CSC 440 Assignment 4: Compression (solo)

Out: Tuesday, March 7th.

Due: Monday, March 27th, 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 and an API. You will fill in several functions. Your choice of *internal* representations is up to you.

A stub file, `huffman.py`, is available on EdStem.

You may, at your choosing, implement this algorithm in another language. See the section on Alternative Implementations below

Functions you will write

`encode(message: bytes) -> Tuple[str, Dict]:`

This takes the `message` which is a `bytes` object (really, just a sequence of bytes that has been read from a file and over which you can iterate). It returns a tuple `(encoded_message, decoder_ring)` in which the `encoded_message` is the ASCII representation of the Huffman-encoded message (i.e. a string of 1s and 0s e.g. `'10010110'`) and the `decoder_ring` is your "decoder ring" needed to decompress that message.

`decode(message: str, decoder_ring: Dict) -> bytes:`

This takes the encoded `message` i.e a string of 1s and 0s and your representation of the "decoder ring" `decoder_ring`. It returns a sequence of bytes representing the decoded message.

Thus,

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

should output the string `'hello'`. Note that the functions `encode` and `decode` are inverses of each other. One undoes what the other does.

`compress(message: bytes) -> Tuple[array, Dict]:`

This takes a `bytes` object, `message`, and returns a tuple `(compressed, decoder_ring)` in which `compressed` is an `array` of bytes containing the Huffman-coded message in binary and the `decoder_ring` needed to decompress the message.

`decompress(message: array, decoder_ring: Dict) -> bytes:`

This takes `message`, an array of bytes from a compressed file, and the `decoder_ring` needed to decompress it. It returns the `bytes` object representing the decompressed message.

Thus,

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

should again output the string `"hello"`. Note that the functions `compress` and `decompress` are inverses of each other.

The difference between `compress`/`decompress` and `encode`/`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. The `encode` and `decode` functions are to help you; they do not save space, but represent each character in the message as a string of 1s and 0s. Once you get `encode` and `decode` to work, `compress` and `decompress` should not be too hard.

Write `encode` and `decode` first!

Using your compressor and decompressor

In the `huffman.py` we provide you, we have already handled file-io and command-line arguments. This way, you can focus on the algorithm to create 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.

You can check for identity between two files with the `diff` command:

```
$ diff test.txt test2.txt
```

If this produces no output on the terminal, then the two files are identical.

How to submit your code

Upload `huffman.py` to Gradescope.

Leaderboard (for fun)

There is a leaderboard enabled for this assignment, based on your runtime performance for 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 is purely for fun once you have a correct implementation.

Alternative Implementations

For this assignment, if you use the python framework provided, your solution must be in Python 3 (not 2.x). However, for this assignment only, you may choose to use another language. If you choose another language, you do not need to follow the specific function layout specified above, but you must support **exactly the same** command-line interface for compression, decompression, encoding, and decoding, specified above.

We have infrastructure on the autograder to support submissions in Rust, or in virtually any other language.

Rust

If you upload a submission in Rust, you must upload a zip file with a specific directory structure. This will be familiar to those of you who took the Rust version of CSC 411. Using `cargo new --bin huffman` will set things up properly:

```
huffman/
    Cargo.toml
    src/
        main.rs
        ... (any other .rs files or subdirectories)
    Readme.md (optional)
```

Now, if you have a large `target` directory in which you've been compiling your program, it will choke Gradescope. I suggest zipping your project with:

```
zip -r huffman.zip huffman -x "arith/Cargo.lock" "arith/target/**" "arith/.git/**"
```

Other languages (e.g. C, C++)

Currently, Gradescope only has the Clang/Clang++ and gcc/g++ compilers installed. And, I have not tested this approach (but I'm happy to fix issues that crop up). Upon request, I could have it install another compiler such as javac (Java) or ghc (Haskell). It should also be possible to use non-statically-compiled languages such as Lua or Ruby (again, installed upon request). The key is to use `Make`.

You must have a `Makefile` at the root of your `huffman` directory, and this `Makefile` must produce an executable file called `huffman` at the root of the `huffman` directory. If you use C, for instance, you would have gcc or clang emit a binary called `huffman` (e.g. `gcc huffman.c -o huffman`).

If you use a non-compiled language such as Ruby, your Makefile must still result in an executable file called `huffman` at the root of the `huffman` directory. So for instance, your Makefile might simply rename a `huffman.rb` file to `huffman`, `chmod 755 huffman`, and ensure that it has a `#!` line like `#!/usr/bin/env ruby`

The directory structure is mostly up to you, but for instance, consider:

```
huffman/
    Makefile      (required)
    src/
        huffman.c
```

and after compilation, this would look like:

```
huffman/
    Makefile
    huffman
    src/
        huffman.c
```

Again, zip this up similarly to the example for Rust above, but you don't need those specific `-x` flags.

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.

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 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 from a string of 1s and 0s?

  byte_str = '1010' byte = f'{byte_str:08b}'

Arrays

Using the `array` library in Python, you can import the `array` data structure:

```
from array import array
```

You can create an empty array of bytes using this command:

```
byte_array = array('B')
```

The `'B'` tells Python to expect bytes to be put into this array. We recommend that you read about other formats in the documentation for `array.array` and use the one that you think will work best.

You can then append to the array of bytes:

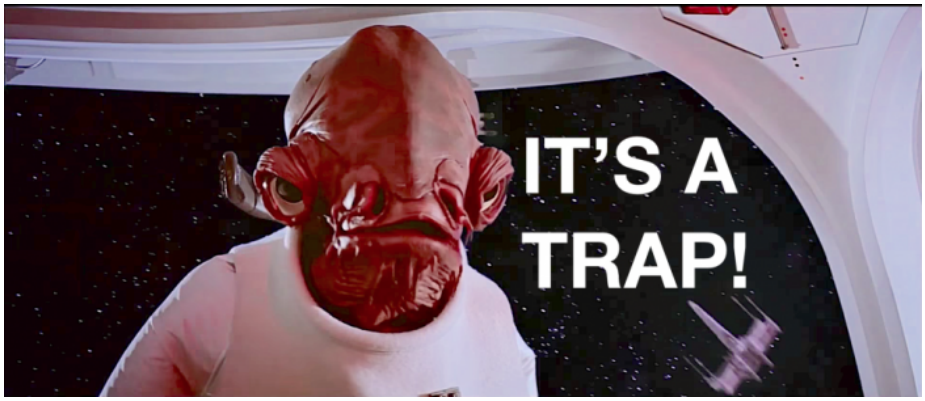
```
byte_array.append(item)
```

The item being appended but be an integer that can fit in one byte:

```
byte_array.append(42)   # valid
byte_array.append(-19)  # not valid
byte_array.append(440)  # not valid
```

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 integer number of bytes. However, the underlying message may not completely fill those bytes i.e. when compressed into a sequence of bits, the number of bits may not be a multiple of 8. So, your compressed message need some *zero-padding*. 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.