CMPUT 274 — Tangible Computing **Version 2.4-23**
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 in `util.py`:

- 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.

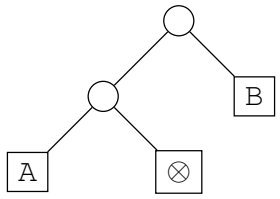
You will have to implement the functions described above in the `util.py` module. You should submit only this module.

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.

    Hint:  Use pickle.dump()
    **Requirement:**  pickle.dump(..., protocol=4)

    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. **Be sure to specify `protocol=4`**.

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.
    '''
```

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://<IP-address>:8000` in your web browser. If you are using multipass, then the `<IP-address>` is what `multipass list` shows you.

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://<IP-address>: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.

Hints:

1. ***** VERY VERY IMPORTANT *** Be careful of the differences with how Windows (e.g., as your Host OS for the VM, and as a shared directory with your VM) handles text files.**

   Some Windows tools (e.g., for git, text editors, IDEs like VS Code) and software automatically change lines to end in CR/LF, which is different from the VM. Do your testing in a non-Windows, non-shared directory with Windows since we will be doing our testing inside the VM.

2. If a file, after compression, decompression, or both, is different by just one byte (and often the last byte), the two most common sources of the bug are:

   (a) Incorrect handling of the EOF case, especially during compression. In this assignment, an EOF is encoded as part of the Huffman tree.

   (b) Failure to properly invoke method `flush()`, **only after writing the entire compressed file**, on the `class BitWriter` from `bitio.py`.

Submission Guidelines:

Using git and GitHub classroom, submit all of the required files by committing and pushing out your work to GitHub. Do NOT include any extra files not required by the specifications.

You are encouraged to commit and push your work to git while working on your solution (e.g., once per day, or even once per hour). Only the last commit and push, prior to the deadline, will be marked. Be careful to **not** do any additional pushes after the deadline.

Git is an excellent tool for backing up your work and git provides a way to revert to a previous version of your file(s), if necessary.

See below for additional hints about git and (possibly) other issues.

In your final commit, and assuming your GitHub identity is **GITHUB-ID**, be sure:

- to maintain the directory structure given to you (via git clone), namely `ASSN2-GITHUB-ID/huffman` (this exact name for the directory) with the following files in that directory:
- `util.py` (this exact name) contains all of your modified Python code and *docstrings-based documentation*.
  - We will only use your `util.py` file. All other files will be unchanged copies that we provide.
- your `README` (use this exact name) conforms with the Code Submission Guidelines.
- your `README.WSL` (use this exact name) **only if** you tested your code using **Windows Subsystem for Linux (WSL)** instead of using multipass-Ubuntu. The contents of the `README.WSL` can be blank, because the contents will be ignored.
- `Makefile` (not marked). Very important to read and understand this Makefile.
- Several test files are provided to you, but will be ignored by the marker.

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.

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.

Marking Rubric:

NOTE: The code must solve the problem algorithmically. If there is any hardcoding for the provided test cases, then zero Correctness marks will be given.

This assessment will be marked out of 100 for:

- `Correctness`: Meets all specifications, generates no extraneous output, requires no unspecified input, and provides correct output files/answers for the test inputs provided. Note: The output of the programs is a file. The Unix `cmp` program will be used to compare files.
  - We will only use your `util.py` file. All other files will be unchanged copies that we provide.
  - `90/90`: Pass all of the provided test inputs
  - `60/90`: Pass any 5 (or more) of the 9 provided test cases
  - `40/90`: Pass any 3 (or more) of the 9 provided test cases
  - `20/90`: Pass any of the provided test inputs
- `Valid Submission: 10/10`: Has proper `README` file. If applicable, has `README.WSL`. All other submission requirements are satisfied

This assignment emphasizes correctness and conformance to submission guidelines.

Although programming style and code quality (as described in the *Code Submission and Style Guidelines* document on eClass) are always important, we will not emphasize nor mark style for this assignment. Future assessments may include additional requirements for style and design. Students are always encouraged to get feedback on quality and style from a TA or instructor.

As discussed in the Course Outline, this is a Consultation Model assessment. You are allowed to discuss the exercise with fellow students.

Furthermore, going beyond basic consultation, you are allowed to help and receive help from fellow students with your code. During Office Hours or otherwise, you might see the code of fellow students and you are allowed to use that code in your own code. However, if more than 10 lines of code originate from any online resource or other person, you must acknowledge the origins of the code with a comment in the source code **and** in your README file.

These *Acknowledgements* can be considered a part of the *Notes and Assumptions* section of the README.

For example, use a comment of the form:

1. `Code from http://xyz`
2. `Code from person:  J. Smith`
3. `Code from TA: A. Singh`
4. `Code from Office Hours on September 27`
5. `Code from document/book:  Miller-Ranum textbook`
6. `Help received from person:  C. Wong`

Failure to properly acknowledge **all** sources of code will be a violation of the policies for CMPUT 274 and will result in a deduction of marks, at the discretion of the Instructor.

Explicitly disallowed is taking more than 10 lines of code that someone else has written, making minor changes to it, and submitting it, even with acknowledgement. Be aware that tools such as MOSS might be used to find code common to multiple submissions or code found on the Internet.

Other Hints and Comments::

1. After submitting your code, create a new clone of your repository, in a different directory. Then, re-test that new clone.

   This will check if all files and updates have been pushed to GitHub, and if there are any inadvertent path dependencies in the code.
2. Be sure to acknowledge all sources of code and help as a source code comment, and in your README file.
3. Some questions on Quizzes, the Midterm, and the Final Exam will be based on the programs you write for Morning Problems, Weekly Exercises, and Assignments. Therefore, you will need to have a proper understanding of why your solution code works (and does not work).
4. Be sure the output (if any) is exactly as described and shown. As with a Python source code file itself, a single character that is wrong, a missing space, an extra space, or even a missing newline, is incorrect. Testing is automated (as in industry) and both input and output must follow the specification.