CPSC 231: Introduction to Computer Science for Computer Science Majors I

Assignment 4: Huffman Coding

Weight: 7%

Collaboration

Discussing the assignment requirements with others is a reasonable thing to do, and an excellent way to learn. However, the work you hand-in must ultimately be your work. This is essential for you to benefit from the learning experience, and for the instructors and TAs to grade you fairly. Handing in work that is not your original work, but is represented as such, is plagiarism and academic misconduct. Penalties for academic misconduct are outlined in the university calendar.

Here are some tips to avoid plagiarism in your programming assignments.

1. Cite all sources of code that you hand-in that are not your original work. You can put the citation into comments in your program. For example, if you find and use code found on a web site, include a comment that says, for example:

   ```
   # the following code is from
   https://www.quackit.com/python/tutorial/python_hello_world.cfm.
   ```

   Use the complete URL so that the marker can check the source.

2. Citing sources avoids accusations of plagiarism and penalties for academic misconduct. **However, you may still get a low grade if you submit code that is not primarily developed by yourself. Cited material should never be used to complete core assignment specifications. You can and should verify and code you are concerned with your instructor/TA before submission.**
3. Discuss and share ideas with other programmers as much as you like, but make sure that when you write your code that it is your own. A good rule of thumb is to wait 20 minutes after talking with somebody before writing your code. If you exchange code with another student, write code while discussing it with a fellow student, or copy code from another person's screen, then this code is not yours.
4. **Collaborative coding is strictly prohibited**. **Your assignment submission must be strictly your code**. Discussing anything beyond assignment requirements and ideas is a strictly forbidden form of collaboration. This includes sharing code, discussing code itself, or modelling code after another student's algorithm. **You can not use (even with citation) another student's code.**
5. Making your code available, even passively, for others to copy, or potentially copy, is also plagiarism.
6. We will be looking for plagiarism in all code submissions, possibly using automated software designed for the task. For example, see Measures of Software Similarity (MOSS - https://theory.stanford.edu/~aiken/moss/).
7. Remember, if you are having trouble with an assignment, it is always better to go to your TA and/or instructor to get help than it is to plagiarize. A common penalty is an F on a plagiarized assignment.

Late Penalty

Late assignments will not be accepted.

Goal

Writing a program with classes object and recursion.

Technology

Python 3

Submission Instructions

You must submit your assignment electronically. Use the Assignment 4 dropbox in D2L for the electronic submission. You can submit multiple times over the top of a previous submission. Do not wait until the last minute to attempt to submit. You are responsible if you attempt this and time runs out. Your assignment must be completed in **Python 3** and be executable with Python version 3.6.8+. You should not import any python libraries to create the Huffman trees or determine a Huffman code.

Description

In this assignment you are given a partially complete program designed to take an ASCII file and compress it. Compression is when you take information that occupies a certain quantity of space and through an algorithmic method reduce it to occupy less space by 'encoding' it. To be useful, this process should contain a method to 'decode' the data back from the compressed form to be read.

Compression is used across electronic media from simple text compression to image compression like jpg and video compression like mp4. In this assignment you will be complete two Python classes to finish the compression program. Instructions in the assignment will lead you through each of the classes to complete parts of each so that the program operates correctly at the end. Similar to how assignment 2 required you to complete functions to make that program complete, this assignment will have you complete objects and their internal methods (object functions) to make it work. This is a common software engineering challenge where a 'class' must be completed according to the specifications given to you.

You will be creating a solution to achieve Huffman Coding. Huffman Coding is optimal when encoding ASCII symbols one at a time. Optimal here is defined as the resulting text is reduced from the original byte quantity to the smallest byte quantity possible. Later courses in Computer Science discuss the math that supports this, however for this assignment we will follow a simple greedy algorithm that gives us a Huffman Code without needing to know why it is correct or optimal.

You do not have to deal with command line arguments, opening files, exceptions, in this assignment. The challenges will be completing two python files each that contain object methods used by the main program. **When completing these files you must implement the Huffman Tree and Encoding Table with your own code and are not allowed to import libraries to accomplish this.**

Huffman Code Example

helloworldhelloworldhelloworldhelloworldhelloworld

is 50 characters long and would take 50 bytes in ASCII to store

00011111 01001110 01001101 11000011 11101001 11001001 10111000 01111101 00111001 00110111 00001111 10100111 00100110 11100001 11110100 11100100 11011100

Is the binary string after encoding this data and only occupies 135 bits or 17 bytes. (The final 0 would be added to make it a rounded set of 8 bit bytes when written to a file.)

If you want to decode this string back to the original, then you would use the following table.

'd':1110
'e':1111
'h':000
'l':10
'o':01
'r':001
'w':110

000 1111 10 10 01 110 01 001 10 1110 is “helloworld”

Usage of Provided Program (i.e. Command Line Arguments)

python CPSC231F21A4.py **(no parameters in Pycharm)**

Prompts for <input_filename> to compress.

python CPSC231F21A4.py <input_filename> **(1 parameter in Pycharm)**

Compresses provided <input_filename> instead of prompting for filename.

python CPSC231F21A4.py <input_filename1> <input_filename2> **(2 parameters in Pycharm)**

Creates Huffman Tree for <input_filename1> and <input_filename1> and determines if structure of the trees are equal (==).

python CPSC231F21A4.py <arg1> <arg2> <arg3> **(3+ parameters in Pycharm)**

(exits program with descriptive error indicating too many arguments)

-V **(optional)** argument that turns on detailed printing **(extra parameter in Pycharm)**

When run with one input file **<input_filename>**, three files are created in **output** folder.

<input_filename>.bin the compressed binary file result.

<input_filename>.tbl the encoding table used to create the compressed result.

<input_filename>.txt the compressed result as a human readable binary string. (not a compressed file)

In the text: helloworldhelloworldhelloworldhelloworldhelloworld

‘r’,’w’,’d’,’e’,’h’ occur 5 times

‘o’ occurs 10 times and ‘l’ occurs 15 times

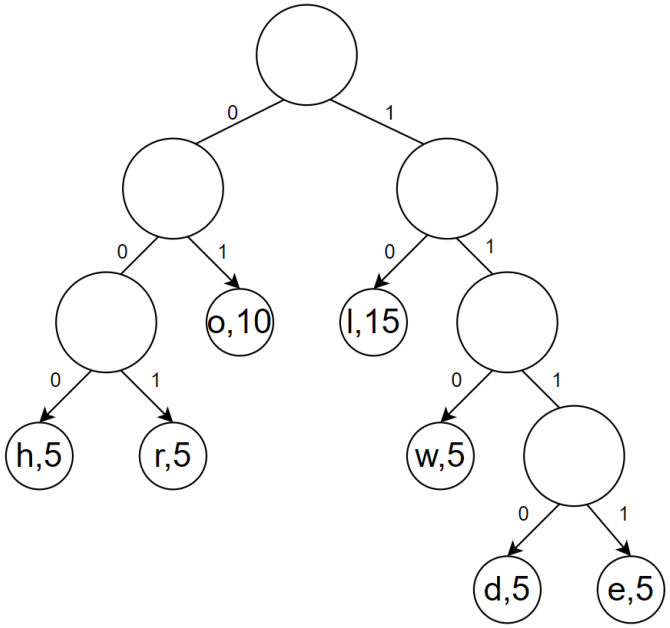
Huffman Trees compress this information by replacing the 8 ASCII bits used for each symbol with a shorter bit sequence. To optimize things longer bit sequences are used for less common characters (those that occur 5 times here) and shorter sequences for more common characters (like ‘l’ and ‘o’). A key requirement for encoding is that the bit sequences are unique from their start to end. You will notice that there is only one path from the top of this tree to each letter. Each of these paths is a unique combination of 0/1. An optimal encoding is a tree in which more common symbols will be closer to the top of the tree (and therefore shorter bit sequences) and those at bottom of tree will be less common (but have longer bit sequences). To determine the structure of these tree we will use a Huffman Coding algorithm that will combine(sum) together less common characters two at a time until all are combined.

Encoding

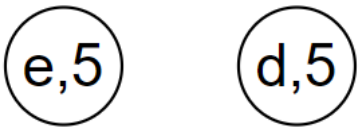
'd':1110
'e':1111
'h':000
'l':10
'o':01
'r':001
'w':110

If you read this tree from top to a letter like ‘w’ you would have 1, 1, 0 which is the encoding of ‘w’.

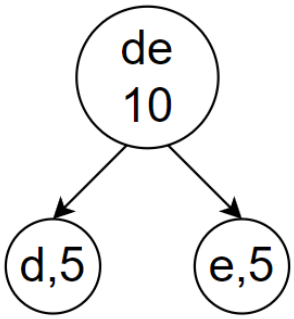
Huffman Tree



Merge-Before



Merge-After

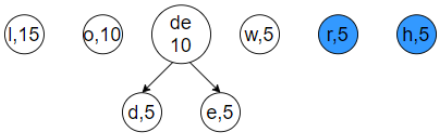


Full Example (sort larger count to front of list, tie -> larger ASCII to front) Merge last 2

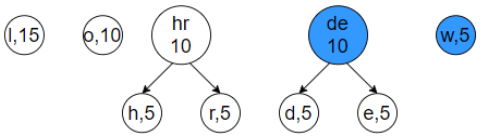
Initial (sorted)

Merge e,d

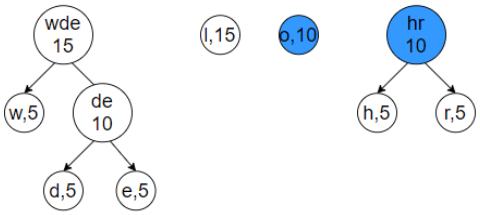
Merge r,h



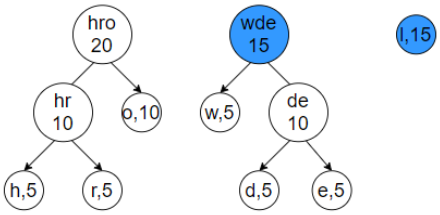
Merge de, w



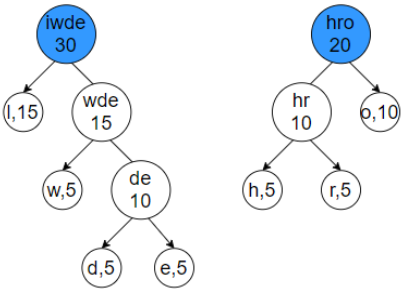
Merge o, hr



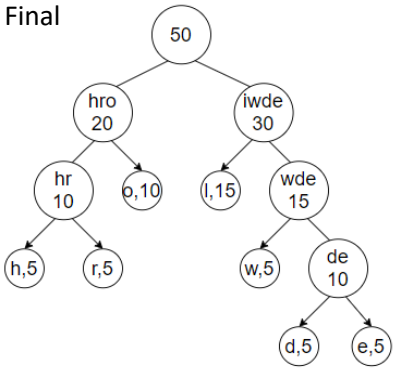
Merge wde, l



Merge iwde,hro



Final



Part 1 HuffmanTree: Constructor/make_trees

You have been provided with a HuffmanTree.py file. Edit this file to complete this part.

Create the constructor __init__()

The constructor will take 5 parameters (of course remember the first parameter of self to make 6 total parameters):

1. the **char** to be stored in the tree
2. the **count** of how many times that **char** occurred in the text
3. a link to a **left** HuffmanTree (default None)
4. a link to a **right** HuffmanTree (default None)
5. a boolean **bit** that stores if this tree was reached with a 0 (False) or a 1 (True) (default None)

Create a class function make_trees(dictionary):

The class function will take 1 parameter. This is **class function** so it should not be indented inside the class (the method does not have self as a parameter). Put it at end of HuffmanTree.py file.

1. Dictionary **dictionary** which stores each symbol from the input file as a key **char** such that **dictionary[char] = count**, where count is how many times that symbol occurred in the file

This method should loop through all symbols (keys) in the dictionary and create a **default** HuffmanTree for each symbol (left,right,bit will all be None). These HuffmanTrees should be collected in a list and returned. Note, you will need to use the constructor from earlier to create a HuffmanTree storing each **char** and associated **count**. Ten keys would produce ten Huffman Trees storing one char/count.

Once Part 1 is complete the first part of the program should operate. At this point the program doesn't do anything interesting but successfully read the input file and create an initial list of HuffmanTrees.

Part 2 HuffmanTree: Comparison/sort

To start the algorithm of Huffman Coding we need to order the list of trees we created earlier such that the trees storing a symbol which occurred the **most** occur **first** and are followed by those that occurred **less**. Those that occurred the **least** at the **end**. *(We will select these least from end using pop() to merge to make Huffman Tree.)*

To accomplish this you need to add a method to your HuffmanTree class. This method will compare one HuffmanTree to an**other** HuffmanTree and allow you to sort a list of HuffmanTrees using list.sort().

Create a method __lt__(other): *[also known as < operator]*

The method will take 1 parameter. This is a method so it should be indented inside the class. (of course remember the first parameter of self to make 2 total parameters)

1. HuffmanTree **other** that is being compared to the current HuffmanTree **self**.

- This method should return True if the **count** for **self** is greater than that of **other**.

- This method should return False if the **count** for **self** is less than that of **other**.
- If the counts are the same, then it should return True if **self**'s **char** is greater than that of **other**, otherwise False**.**

With this complete, the initial list of HuffmanTrees can be sorted. However, things are still challenging because we can't see (print) what the order is in a useful way. You'll notice the print statements in the code aren't very informative. Part 3 will fix that.

Part 3 HuffmanTree: String/Representation

To accomplish this you need to two methods to your HuffmanTree class. These methods will create a printable short form of each HuffmanTree that can be produced using str() and a detailed formed using repr().

Create a method __str__():

The method will take 0 parameters. This is a method so it should be indented inside the class. (of course remember the first parameter of self to make 1 total parameters)

This method should produce a string of the form **"(char,count,left,right,bitstring)"**

Note that if left or right is None then it should use "None". For bit=True it should use "1" and bit=False it should use "0".

Use **repr(char)** for **char** because it will show the space symbol as ' ', <enter> as '\n', and <tab> as '\t'.

Create a method __repr__():

The method will take 0 parameters. This is a method so it should be indented inside the class. (of course remember the first parameter of self to make 1 total parameters)

This method should produce a string of the form **"HuffmanTree(char,count,left,right,bit)"** where each value in it like **char,count,etc.** is the representation form **repr(char),repr(count),etc.**. Note that repr(left), repr(right) is a recursive call on the **left** HuffmanTree and the **right** HuffmanTree.

With this complete the program will now print the sorted list of HuffmanTrees. Now the program can attempt to make the final HuffmanCoding tree. The algorithm is already written in the program, but it relies on a merge operation to combine HuffmanTrees. Part 4 will create that.

Part 4 HuffmanTree: Merge/Recursive Structure

To complete part 4 you need to create another **class function**.

Create a class function merge(t1, t2)

The method will take 2 parameters. This is class function so it should not be indented inside the class, and the method does not have self as a parameter

1. Huffman tree **t1**
2. Huffman tree **t2**

This method should return a new HuffmanTree. This new HuffmanTree should be created using the previous constructor. This HuffmanTree will store

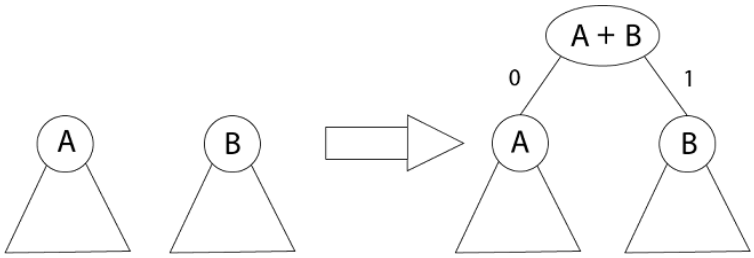
char = A.char + B.char

count = A.count + B.count

left=A

right=B

bit = None



To store the encoding of bits, assign the bit 0 to **A** and the bit 1 to **B** using the **bit** boolean for each of left/right.

To determine which of t1/t2 is A and B we want A to be tree with a smaller **count**. If **count**'s are equal, then put tree with smaller **char** (ASCII value) in A.

With this complete your program creates a final HuffmanTree. We have one more method to add to the HuffmanTree class. This method will let us compare the HuffmanTrees create when we are given two input files **<input_filename1> <input_filename2>.**

Part 5 HuffmanTree: Equality/Recursive Function

To accomplish this you need to add a method to your HuffmanTree class. This method will compare one HuffmanTree to an**other** HuffmanTree and allow you decided if they are equal (**==)**.

Create a method __eq__(other):

The method will take 1 parameter. This is a method so it should be indented inside the class. (of course remember the first parameter of self to make 2 total parameters)

1. HuffmanTree **other** that is being compared to the current HuffmanTree **self**.

This method should return False if other is None.

This method should return True if the **char**, **left**, and **right** of **self/other** are all **==.** (Note, we don't ignore **char** in __eq__, but we do ignore **count**.)

Otherwise, if should return False.

This program should now be able to run for usages that involve giving two input files.

Ex. **sampletree2.dat** and **sampletree2-1.dat** should produce “Trees are the same in structure!”

```
Tree 1:
        -1-> ('c',3)
    -1->
        -0-> ('b',2)
-->
    -0-> ('a',4)

Tree 2:
        -1-> ('c',3)
    -1->
        -0-> ('b',3)
-->
    -0-> ('a',4)
```

sampletree2.dat and **sampletree3.dat** should produce “Trees are the different in structure!”

```
Tree 1:
        -1-> ('c',3)
    -1->
        -0-> ('b',2)
-->
    -0-> ('a',4)

Tree 2:
    -1-> ('c',6)
-->
        -1-> ('b',3)
    -0->
        -0-> ('a',2)
```

We still need to make the program encode our input file into and output file. This will need us to complete a second class. This class is partially complete. It already has a **encode** method that will take text in text form and turn it into a **binary string**. However, this **EncodingTable** still needs to be constructed.

Part 6 Encoding Table: String

You have been provided with a EncodingTree.py file. Edit this file to complete this part. Please note until you complete Part 7, there will be nothing in the object to print.

We want to be able to visualize our encoding table in a consistent way. To accomplish this you need to add a method to your EncodingTable class. This method will create a printable short form of each EncodingTable that can be produced using str().

Create a method __str__():

The method will take 0 parameters. This is a method so it should be indented inside the class. (of course remember the first parameter of self to make 1 total parameters)

This method should produce a string from the dictionary **encode** storing the encoding where each line in the string is an entry such as

'd':1110
'e':1111
'h':000
'l':10
'o':01
'r':001
'w':110

These entries should be sorted by character.
Use **repr(char)** for **char** because it will show the space symbol as ' ' and enter as '\n'.

Part 7 Encoding Table: Constructor/Recursive Method

You have been provided with a EncodingTree.py file. Edit this file to complete this part.

Examine the constructor __init__()

The constructor will take 1 parameter (of course remember the first parameter of self to make 2 total parameters):

1. The HuffmanTree **tree** to be converted into an **encode** dictionary

This method is actually complete for you, it is the second recursive function **recurse** that needs to be completed.

Complete the method recurse(tree, code)

This method will take 2 parameters (of course remember the first parameter of self to make 3 total parameters):

1. the HuffmanTree **tree** to use to determine the codes
2. the current string **code** created so far

This is a recursive function. Which means I will define this function recursively for you.

Every recursive call: If the **bit** for the **tree** is not **None,** then modify the **code** to add a "1" if the bit was True or "0" if the bit was False.

Base Case: If you reach a HuffmanTree **tree** which has a None **left** and **right** sub-tree. Then store the current parameter **code** in the dictionary **self.encode** at the symbol **tree.char.**

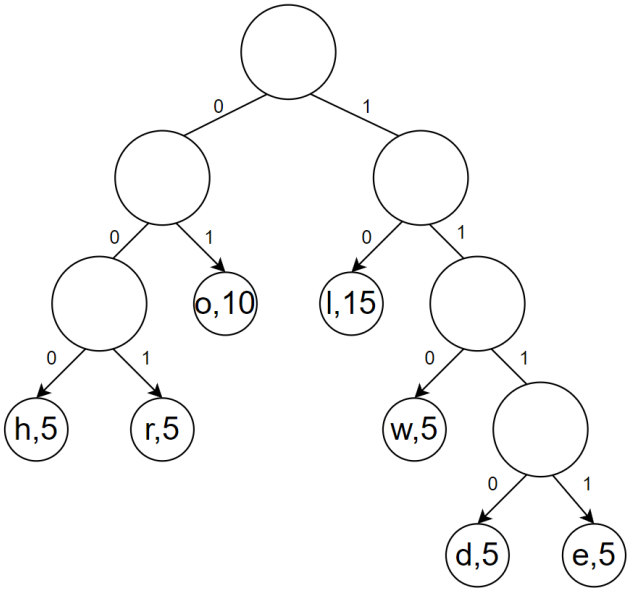
Recursive Case: If you aren't at the base case, then recursively call this method on **tree.left** with the current **code,** and on **tree.right** with the current **code.**

Idea of code:

We start at top of tree with code = "". We will go down left side first, but eventually will also explore down right side. We add "1" or "0" to this code based on if we have bit==True or False stored in the sub-tree we move into.

If we reach a bottom node and can't go further. We'll have tracked the code from the top to that point. We can now store this string into our dictionary of encodings for the character store at this point at the bottom of the tree.

Once we store a code our function will fall out to the previous location up the tree, and we will explore the right side like we did with the left.



For example, we would start at the top of tree. We have a bit==None so we don't add (0/1). Then we recurse on left side with code="". We have bit==False so we add "0". Then we recurse on left side with code="0". We have bit==False so we add "0". Then we recurse on left side with code="00". We have bit==False so we add "0". Our left=right=None so we store for 'h' the code="000" and return to the node above. Then we recurse on right side with code="00". We have bit==True so we add "1". Our left=right=None so we store for 'r' the code="001".

Additional Specifications:

Ensure that your program meets all the following requirements:

- You should have the class, your name, your tutorial, your student id, the date, and description at the top of your HuffmanTree/EncodingTable files in comments. Marks are given for these.
- You should not rename HuffmanTree/EncodingTable/CPSC231F21A4 files.
- **You should not import ANY libraries to complete the regular assignment.**
- **You should not need any constants for your classes. However you might use one for strings "0" and "1".** Use constants appropriately. Your TA may note one or two magic numbers as a comment, but more will result in lost marks.
- **Follow the variable/function names given in the assignment description.** The CPSC231F21A4.py file relies on specific names be followed in some cases when using your classes.
- Use in-line comments to indicate blocks of code and describe decisions or complex expressions.

- You should comment your functions descriptively. (see Assignment 2)
- **Break and continue are generally considered bad form when learning to program**. As a result, you are **NOT** allowed to use them when creating your solution to this assignment. In general, their use can be avoided by using a combination of if statements and writing better conditions on your while loops. You are allowed to use return within a loop inside a function, or sys.exit(1) in a loop of a function when finding an error to exit.
- You should not perform error checking in this assignment when writing your classes. The provided code already performs the input error checking necessary.

Bonus:

For half of the bonus write another program called CPSC231F21A4-Name-Bonus.py that is able to read an encoded file like samplehello.dat.txt and an encoding table like samplehello.dat.tbl and will decode that file.

Ex. CPSC231F21A4-Name-Bonus.py samplehello.dat.txt samplehello.dat.tbl

Will print helloworldhelloworldhelloworldhelloworldhelloworld to terminal.

For the full bonus, expand the previous program that when using a "-b" flag will read a binary file like samplehello.dat.bin and decode that file as well using the encoding table like samplehello.dat.tbl.

Ex. CPSC231F21A4-Name-Bonus.py samplehello.dat.bin samplehello.dat.tbl -b

Will print helloworldhelloworldhelloworldhelloworldhelloworld to terminal.

Grading:

This assignment will be graded on a combination of functionality and style. A base grade will be determined from the general level of functionality of the program (Does it handle arguments correctly? Does it read input files correctly? Does it draw correctly? Does it print the requested output correctly?). The base grade will be recorded as a mark out of 12.

The assignment will be graded out of 12, with the grade based on the program's level of functionality and conformance to the specifications. The program must not have syntax errors to get more than an F. A program with runtime errors that occur during **proper usage of the program**, every time it runs, will not get better than a C grade. Runtime errors are your program crashing.

Your TA will begin by grading your code with a general functionality grade starting point and subtract marks when smaller specifications are unfilled.

The total mark achieved for the assignment will be translated into a letter grade using the following table:

Submit the following using the Assignment 4 Dropbox in D2L:

1. HuffmanTree.py EncodingTable.py (optional CPSC231F21A4-Name-Bonus.py)

Grading:

Mark	Letter Grade	Starting Point Guidelines (not a final grading scheme!)
13	A+	Appears to fulfill assignment and bonus spec
12	A	Appears to fulfill assignment spec
11	A-	
10	B+	
9	B	Can make Huffman Trees and check equality but Encoding Table not working
8	B-	
7	C+	
6	C	Creates Huffman Trees but can't merge them
5	C-	
4	D+	
3	D	Can only create Huffman Trees but not sort them
0-2	F	Syntax errors or barely started code

As a reminder, the University of Calgary assigns the following meaning to letter grades:

A: Excellent – Superior performance showing a comprehensive understanding of the subject matter

B: Good – Clearly above average performance with generally complete knowledge of the subject matter

C: Satisfactory – Basic understanding of the subject matter

D: Minimal Pass – Marginal performance; Generally insufficient preparation for subsequent courses in the same subject

F: Fail – Unsatisfactory performance