Programming Assignment – Huffman Encoding and bzip

Complete this programming assignment solo or (preferably) with a partner. If you worked with the same partner in the previous programming assignment, please find a new partner to work with. Make use of the online sign-up sheet.

Introduction

You are going to implement a data compression program in Python using Huffman codes and the Burrows-Wheeler transform, which, together, form the basis of the bzip2 compression algorithm. Bzip2 consistently compresses at better ratios than ubiquitous compression algorithms like gzip and PKZIP. The basic idea behind the compression algorithm is:

1. Transform an input text file into a text file where the same letter appears more frequently together. We accomplish this with the **Burrows-Wheeler transform** (BWT).
2. Convert the BWT into a text file where small integers appear more frequently than large ones, or, **move-to-front encoding**.
3. Given a BWT and move-to-front encoded text, compress it by encoding common letters with short codewords and rare letters with long codewords, or, **Huffman encoding**.

Your task will be to implement an efficient **inverse BWT** and **Huffman encoding and decoding**. We provide a framework; you will fill in several functions. Your choice of *internal* representations is up to you. A stub file, `bwt_huffman.py`, is available for download on HuskyCT.

Burrows-Wheeler Transform

We implemented the BWT and the inverse BWT in the BWT worksheet (in the homeworks section of HuskyCT) but these are inefficient. I have provided an efficient BWT function `bwt(msg)` but you will have to implement an efficient `ibwt(msg)` function for the inverse Burrows-Wheeler transform. The key to the efficient iBWT is the First-Last property of the BWT. This property states that the characters in the Burrows Wheeler transformed text are in the same relative order as the sorted text. So we can sort the text and then use the BWT indices to find the preceeding characters in the sorted array (because they must be in the same relative order as the BWT). See the reading listed on the BWT worksheet in the homeworks section of HuskyCT and the notes on BWT.

Move-to-front encoding.

Move-to-front encoding exploits local correlations to represent the data as a series of small integers. I will describe move-to-front with respect to ASCII characters, but the implementation in the code uses their binary representation. First, initialize a list of the ASCII alphabet where the i^{th} character appears in the i^{th} position of the list. We iterate through characters of the file and output the index in the list where the character appears. We then move the character to the front of the list. This has the effect of encoding frequent and spacially close characters with small integers.

For Move-to-front decoding, we again initialize the ASCII alphabet list. We iterate through the encoded string (which correspond to positions in the ASCII list), print out the character corresponding to the read index, and then move that character to the front of the list. Move-to-front encoding and decoding are implemented for you in `mtf` and `imtf`.

Huffman encoding.

You will write the following functions.

```
encode(msg):
```

This takes a byte sequence for which you can iterate, `msg`, and returns a tuple `(enc,decoderRing)` in which `enc` is the string representation of the Huffman-encoded message (e.g. `"1001011"`) and `decoderRing` is your "decoder ring" needed to decompress that message.

```
decode(cmsg, decoderRing):
```

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

```
cmsg, ring = encode(msg)
print(decode(cmsg, ring))
```

should output the bytearray representation of the string `msg`

```
compress(msg, useBWT):
```

This takes a string or byte sequence for which you can iterate, `msg`, and returns a tuple `(compressed, decoderRing)` in which `compressed` is a *byte array* containing the Huffman-coded message in binary, and `decoderRing` is again the "decoder ring" needed to decompress the message. `useBWT` is a binary flag which indicates whether or not you should run the BWT (this is done for you).

```
    decompress(compressed, decoderRing):
```

This takes a Huffman-coded message in binary, and the decoder ring needed to decompress it. It returns a bytearray which is the decompressed message. Thus,

```
    comp, decoderRing = compress(msg)
    print(decompress(comp, decoderRing))
```

should again output the bytearray representation of the `msg` 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 three 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. The usage of the program is:

```
usage: bwt_huffman.py [-h] (-c | -d | -v | -w) -i INPUT -o OUTPUT [-b]

The program compresses binary and
plain text files using the Burrows-Wheeler transform, move-to-front coding,
and Huffman coding.

optional arguments:
  -h, --help            show this help message and exit
  -c                    Compresses a stream of bytes (e.g. file) into a bytes.
  -d                    Decompresses a compressed file back into the original
                        input
  -v                    Encodes a stream of bytes (e.g. file) into a binary
                        string using Huffman encoding.
  -w                    Decodes a Huffman encoded binary string into bytes.
  -i INPUT, --input INPUT
                        Input file path
  -o OUTPUT, --output OUTPUT
                        Output file path
  -b, --binary          Use this option if the file is binary and therefore do
                        not want to use the BWT.
```

Actually using your compressor

In the stub bwt_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 bwt_huffman.py -c -b -i image.png -o test.huf
```

will compress a binary file image.png and store it as test.huf, while

```
python bwt_huffman.py -d -b -i test.huf -o uncompressed_image.png
```

will decompress test.huf and store it as uncompressed_image.png, at which point image.png and uncompressed_image.png should be identical.

For textual data,

```
python bwt_huffman.py -c -i text.txt -o test.huf
```

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

```
python bwt_huffman.py -d -i test.huf -o uncompressed_text.txt
```

will decompress test.huf and store it as uncompressed_text.txt, at which point text.txt and uncompressed_text.txt should be identical. You can also use the -v and -w flags to text your Huffman encoding and decoding.

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 `bytearray` data structure to store your sequence of 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. Think about what data structure will efficiently support the operations needed.

How to submit your code

Upload `bwt_huffman.py` and your benchmarking PDF separately to Gradescope.

Implementation

We will evaluate your solutions in Python 3.8

Analysis

Benchmark your implementation on several inputs and compare your implementation with several other compression software (e.g. zip, rar, bzip, gzip). The compression ratio is the ratio between the uncompressed size and compressed size of a file. What are the compression ratios for each file? How does enabling or disabling the move-to-front and BWT affect the compression ratio, compression time, and decompression time? Why does (or doesn't) the move-to-front and BWT preprocessing steps affect the compression ratio? Provide a theoretical argument or empirical results.

Grading Rubric

Your grade will be based on three components:

- Functional correctness (50%). This includes a requirement that your compression actually reduces the size of a (fairly large) file. It also means you have to get the bitpacking and iBWT implemented, not just produce an ASCII string of 1s and 0s. Your implementation will be tested on binary data (with the -b flag) and text data.
- Benchmarking comparison and code design (30%). This includes an evaluation of your benchmarking. Do you only compare against a single other compression algorithm, or several? Do you evaluate the compression ratio and running time? We will not be providing you with example data, you should identify input to test your algorithm.
- Analysis of the algorithm (10%). Our evaluation on your answers to the *Analysis* section questions.
- Design and representation (10%). Our evaluation of how well your code is documented and structured. Do you use appropriate data structures (e.g. to get the lowest frequency symbol in the Huffman encoding algorithm)?