CSC 440 Assignment 4: Compression (solo)

Out: Tuesday, March 19th.

Due: Tuesday, March 26th, by 11:59PM

Introduction

This is a solo assignment

You are going to implement a data compression program in Python using Huffman codes. We provide a framework; you will fill in several functions. Your choice of *internal* representations is up to you.

A stub file, `huffman.py`, is available for download on Sakai.

Functions you will write

```
    encode(msg):
```

This takes a `bytes` object (really, just a sequence of bytes over which you can iterate), `msg`, and returns a tuple `(enc, ring)` in which `enc` is the ASCII representation of the Huffman-encoded message (e.g. `"1001011"`) and `ring` is your "decoder ring" needed to decompress that message.

```
    decode(enc, ring):
```

This takes a string, `enc`, which must contain only 0s and 1s, and your representation of the "decoder ring" `ring`, and returns a string `msg` which is the decompressed message. Thus,

```
    enc, ring = encode("hello")
    print decode(enc, ring)
```

should output the string `"hello"`

```
    compress(msg):
```

This takes a `bytes` object, `msg`, and returns a tuple `(compressed, ring)` in which `compressed` is an `array` of bytes (`array.array('B')` which you may think of as a "bitstring") containing the Huffman-coded message in binary, and `ring` is again the "decoder ring" needed to decompress the message.

```
decompress(compressed, ring):
```

This takes a bitstring (`array.array('B')`) containing the Huffman-coded message in binary, and the decoder ring needed to decompress it. It returns the `bytes` object which is the decompressed message. Thus,

```
comp, ring = compress("hello")
print decompress(comp, ring)
```

should again output the string `"hello"`. The difference between `compress`/`decompress` and `code`/`decode` is that `compress` returns a non-human readable, actually *compressed* binary form of a message. That is, the result of `compress` will ultimately be a file on disk that is ***smaller*** than the original input, *as long as the input is compressible.* Recall that already-compressed formats such as `PNG`, `MP3`, and `JPEG` are not further compressible.

Essentially, you will write two versions of a compressor-decompressor loop. `encode` and `decode` are to help you; they do not save space, but represent each character in the message as a string of 0s and 1s. Once you get `encode` and `decode` to work, `compress` and `decompress` should not be too hard; most of the work will involve bit manipulations. ***Write `encode` and `decode` first!***

Actually using your compressor

In the stub huffman.py we provide you, there are already functions to handle file I/O and command-line arguments. This way, you can focus on the algorithm but still write a working compression tool. Once you have compress and decompress working,

```
python huffman.py -c test.txt test.huf
```

will compress the file test.txt and store it as test.huf, while

```
python huffman.py -d test.huf test2.txt
```

will decompress test.huf and store it as test2.txt, at which point test.txt and test2.txt should be identical.

How to submit your code

Upload `huffman.py` to Gradescope.

There is a leaderboard enabled for this assignment, based on your running time compressing and then decompressing a large, compressible binary file (a TIFF image). You can only get on the leaderboard if your solution is correct. Your grade doesn't depend on your leaderboard score; it's purely for fun once you've gotten the implementation working correctly.

For this assignment, your solution must be in Python.

Your solution should be simple enough that it works in both Python 2.7 and 3.x. We will evaluate your solutions in Python 3.

This is a SOLO assignment!

This is a solo assignment. You may not work with a partner! You may not show your code to any other classmate, or anyone who is not a member of the course staff (instructor or TAs). You also may not allow your code to be seen by anyone who is not a member of the course staff. Please see the syllabus section on Academic Integrity, or ask the instructor if you have any questions.

You MAY discuss conceptual issues, your understanding of the algorithm, and even choices of data structures and representations with your classmates. But you may not share code.

This is a solo assignment

Lateness

Submissions will not be accepted after the due date, except with the intervention of the Dean's Office in the case of serious medical, family, or other personal crises.

Grading Rubric

Your grade will be based on two components:

- Functional Correctness (50%)
  - This includes a requirement that your compression actually reduce the size of a (fairly large) file.
  - This means you have to get the bitpacking implemented, not just produce an ASCII string of 1s and 0s.
- Design and Representation (50%)
  - Documentation of *useful* invariants counts as a moderate part of this
  - But focus on the correctness of your compression; greedy algorithms are easy to prove termination.
  - Choice of proper data structures based on their cost models

For this assignment, you get a bit of a break from worrying about program inputs, since you are writing a function that conforms to an API. However, if your implementation does not respect the API (conventions) we have specified, you will receive no credit for functional correctness.

Design and Representation is our evaluation of the structure of your program, the choice of representations. How do you represent your tree? What data structure(s) do you use to build the Huffman tree?

Remember, the leaderboard is just for fun; your grade does not depend on your leaderboard rank.

Hints

Bit manipulations in Python

- `1001` is just a decimal number that happens to be ones and zeros
- `"1001"` is a string. Useful for printing (it's what `encode` and `decode` deal with) but not for compression.
- So how do we build an arbitrary binary value?

  ```
  buff = 0
  buff = (buff << 1) | 0x01
  buff = (buff << 1)
  ```

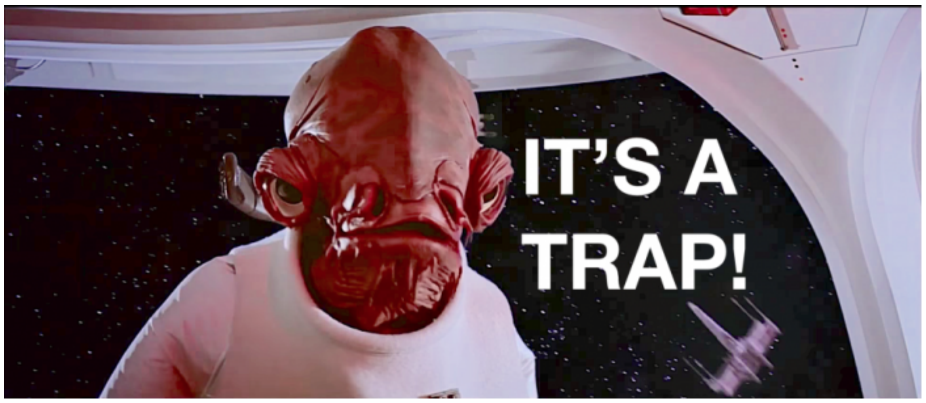
Arrays

- I recommend you use the Python `array.array` data structure to store your sequence of one-byte codewords; I've prepopulated your `compress()` and `decompress()` functions with the constructors for this data structure.
- Python 3 is fussier than Python 2 about conversions between byte-arrays and strings. My solution got more complicated when I ported it. Beware. Look at the documentation for `bytearray` and `array.array`.

Data structures

- When building a Huffman tree, we need to repeatedly get the smallest (least frequent) item from the frequency table.
- What data structure will efficiently support the operations needed?
- Ask yourself, or benchmark to check: will this dominate the runtime on a large input?
  - Remember, you can figure out where the time is spent with `cProfile`

- Your internal representation of a Huffman tree is entirely up to you. Don't overcomplicate it.
- When it comes to clever data structures and performance tuning, go where the money is. Remember, there's always the possibility that…



Zero-padding

- When you *compress* your encoded message, it will result in an integral number of bytes. But, the underlying message may not completely fill those bytes, as the compressed message, in bits, may not be a multiple of 8. So, your compressed message may be *zero-padded*. Your decompressor will need to somehow know about these extra zeros, so they aren't erroneously decoded.

Reminder

This is a solo project. No partners.