CMPUT 274 — Tangible Computing **Version 1.2-21**
Assignment 2 — Huffman Coding

Introduction

This assignment will require you to know something about:

1. The basics of binary representation and number systems.
2. How characters are represented on a computer in binary.
3. Simple binary trees and how to traverse a tree from root to leaf.
4. The Huffman compression algorithm.
5. Classes in Python.
6. The BitReader and BitWriter classes implemented in `bitio.py`
7. The Pickle module in Python. (Do NOT use JSON files.)

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.

Important: You are only responsible for implementing code in `util.py`. The steps below are a broad overview of what will be done for the project; however, all of the code in `huffman.py` and the other files are provided.

Encoding a message using Huffman codes requires the following steps:

1. Read through the message and construct a frequency table counting how many times each symbol (i.e. byte in our case) occurs in the input (`huffman.make_freq_table` does this).
2. Construct a Huffman tree (called `tree`) using the frequency table (`huffman.make_tree` does this).
3. Write `tree` to the output file, so that the recipient of the message knows how to decode it (you will write the code to do this in `util.write_tree`).
4. Read each byte from the uncompressed input file, encode it using `tree`, and write the code sequence of bits to the compressed output file. The function `huffman.make_encoding_table` takes `tree` and produces a dictionary mapping bytes to bit sequences; constructing this dictionary once before you start coding the message will make your task much easier,

Decoding a message produced in this way requires the following steps:

5. Read the description of `tree` from the compressed file, thus reconstructing the original Huffman tree that was used to encode the message (you will write the code to do this in `util.read_tree` using the `pickle` module in Python to deserialize the tree.).
6. Repeatedly read coded bits from the file, decode them using `tree` (the `util.decode_byte` function does this), and write the decoded byte to the uncompressed output.

You will implement the following functionality:

- The `util.write_tree` function to write Huffman trees into files using the `pickle` module in Python to serialize the tree.
- The `util.compress` function to accomplish steps 3 and 4 above (using `util.write_tree` for step 3).
- The `util.decode_byte` function that will return a single byte representing the next character of the original text that is encoded in the BitReader corresponding to the compressed text.
- The `util.read_tree` function to read Huffman trees from files (step 5 above)
- The `util.decompress` function to accomplish steps 5 and 6 above (using `util.read_tree` for step 5).

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. To perform bitwise input and output, the `bitio.py` module introduced in class is also provided.

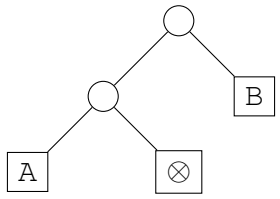
You will have to implement the functions described above in the `util.py` module. You should submit only this module.

Representing Huffman Trees in Python

We represent trees as instances of the following classes in the `huffman.py` module:

- `TreeLeaf`: Leaves representing symbols (i.e. bytes, or numbers in the 0-255 range) as well as the special "end-of-message" symbol, which occurs only once in each message, marking its end.
- `TreeBranch`: A branch, which in turn consists of two subtrees.

For example, consider the following Huffman tree, which has leaves for the symbols `A`, `B`, and the special end-of-message symbol $\otimes$.



In Python, we would represent the above tree as

```
TreeBranch(
    TreeBranch(
        TreeLeaf(ord('A')),
        TreeLeaf(None)
    ),
    TreeLeaf(ord('B'))
)
```

The symbol associated with each leaf is stored as a byte, which is an integer in the range 0–255. This is why the symbol `A` is represented by its ASCII value 65, given by `ord('A')`.

Representing Huffman Trees as Bit Sequences

As we saw above, we want to be able to read and write Huffman trees into files. Note that this is very different from *encoding* a message using a Huffman tree; here we are concerned with representing the tree itself. In this assignment, you will use the `pickle` module in Python to convert the tree from a Python form to a serialized form that can be stored in a file.

Effectively, this means that you will convert the Tree class as it is stored in Python into a form that can be compressed and stored into a file. Your `read_tree` and `write_tree` functions will use `pickle` to convert between the Python representation of Huffman trees and the serialized representation. More precisely, the `read_tree` function should take a file object stream and use `pickle` to reconstruct a full tree according to the above format, and then return the tree. The `write_tree` function should write a tree to the provided file object stream also using `pickle`.

Task 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(tree_stream):
    '''Read a description of a Huffman tree from the given compressed
    tree stream, and use the pickle module to construct the tree object.
    Then, return the root node of the tree itself.

    Args:
      tree_stream: The compressed stream to read the tree from.

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

The `util.decode_byte` Function

You will next implement the `decode_tree` function in the `util.py` module. It will have the following specification.

```
def decode_byte(tree, bitreader):
    """
    Reads bits from the bit reader and traverses the tree from
    the root to a leaf. Once a leaf is reached, bits are no longer read
    and the value of that leaf is returned.

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

    Returns:
      Next byte of the compressed bit stream.
    """
```

The `util.decompress` Function

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

```
def decompress(compressed, uncompressed):
    '''First, read a Huffman tree from the 'tree_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.
    '''
```

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.

Task 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, tree_stream):
    '''Write the specified Huffman tree to the given tree_stream
    using pickle.

    Args:
      tree: A Huffman tree.
      tree_stream: The binary file to write the tree to.
    '''
```

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 'tree_stream' using the
    write_tree function. Then use the same tree to encode the data
    from the input stream 'uncompressed' and write it to 'compressed'.
    If there are any partially-written bytes remaining at the end,
    write 0 bits to form a complete byte.

    Flush the bitwriter after writing the entire compressed file.

    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.
      tree_stream: A file stream where the tree data should be dumped.
    '''
```

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

Testing Your Code

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.

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.

Your archive when extracted should contain only these two files, which should not reside within a directory!

Submission Guidelines:

Submit all of the required files (and no other files) as **one** properly formed compressed archive called either `huffman.tar.gz`, or `huffman.tgz`, or `huffman.zip` (for full marks, please do **not** use `.rar`):

- when your archive is extracted, it should result in exactly *one directory* called `huffman` (use this exact name) with the following files in that directory:
- `util.py` (use this exact name) **(NOT `huffman.py`)** modified and containing your Python code and *docstrings-based documentation*.
- your `README` (use this exact name) conforms with the Code Submission Guidelines.
- No other files should be submitted.

Note that your files and functions must be named **exactly** as specified above. Do **not** have any extraneous calls to `input()`, `print()`, or other I/O functions.

A new tool has been developed by the TAs to help check and validate the format of your `tar` or `zip` file *prior* to submission. To run it, you will need to download it into the VM, and place it in the same directory as your compressed archive (e.g., `huffman.zip`). You can read detailed instructions and more explanation about this new tool in Submission Validator: Instructions (at the top of the Weekly Exercises tab), or run:

```
python3 submission_validator.py --help
```

after you have downloaded the script to see abbreviated instructions printed to the terminal. If your submission passes this validation process, and all validation instructions have been followed properly, you will not lose any marks related to the format of your submission. (Of course, marks can still be deducted for correctness, design, and style reasons, but not for submission correctness.)

When your marked assignment is returned to you, there is a 7-day window to request the reconsideration of any aspect of the mark. After the window, we will only change a mark if there is a clear mistake on our part (e.g., incorrect arithmetic, incorrect recording of the mark). At any time during the term, you can request additional feedback on your submission.