**CMPUT 275 — Tangible Computing** **Winter 2018**
**Assignment 2 — Huffman Coding**

# Introduction

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.

Encode 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`).
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 in the format described below (step 3 above).
- 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, 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 above in the `util.py` module. You should submit only this module. **The assignment is due Monday, March 26 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.

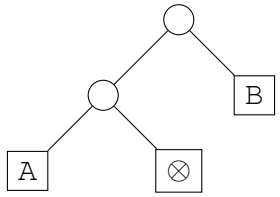
**Notes:** Yes, we realize that you can simply use the `pickle` module in Python to serialize the tree (i.e. make the `util.write_tree` and `util.read_tree` parts trivial). We are asking you to implement this functionality yourself rather than relying on existing modules.

## 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 ⊗.




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. We use the following format to represent trees:

- `TreeLeaf` is represented by the two bits `01` followed by 8 bits for the leaf's symbol, unless it is the special end-of-message symbol which is represented by `00`. For example, `TreeLeaf(ord('A'))` is represented by the bit sequence `0101000001`, i.e. `01` followed by the binary representation of 65, and `TreeLeaf(None)` is represented by `00`.

- `TreeBranch` is represented by the single bit `1` followed by representations of the left subtree and the right subtree. For example, a branch with the left subtree being a leaf for symbol `A` and the right subtree being the end-of-message leaf would be represented by the bit sequence `1010100000100`.

You should now be able to work out that the Huffman tree pictured above is represented by the following bit sequence: (the spaces are just for clarity)
`1 1 01 01000001 00 01 01000010`

You should also be able to see that there is no need to terminate the bit sequence with a special end marker: you can always tell when you've read enough bits to construct the complete tree.

Your `read_tree` and `write_tree` functions must be able to convert between the Python representation of Huffman trees and the bit representation described here. More precisely, the `read_tree` function should take an instance of `BitReader` and read enough bits from it 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 instance of `BitWriter` in the above format.

# 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 (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:
      * TreeLeaf that is None (the special "end of message" character)
          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 `util.decode_byte` Function

You will first 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 leave 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.
    '''
    pass
```

## 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 '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.

# 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, 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.

    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.
    '''
    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.

# 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.

# Submitting

Please submit the modified `util.py` file only. Do not zip it!