Huffman Coding

For this assignment, you will be writing programs that compress and decompress files using Huffman codes. The compressor will be a command-line utility that encodes any file into a compressed version with a `.huf` extension. The decompressor will be a web server that will let you directly browse compressed files, decoding them on-the-fly as they are being sent to your web browser.

You will implement the following functionality:

1. Writing Huffman trees into files and reading them back.
2. Compressing data into Huffman encoded form using a provided tree.
3. Decompressing data in Huffman encoded form to recover the original.

More details on each task are provided below. You will use the `huffman.py` module developed in class to create Huffman trees and use them for encoding and decoding; this code is included with the assignment, along with the required `minheap.py`. To perform bitwise input and output, the `bitio.py` module introduced in class is also provided.

You will have to implement the functions described below in the `util.py` module. You should submit all the code required for a working project, including the provided files and your additions to `util.py`. **The assignment is due Monday, March 14 at 11:55 pm.** You can submit multiple times; only the last submission will be graded. You should submit early to avoid being cutoff because the server is overloaded.

Part I: Decompression

For this part of the assignment you will write code that reads a description of a Huffman tree from an input stream, constructs the tree, and uses it to decode the rest of the input stream. The code we provide will set up a simple web server that uses your decompression routines to serve compressed files to a web browser.

The `util.read_tree` Function

You will first implement the `read_tree` function in the `util.py` module. It will have the following specification.

```
def read_tree (bitreader):
    '''Read a description of a Huffman tree from the given bit reader,
    and construct and return the tree. When this function returns, the
    bit reader should be ready to read the next bit immediately
    following the tree description.

    Huffman trees are stored in the following format:
      * TreeLeafEndMessage is represented by the two bits 00.
      * TreeLeaf is represented by the two bits 01, followed by 8 bits
          for the symbol at that leaf.
      * TreeBranch is represented by the single bit 1, followed by a
          description of the left subtree and then the right subtree.

    Args:
      bitreader: An instance of bitio.BitReader to read the tree from.

    Returns:
      A Huffman tree constructed according to the given description.
    '''
    pass
```

The `decompress` Function

You will use the above `read_tree` function to implement the following `decompress` function in the `util` module.

```
def decompress (compressed, uncompressed):
    '''First, read a Huffman tree from the 'compressed' stream using your
    read_tree function. Then use that tree to decode the rest of the
    stream and write the resulting symbols to the 'uncompressed'
    stream.

    Args:
      compressed: A file stream from which compressed input is read.
      uncompressed: A writable file stream to which the uncompressed
          output is written.

    '''
    pass
```

You will have to construct a `bitio.BitReader` object wrapping the `compressed` stream to be able to read the input one bit at a time. As soon as you decode the end-of-message symbol, you should stop reading.

Running the Web Server

Once you have implemented the `decompress` function, you will be able to run the `webserver.py` script to serve compressed files. To try this out, change to the `wwwroot/` directory included with the assignment and run

```
python3 ../webserver.py
```

Then open the url `http://localhost:8000` in your web browser. If all goes well, you should see a web page including an image. Compressed versions of the web page and the image are stored as `index.html.huf` and `huffman.bmp.huf` in the `wwwroot/` directory. The web server is using your `decompress` function to decompress these files and serve them to your web browser.

Part II: Compression

The code we provide will open an input file, construct a frequency table for the bytes it contains, and generate a Huffman tree for that frequency table. You will write code that writes this tree to the output file using the format described below, followed by the actual coded input.

The `util.write_tree` Function

You will first implement the `write_tree` function in the `util.py` module. It will have the following specification.

```
def write_tree (tree, bitwriter):
    '''Write the specified Huffman tree to the given bit writer.  The
    tree is written in the format described above for the read_tree
    function.

    DO NOT flush the bit writer after writing the tree.

    Args:
      tree: A Huffman tree.
      bitwriter: An instance of bitio.BitWriter to write the tree to.
    '''
    pass
```

As noted in the specification, **do not** flush the bit writer when you've written the tree; the coded data will be written out directly following the tree with no extraneous zero bits in between.

The `util.compress` Function

You will use the above `write_tree` function to implement the following `compress` function in the `util.py` module.

```
def compress (tree, uncompressed, compressed):
    '''First write the given tree to the stream 'compressed' using the
    write_tree function. Then use the same tree to encode the data
    from the input stream 'uncompressed' and write it to 'compressed'.
    Finally, write the code for the end-of-message sequence, and
    if there are any partially-written bytes remaining, pad them with
    0s and write a complete byte.

    Args:
      tree: A Huffman tree.
      uncompressed: A file stream from which you can read the input.
      compressed: A file stream that will receive the tree description
          and the coded input data.
    '''
    pass
```

You will have to construct a `bitio.BitWriter` wrapping the output stream `compressed`. You will also find the `huffman.make_encoding_table` function useful.

Running the Compressor

Once you have implemented the `util.compress` function, you will be able to run the `compress.py` script to compress files. For example, to add a new file `somefile.pdf` to be served by the web server, copy it to the `wwwroot/` directory, change to that directory, and run

```
python3 ../compress.py somefile.pdf
```

This will generate `somefile.pdf.huf` and you will be able to access the decompressed version at the URL `http://localhost:8000/somefile.pdf`. You should download the decompressed file and compare it to the original using the `cmp` command, to make sure there are no differences.

Submitting

Please submit all the Python code needed to run the assignment, including the files we provide. Submit all these files as a compressed zip archive.