# CS 3430: SciComp with Py
# Assignment 12 (Last One!)
# Encoding and Decoding with Huffman Trees

Vladimir Kulyukin
Department of Computer Science
Utah State University

April 11, 2020

## 1 Learning Objectives

1. Encoding
2. Decoding
3. Huffman Trees
4. Variable-Length Codes

## Introduction

In this assignment, you'll learn how to do variable-length encoding and decoding with Huffman trees. Recall that the objective of encoding is to come up with a map that assigns each symbol (e.g., a character or a number) in a message a specific code (i.e., a bit string). One of the best known encoding algorithms is Morse code. There are two types of codes: fixed-length and variable-length. In a fixed-length code, every character is assigned the same number of bits. For example, in standard ASCII, every character is assigned exactly 7 bits, which gives us $2^7 = 128$ possible character encodings. In general, if we want to encode $n$ different characters, also known as symbols, we need $log_2(n)$ bits per symbol. Thus, if our alphabet consists of 8 characters: A, B, C, D, E, F, G, and H, we need $log_2(8) = 3$ bits per character.

## Problem 1 (0 points)

Review the materials for Lectures 23 and 24 on Huffman trees. Make sure you're comfortable with such concepts as encoding, decoding, fixed- and variable-length codes, encoding/decoding equations, symbol weights, symbol weight maps, the Huffman tree generation algorithm, and node mergers. I'll take these concepts for granted in my narrative below.

## Problem 2 (4 points)

### Huffman Tree Nodes

Our first task is to use OOP to implement a Huffman tree node class out of which we'll build Huffman trees. The constructor of the class `HuffmanTreeNode` takes a set of characters and their weights. If a node's a leaf, its symbol set will consist of only one character. I've included in the assignment my source in `HuffmanTreeNode.py`. I'd like to quickly draw your attention to two magic methods: `__str__` and `__eq__`.

```
def __str__(self):
    return 'HTN(' + str(self.__symbols) + ', ' + str(self.__weight) + ')'

def __eq__(self, htn):
    return self.getWeight() == htn.getWeight() and \
           len(self.getSymbols().difference(htn.getSymbols())) == 0 and \
           len(htn.getSymbols().difference(self.getSymbols())) == 0
```

The method `__str__()` is called every time you call `str(x)` where `x` is a HuffmanTreeNode object. Here's an example.

```
>>> from HuffmanTreeNode import HuffmanTreeNode
>>> htn1 = HuffmanTreeNode(symbols=set(['A']), weight=8)
>>> str(htn1)
"HTN({'A'}, 8)"
>>> htn2 = HuffmanTreeNode(symbols=set([3, 4, 1]), weight=10)
>>> str(htn2)
'HTN({1, 3, 4}, 10)'
>>> print('My nodes are {} and {}'.format(htn1, htn2))
My nodes are HTN({'A'}, 8) and HTN({1, 3, 4}, 10)
```

The method `__eq__()` is called every time you call `x == y` where both `x` and `y` are HuffmanTreeNode objects. Take a look at `test_ht_ut00()` in `HuffmanTreeUTs.py`. In the following code segment both assertions pass because `htn01` and `htn02` are equal.

```
htn01 = HuffmanTreeNode(symbols=set(['A', 'B', 'C']), weight=10)
htn02 = HuffmanTreeNode(symbols=set(['B', 'C', 'A']), weight=10)
assert htn01 == htn02
assert htn02 == htn01
```

In this code segment both assertions pass because `htn03` and `htn03` are not equal.

```
htn03 = HuffmanTreeNode(symbols=set([1, 2, 3]), weight=5)
htn04 = HuffmanTreeNode(symbols=set([2, 3, 1, 5, 4]), weight=5)
assert htn03 != htn02
assert htn02 != htn03
```

## From Huffman Nodes to Huffman Trees

Our implementation of a Huffman tree will have only one member variable – the root. Once we have the root, we can get to all nodes by using `HuffmanTreeNode.getLeftChild()` and `HuffmanTreeNode.getRightChild()`. The code segment below is in `HuffmanTree.py`.

```
from HuffmanTreeNode import HuffmanTreeNode

class HuffmanTree(object):
    def __init__(self, root=None):
        self.__root = root

    def getRoot(self):
        return self.__root
```

When constructing a Huffman tree from a list of Huffman node objects, we find the two nodes with the lowest weights and merge them to produce the new node that has these nodes as its left and right children and whose weight is the sum of the weights of the two children (see slides 6, 7 in `cs3430_s20_lec24_huffman_part02.pdf`). We remove the two child nodes from the node list and add the new node back to the list. The procedure goes on until there is only one node left in the list. This node becomes the root of the Huffman tree and the procedure terminates by returning the root node. You may want to make a brief pause here and review slide 5 in `cs3430_s20_lec24_huffman_part02.pdf` and the subsequent extended example of a Huffman tree construction.

Here's an implementation of the method for merging two nodes in `HuffmanTree.py`. It's a static method, i.e., it doesn't require an `HuffmanTree` object and can be called as `HuffmanTree.mergeTwoNodes(n1, n2)`.

```
    @staticmethod
    def mergeTwoNodes(htn1, htn2):
        symbols = set(htn1.getSymbols())
        for i in htn2.getSymbols():
            symbols.add(i)
        n = HuffmanTreeNode(symbols=symbols, weight=htn1.getWeight() + htn2.getWeight())
        n.setLeftChild(htn1)
        n.setRightChild(htn2)
        return n
```

Implement another static method, `HuffmanTree.fromListOfHuffmanTreeNodes()` that takes a list of `HuffmanTreeNode` objects and returns a `HuffmanTree` object constructed according to the Huffman tree construction algorithm on slide 5 in `cs3430_s20_lec24_huffman_part02.pdf`.

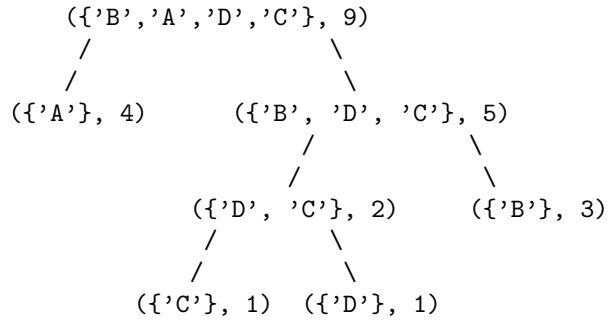
Let's run `test_ht_ut01()` in `HuffmanTreeUTs.py`.

```
char_freqs01 = [('A', 4), ('B', 3), ('C', 1), ('D', 1)]
def test_ht_ut01(self, tn=1):
    hnodes = [HuffmanTreeNode(symbols=set([kv[0]]), weight=kv[1])
              for kv in HuffmanTreeUTs.char_freqs01]
    ht = HuffmanTree.fromListOfHuffmanTreeNodes(hnodes)
    assert ht is not None
    HuffmanTree.displayHuffmanTree(ht)
```

Here's the Huffman tree displayed with `HuffmanTree.displayHuffmanTree()` that my implementation constructs.

```
{'B', 'A', 'D', 'C'}:9
        {'A'}:4
        {'B', 'D', 'C'}:5
               {'D', 'C'}:2
                        {'C'}:1
                        {'D'}:1
               {'B'}:3
```

This Huffman tree can drawn as follows.




Recall that there can be several equivalent Huffman trees constructed from the same symbol weight map. The tree your algorithm will construct will depend on how you find the two nodes with the lowest weights and arbitrarily break the ties. However, the encoding lengths in your tree must be the same as the encoding lengths in the above tree. In other words, the encoding of `'A'` is 1 bit long; the encodings of `'C'` and `'D'` are 3 bits long; and the encoding of `'B'` is 2 bits long. I've added a detailed debugging trace my implementation produces in a comment right before `test_ht_ut01(self, tn=1)` in `HuffmanTreeUTs.py`.

Implement the methods `encodeSymbol()` and `encodeText()` to encode individual characters and texts, respectively. Actually, `encodeText()` simply concatenates into one string the outputs of `encodeSymbol()` for individual characters. So, once you implement `encodeSymbol()`, `encodeText()` is a simple `for`-loop.

Both methods should return strings of characters `'0'` and `'1'`. We'll talk about bit fiddling later. The method `test_ht_ut03(self, tn=3)` in `HuffmanTreeUTs.py` constructs the Huffman trees on slides on 18 and 19 in `cs343_s20_lec24_huffman_part02.pdf` and prints out the encoding of each character in the alphabet. Here's part of the output generated by this unit test.

```
SCIP Huffman Tree Encodings:
encode(A) = 0
encode(B) = 100
encode(C) = 1010
encode(D) = 1011
encode(E) = 1100
encode(F) = 1101
encode(G) = 1110
encode(H) = 1111
Alternative Huffman Tree Encodings:
encode(A) = 0
encode(B) = 111
encode(C) = 1000
encode(D) = 1001
encode(E) = 1010
encode(F) = 1011
encode(G) = 1100
encode(H) = 1101
```

Implement the method `decode()` in `HuffmanTree.py` that takes a string of the `'0'` and `'1'` characters and decodes it with a given Huffman tree. The code segment below is from `test_ht_ut04(self, tn=4)` and tests the encode/decode equations on slide 23 in `cs3430_s20_lec23_huffman.pdf` with the Huffman trees on slides on 18 and 19 in `cs343_s20_lec24_huffman_part02.pdf`.

```
txt0 = 'AABCCDEFG'
txt1 = 'AABBBBHHHFFGGGDDDDEEEECCCCCCCCDDDDDHHHHHAAAAAA'
enc0 = scip_ht.encodeText(txt0)
print('txt0 = {}'.format(txt0))
print('enc0 = {}'.format(enc0))
enc1 = scip_ht.encodeText(txt1)
print('txt1 = {}'.format(txt1))
print('enc1 = {}'.format(enc0))
dec0 = scip_ht.decode(enc0)
print('dec0 = {}'.format(dec0))
dec1 = scip_ht.decode(enc1)
print('dec1 = {}'.format(dec1))
assert dec0 == txt0
assert dec1 == txt1
```

Here's the output.

```
txt0 = AABCCDEFG
enc0 = 00100101010101011110011011110
txt1 = AABBBBHHHFFGGGDDDDEEEECCCCCCCCDDDDDHHHHHAAAAAA
enc1 = 00100101010101011110011011110
dec0 = AABCCDEFG
dec1 = AABBBBHHHFFGGGDDDDEEEECCCCCCCCDDDDDHHHHHAAAAAA
```

## Bit Fiddling

The above implementation of a Huffman tree is conceptually adequate in that it does produce 0-1 encodings. However, it's not adequate from the point of view of data compression in that it doesn't reduce the size of the encoded text. It returns *strings* of 0-1 characters instead of strings bits. To address this problem, we need to implement a Huffman tree class that generates raw bytes instead of characters. Bit fiddling is not within the scope of this class. I hope that you either have had some exposure to it in a computer architecture or an OS class or, if not, will get some exposure to it some day. Since bit fiddling, in and of itself, doesn't have much to do with scientific computing, I won't dive into it here.

To experiment with bit fiddling, I've given you my implementation of a binary Huffman tree class in `BinHuffmanTree.py`. The class uses the methods in `HuffmanTree.py` but encodes texts as bit strings and decodes bit strings back into texts. The constructor of `BinHuffmanTree` takes the root node of a `HuffmanTree` object. The method `encodeText()` of `BinHuffmanTree` takes a text message and returns the encoding of this message as a byte array, the number of bytes in the encoding, and the number of padded bits, i.e., the bits we have to add to the end of the encoding to make its length to be divisble by 8.

The method `encodeTextToFile()` of `BinHuffmanTree` takes a text message and a file path and produces two files. The first file has the extension `.bin` and contains the bytes of the encoded message (i.e., the encoding). The second file has the extension `.txt` and contains two lines, each of which has one integer: the first integer denotes the number of bytes in the `.bin` file and the second integer denotes the number of the padded bits added to generate `.bin` file.

The method `decodeTextFromFile()` of `BinHuffmanTree` takes a file path, adds the `.bin` and .txt extensions to the file path to open the needed files and then decodes the `.bin` file to produce the original text.

Let's run `test_ht_ut06(self, tn=6)` and see the results. This unit test encodes and decodes two texts defined below.

```
txt0 = 'AABCCDEFG'
txt1 = 'AABBBBHHHFFGGGDDDDEEEECCCCCCCCDDDDDHHHHHAAAAAA'
```

On my Bionic Beaver (Ubuntu 18.04 LTS) with Python 3.6.7, the following files are generated in `data/`. The first column gives the number of bytes.

```
 4 Apr 11 16:22 test_txt0.bin
 4 Apr 11 16:22 test_txt0_pb.txt
 9 Apr 11 16:22 test_txt0.txt
```

```
20 Apr 11 16:22 test_txt1.bin
 5 Apr 11 16:22 test_txt1_pb.txt
46 Apr 11 16:22 test_txt1.txt
```

The files `test_txt0.bin` and `test_txt0_pb.txt` encode `txt0` saved in `test_txt0.txt`. The extension `_pb` stands for padded bits. The combined size of the two encoded files, `test_txt0.bin` and `test_txt0_pb.txt`, is 4 + 4 = 8 bytes, whereas the size of the file with the original text, `test_txt0.txt`, is 9 bytes. In other words, we managed to save a byte.

The files `test_txt1.bin` and `test_txt1_pb.txt` encode `txt1` saved in `test_txt1.txt`. The size of the two encoded files, `test_txt1.bin` and `test_txt1_pb.txt`, is 20 + 5 = 25 bytes, whereas the size of the file with the original text, `test_txt1.txt`, is 46 bytes. This time we managed to compress by 46 - 25 = 21 bytes, which is not bad at all!

### Encoding and Decoding the Great "Moby Dick" by Herman Melville

The zip archive for this assignment contains the files `data/moby_dick_ch01.txt` and `data/moby_dick_ch02.txt` that contain the first two chapters of the great "Moby Dick" by Herman Melville. I downloaded these from Project Gutenberg (`https://www.gutenberg.org/ebooks/2701`).

On slide 27 in `cs3430_s20_lec24_huffman_part02.pdf` I posed the question of where the symbol weight maps come from. The class `CharFreqMap.py` gives a concrete answer to this question through the static method `computeCharFreqMap()` that creates a dictionary mapping all characters in a given text file to their frequencies in that file. If you save all chapters of "Moby Dick" in one text file, it'll create the frequency map of the entire novel. The static method `freqMapToListOfHuffmanTreeNodes()` in the `HuffmanTree` class takes this map and creates a list of `HuffmanTreeNode` objects.

The unit tests `test_ht_ut07(self, tn=7)` and `test_ht_ut08(self, tn=8)` compute character frequencies maps, build the corresponding Huffman trees, and use them to encode the first two chapters of "Moby Dick." Let's run them and look at what they save in `data/`. The first column gives the number of bytes for each file.

```
 6722 Apr 11 17:01 moby_dick_ch01.bin
    7 Apr 11 17:01 moby_dick_ch01_pb.txt
12285 Apr 11 17:01 moby_dick_ch01.txt
 4412 Apr 11 17:01 moby_dick_ch02.bin
    7 Apr 11 17:01 moby_dick_ch02_pb.txt
 8030 Apr 11 17:01 moby_dick_ch02.txt
```

Let's analyze what's happened. The size of `moby_dick_ch01.txt` is 12,285 bytes. The size of the two encoding files, `moby_dick_ch01_pb.txt` and `moby_dick_ch01.bin`, is 6,722 bytes + 7 bytes = 6,729 bytes. This gives us 12,285 - 6,729 = 5,556 bytes of compression. The size of `moby_dick_ch02.txt` is 8,030 bytes. The size of the two encoding files, `moby_dick_ch02_pb.txt` and `moby_dick_ch02.bin`, is 4,412 + 7 = 4,419. This gives us 8,030 - 4,412 = 3,618 bytes in savings. We did well!

A final touch of digital text oil to this picture of beautiful Huffman trees in our last assignment of this semester. Huffman trees generated from gigabytes or terabytes of data are precious and should be cherised and, therefore, persisted. The `BinHuffmanTree.py` has two methods `persist()` and `load()` that use `pickle` to persist generated trees and read them back into a running Python. You can run, if you want, the unit tests `test_ht_ut09(self, tn=9)`, `test_ht_ut10(self, tn=10)`, and `test_ht_ut11(self, tn=11)` and behold the awesome power of persistence!

## What To Submit

You'll have to modify only `HuffmanNodeTree.py`. But please zip all the files (except for the data directory) into `hw12.zip` and submit it in Canvas. It'll make grading your submissions faster for us.

Happy Hacking!