# CSC 440 Assignment 4: Compression

Out: Tuesday, March 6th

Due: Tuesday, March 20th, by 3:30PM

## Introduction

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

`code(msg):`

This takes a string, `msg`, and returns a tuple `(str, tree)` in which `str` is the ASCII representation of the Huffman-encoded message (e.g. `"1001011"`) and `tree` is your representation of the Huffman tree needed to decompress that message.

`decode(str, tree):`

This takes a string, `str`, which must contain only 0s and 1s, and your representation of the Huffman tree `tree`, and returns a string `msg` which is the decompressed message. Thus,

```
msg, tree = code("hello")
print decode(msg, tree)
```

should output the string `"hello"`

`compress(msg, tree):`

This takes a string, `msg`, and returns a tuple `(compressed, tree)` in which `compressed` is a *bitstring* (basically, just a number) containing the Huffman-coded message in binary, and `tree` is again the Huffman tree needed to decompress the message.

`decompress(str, tree):`

This takes a bitstring containing the Huffman-coded message in binary, and the Huffman tree needed to decompress it. It returns the ASCII string which is the decompressed message. Thus,

```
msg, tree = compress("hello")
print decompress(msg, tree)
```

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.

Essentially, you will write two versions of a compressor-decompressor loop. `code` 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 `code` and `decode` to work, `compress` and `decompress` should not be too hard; most of the work will involve bit manipulations.

### 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.

## 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 code 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.

### 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?

### How to submit your code

Upload `huffman.py` to Gradescope.

## 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 2.7.

## Pair Programming

You will work with a partner for this entire assignment. **If you have worked with the same partner for the last 3 assignments, it's time to find a new partner!** Please refer to the pair programming policy in the syllabus.

## 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?