, which you will use to get further access. The name of the handle is up to you. You can write a string `s` to a file with, e.g.:

```
huffman.write(s.encode('latin-1'))   # Encode is only needed for "wb" files
```

When you are done with a file you should close it:

```
huffman.close()
```

Reading a file is a bit more complicated, because Python gives you several options. Perhaps the most straightforward is `read`, which by default swallows up the entire file and returns it as a string. For example:

```
huff2 = open("huff.txt", "rb")
huffstring = huff2.read()
huff2.close()
# Convert to Python characters
huffstring = huffstring.decode('latin-1')
# You can now process the data in huff2 using normal
string operations
```

Since files can be (very) large, this isn't always the best approach. Instead, you could read one character at a time (which is very inefficient) with:

```
huffchar = huff2.read(1)
```

When there are no more characters left in the file, `read` will return the empty string.

A compromise between swallowing the whole file and being too slow is `readline`, which reads one line (up to a newline) at a time:

```
huffline = huff2.readline()
```

Again, an empty string means there's nothing more. `Readline` isn't very useful for this assignment, since compressed files aren't divided into neat lines. But you might find it useful in the future.

Finally, if you want to process a file line by line, you can also use a for loop. Here's a complete example that prints all the comments in a Python file (and, perhaps, other lines, as well…):

```
program = input("Enter a Python file name: ")
handle = open(program, "r")
for line in handle:
    if '#' in line:
        print(line, '')
handle.close()
```

(The trailing empty string in "`print(line, '')`" keeps Python from adding an extra newline at the end of each comment, since the result of `readline` already contains one.)

You'll note in the above example that we didn't use "rb" for the open mode, and that we didn't do the "decode" operation. This has to do with how Python handles strings. In normal circumstances you should use just "r" and "w" for files that contain text, and then you don't need to decode the file. However, Huffman is an unusual program and so you'll need to follow the recipe above.

Submitting a Testing Document

Like most software companies, Millisoft requires that you test each function immediately after you write it and that you submit a document summarizing your tests. Your document will be called `tests.txt`.

For example, if a function is supposed to accept a dictionary of letter frequencies as its argument and return a Huffman tree (as a tuple) as its result, you can test it by constructing a small dictionary of frequencies and giving it to this function. The test should be small enough that you can figure out by hand what the answer should be. Then check the function's result against your answer. Then, add to your `tests.txt` file something like "For the buildTree function, I used the following test arguments (list your test arguments) and verified that each one of them produced the correct tree."

Most software companies require that their software engineers submit such a test summary as evidence of reliability of the software. For your own sake, the programming will be much more efficient if you achieve metaphysical happiness with each function before moving on to the next function. The debugging time will be dramatically reduced!

Sage Advice and Partial Credit

If you design your program carefully and test equally carefully, this assignment should be fun and instructive. The total number of lines of code in this assignment is about the same as, or even slightly less than, in the previous one. However, if you can't complete the assignment for any reason, then be sure to implement as many of the parts of the algorithm as possible to get at least some fraction of the credit for your work.

For example, building the frequency table, constructing the Huffman tree, and building the prefix code from the tree demonstrates that you have built some of the core functionality. Next, generating the binary string that encodes your file using the prefix code demonstrates a bit more functionality. Packing the binary string into 8-bit chunks and writing these as characters to a file provides evidence of progress on the compression part of the code.

Some Details

- How you wish to save the Huffman dictionary (the prefix code for all of the symbols) in your compressed file is entirely up to you. Please include a detailed comment in your `compress.py` file to explain your scheme. (But see the hints below.)
- You may find it useful to use some of the base-conversion code that you wrote in an earlier assignment or to modify this code slightly. You may use those functions (paste them into whichever files you need them).
- You may find it useful to use functional-programming mechanisms in some places. In particular, using recursion will be very helpful. In addition, using `map` or a list comprehension with anonymous functions can make your code much simpler and more elegant. Please keep this in mind.
- Notice that although we discussed Huffman Coding by talking about frequencies of symbols, the frequency of a symbol is just the number of times it occurs divided by the total number of occurrences of all symbols. That is, it is a "normalized" count. In fact, we can just use the counts (actual number of occurrences of the symbols) without normalizing, and the algorithm will work just as well. This is easier! We'll keep using the term "frequency" below, but you can think of this as just the "count" instead.
- You may find it useful to use Python's dictionary type to keep your code clean and elegant. In particular, imagine that your program constructs a dictionary where each symbol in the file has an associated frequency. Then, you can use that dictionary to build up the Huffman Tree. This is a nice way of doing business! Please refer to your lecture notes for everything you need to know about the dictionary type. Remember, though, that dictionaries can only have keys that are immutable. Therefore, one can have a dictionary of numbers, strings, and tuples, but *not* of lists!
- When you write your compressed file, the bits you write probably won't wind up on an even byte boundary, so you'll have some leftovers. That can be a problem when you read the file back: what do those leftover bits decode to? The solution is to add one extra "character" to your symbol dictionary. The extra entry represents EOF (end of file) and has a frequency of 1. Be sure EOF is decoded to something that you can tell from a legitimate character, such as the string "EOF" or the Python symbol `None`.

Some Helpful Hints

There are a few Python tricks that you can use to make your life easier:

- You can't write a Python string to a file that has been opened in "wb" mode. Instead you have to give it a `bytes` object. You can convert a string named `s` to `bytes`with `s.encode('latin-1')`. So, for example, you might write `s` to a file `f` with `f.write(s.encode('latin-1'))`.
- Similarly, when you read from a file that has been opened in "rb" mode you will get a `bytes` object. You can convert a bytes object `b` back to a regular string with `b.decode('latin-1')`.
- The easiest (though not the most efficient) way to write the tree to the compressed file is to express it as a list or tuple, and then convert it to a string with `str`. When you read the tree back, if you read it into a string `s`, you can then convert it back to a list or tuple with `eval(s)`. (But when you read it back in, how will you know where it ends? One way is to arrange to know how many characters it occupies.)
- When you are building your Huffman tree, you will need to sort your list of [frequency, character] pairs. If those are stored in `freqs`, you can use `freqs.sort(key = lambda x: x[0])` to sort based on the first element in the pair. (If you don't use the `key` argument, you may get a Python runtime error.)

Submit!

Place your files:

- `compress.py`
- `uncompress.py`
- `design.txt`
- `tests.txt`
- `huff.txt` (already there)

into a folder named `hw9pr2` and zip up that folder into a file named `hw9pr2.zip`.

Then, submit your `hw9pr2.zip` file in the usual way...