UNIVERSITY OF SASKATCHEWAN
Department of Computer Science
176 Thorvaldson Building
110 Science Place, Saskatoon, SK, S7N 5C9, Canada
Telephine: (306) 966-4886, Facimile: (306) 966-4884

Assignment 8

A full application

Date Due: April 6, 2017, 7pm **Total Marks: 50**

Question 0 (10 points):

Purpose: To force the use of Version Control in Assignment 8

Degree of Difficulty: Easy

You are expected to practice using Version Control for Assignment 8. This is a tool that you need to be forced to use, even when you don't need it, so that when you do need it, you are already familiar with it. Do the following steps.

1. Create a new PyCharm project for Assignment 8.
2. Use `Enable Version Control Integration...` to initialize Git for your project.
3. Download the files `Huffman.py`, `Encoder.py`, `Decoder.py`, and add them to your project.
4. Before you do any coding or start any other questions, make an initial commit.
5. As you work on each question, use Version Control frequently at various times when you have implemented an initial design, fixed a bug, completed a question, or want to try something different. Make your most professional attempt to use the software appropriately.
6. When you are finished your assignment, open the terminal in your Assignment 8 project folder, and enter the command: `git --no-pager log` (double dash before the word 'no'). The easiest way to do this is to use PyCharm, locate PyCharm's `Terminal` panel at the bottom of the PyCharm window, and type your command-line work there.

You may need to work in the lab for this; Git is installed there. Not having Git installed is not really an excuse. It's like driving a car without wearing a seatbelt. It's not an excuse to say "My car doesn't have a seatbelt."

What to Hand In

After completing and submitting your work for Questions 1-3, open a terminal window in your Assignment 8 project folder. Run the following command in the terminal: `git --no-pager log` (double dash before the word 'no'). Git will output the full contents of your interactions with Git in the console. Copy/paste this into a text file named `a8-git.log`.

If you are working on several different computers, you may copy/paste output from all of them, and submit them as a single file.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 10 marks: The log file shows that you used Git as part of your work for Assignment 8. For full marks, your log file contains
  - Meaningful commit messages.
  - At least two commits per question for a total of at least 6 commits. And frankly, if you only have 6 commits, you're pretending.

Solution: The submitted log might in the form of a text file, or a couple of screenshots or camera grabs of the PyCharm interface. Ideally, the log file was submitted, but there were enough problems with Windows and setting the path properly that we have to be a little generous here.

Notes to markers:

- There are two things to look for: the number of commits and the quality of the commit messages.
- The meaningful commit messages criterion: a message has to be about the work. Implemented a function, fixed a bug, started a question, added testing, those kinds of things.
- Give 5 marks for good commit messages; 3 marks for anything else.
- Students were told at least six commits (2 per question).
- Give 5 marks for 6 commits or more, and 2 marks for anything less than 6.

Question 1 (10 points):

Purpose: To implement an interesting recursive algorithm based on tree traversals.

Degree of Difficulty : Moderate

On Moodle you will find a Python class file named `Huffman.py`, which contains the definition for the `HuffmanTree` class. The Huffman class defines four attributes:

- `__freq`: an integer representing the frequency data for the node.
- `__char`: a character value
- `left`: a left subtree
- `right`: a right subtree

Most of the class is complete, including the initializer, and a couple of methods. You should study the definition carefully. Because A Huffman tree is a binary tree, it should feel familiar.

The file `Huffman.py` also defines a script that builds a Huffman tree corresponding to the example shown in class. Starting from a list of frequencies, the example builds Huffman tree leaf nodes first, then through a series of combination steps, results in a completed Huffman tree. At the end, the script calls a method named `build_codec()` which is part of the class definition, but not fully implemented. Currently the script displays a bunch of trees, but at the end, when it tries to display a dicitonary, it displays nothing because the dictionary contains no data.

Your task in this question is to implement the method `build_codec()`. Assuming the Huffman tree is correctly constructed, this method should traverse the complete tree, using one of the standard traversals, so that by the end, it will produce a dictionary containing characters (key) and codes (values) made from the digits `'0'` and `'1'`.

The code for a character depends on the path from the root to the leaf. Every leaf in any tree has a unique path from the root to the leaf; this is a property of all trees. Because the path is unique, the path can be described by a sequence of choices at each node starting at the root: go left, or go right. If we represent the decision to go left with `'0'` and the decision to go right with `'1'`, each leaf node has a unique code.

Implement `build_codec()`. When complete, the script will produce a list of codes that should be very similar to the codes presented for the example in class. A correct implementation of `build_codec()` output of the script should end with the following, or something fairly similar:

```
I:11
D:100
G:1010
E:01
A:00
C:1011
```

The script outputs various trees as well; your work is to obtain the codes, as shown. The order of the codes displayed doesn't matter.

Make sure you make full use of Version Control, by making regular commits. You will be expected to hand in a log file of your interaction with Git for this assignment, so be sure to practice using it.

What to Hand In

- Your Python file named `a8q1.py` containing all of the code provided, and a completed `build_codec()` method.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 5 marks: Your implementation of `build_codec()` correctly produces codes from a correctly constructed Huffman tree.
- 5 marks: Your implementation `build_codec()` is well-designed, documented, and shows evidence of defensive programming.

Solution: A complete solution appears in the file `Huffman.py`.

The relevant method is named `build_codec()`. The snippet below shows one way to implement it. The method itself is not recursive; recursion is performed by the internal function `encoder()`. Notice that the dictionary named `codes` is local to the method. We've done this a couple of times. The `encoder()` function is similar to a tree traversal where nothing happens at the treenode if it is not a leaf.

```
    def build_codec(self):
        """
        Purpose:
            Build a dictionary of char-code pairs from the Huffman tree.
        Return:
            :return: a dictionary with character as key, code as value
        """
        codes = {}
        def encoder(tree, code):
            if tree.is_leaf():
                codes[tree.get_char()] = code
            else:
                encoder(tree.left, code+'0')
                encoder(tree.right, code+'1')

        if self.is_leaf():
            codes[self.__char] = '0'
        else:
            encoder(self,'')
        return codes
```

The main trick with building a code is that the code is built up as the tree is descended. A binary digit is added to the string in the recursive call. This ensures that each call has a unique code; if strings were mutable, this would not work!

Notes to markers:

- There has to be some form of recursion going on.
- The simplest way to build the code is as shown, but it's not the only way. It is possible to use a list, or do some of the work in the body of the function.
- For full marks for correctness, the codes just need to be entered into the dictionary correctly.
- For full marks for design and style, the method needs to be documented fully, as shown. The `self` object need not be discussed.
- It's okay if marks are allocated generously here.

Corrections and clarifications

1. **29/03/2017**: For the function `build_codec()`: If you assume that the Huffman tree has at least 3 nodes (a root with 2 leaf-children), then a tree traversal will be able to generate the codes. However, if the Huffman tree consists of a leaf, representing one character, then a traversal may not be able to generate the codes. Thinking ahead to Question 2, this could happen if you are given a file consisting of possibly several repetitions of a single character, such as `'aaaaaaa'`.

   In the case of a Huffman tree with exactly one node, a leaf, return a dictionary with a single entry: the character, and a one digit code (either `'0'` or `'1'`; it doesn't matter). This should be done as an alternative or special case, instead of the tree traversal.

Question 2 (20 points):

Purpose: To implement the Huffman algorithm for Huffman codes. To implement output to a file with a specific format.

Degree of Difficulty: Moderate

On Moodle you will find a Python class file named `Encoder.py`, which contains a collection of functions to encode text files using Huffman coding. Read the code carefully. Several of the functions are very simple, reading data from a file, and writing data to a file. There is a function named `count_characters` which should be familiar to you from your work on Assignment 7.

There are 2 functions in the program that are not fully implemented:

- `build_codec(freq_list)` This function has to construct the Huffman tree from the given frequency list `freq_list`. Further information about this function is below.
- `encode(strings, codec)` The `encode()` function prepares all the data for output. Further information about this function is below.

`build_codec(freq_list)` The algorithm to build a Huffman tree was discussed in class. It has the following steps.

1. Create a leaf for each character-frequency pair in the frequency list (something like this appeared in the script for Question 1). Put all the leaf nodes into a list of trees.
2. Repeat the following steps until the list of trees has exactly one tree in it:
   (a) Remove the two trees with the lowest frequency. Ties do not matter, so if several trees are tied for the smallest frequency, you may pick any one of them.
   (b) Combine the two trees combining their frequencies. The Huffman object initializer will do the work, if you use it correctly.
   (c) Put the combined tree back in the list.
3. Once all the trees have been combined into a single tree, use the method you implemented for Question 1 to build the codec, that is, the dictionary of character (key) code (value) pairs.

`encode(strings, codec)` This function has to prepare all the data for output to a file. The format of the output file is as follows:

1. Line 0: two integers, a and b, separated by a space. The first integer, a, indicates that there are a code-character lines in the file. The second, b indicates that there are b lines of coded text in the file after the code-character lines. See the example encoded files, and also the program for Question 3.
2. Lines 1 through a: These are the kinds of code-character lines that were used in Assignment 7: a sequence of digits '0' and '1', followed by a colon (:) followed by a character in single quotes.
3. Lines $a + 1$ through $a + b + 1$: lines of coded text.

In Assignment 7, we assumed exactly 1 line of coded text; here, we have more than one, but b tells us exactly how many.

The function `encode` simply prepares a list of strings, one string for each line in the output file. The work is simply to create the strings, so that `write_file()` can send them to the output.

You can use the code in `Decode.py`, given to you for Question 3, to test your output. It should be able to read a correctly formatted coded file.

Be sure to keep on using the Version Control tools, at least after you've finished each function, but possibly more often. You will be expected to hand in a log file of your interaction with Git for this assignment, so be sure to practice using it.

What to Hand In

Your Python file named `a8q2.py` containing all of the code provided, including complete implementations for

- `build_codec()`
- `encode()`

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 5 marks: Your implementation of `build_codec()` correctly constructs a Huffman tree.
- 5 marks: Your implementation `build_codec()` is well-designed, documented, and shows evidence of defensive programming.
- 5 marks: Your implementation of `encode()` correctly constructs a list of strings for output (but does not output them directly).
- 5 marks: Your implementation `encode()` is well-designed, documented, and shows evidence of defensive programming.

Solution: A complete solution can be found in the file `Encoder.py`, with help from the file `HuffmanHeap.py`. The relevant methods are presented below.

The Python code for building a Huffman tree is found in the method `build_codec()`. The Python code has enough abstraction that the Huffman tree building algorithm is pretty easy to follow: create a list of trees, take two smallest trees, combine them, and replace the result back in the list.

```
def build_codec(freq_list):
    """
    Purpose:
        Build a dictionary containing character:code pairs from
        the frequency list.
    Pre-conditions:
        :param freq_list: A list of (character,frequency) pairs.
    Return:
        :return: a dictionary
    """
    # sort the frequency list
    freq_list.sort(key=lambda p: p[1])

    # create the queue of Huffman trees
    # note: a new ADT for this purpose!
    hq = HH.HuffmanHeap(
            [HT.HuffmanTree(freq=f, char=c) for c, f in freq_list]
            )

    # dequeue 2 trees, combine them, and enqueue the resulting tree
    while len(hq) > 1:
        t1 = hq.dequeue()
        t2 = hq.dequeue()
```

```
        hq.enqueue(HT.HuffmanTree(left=t1, right=t2))

    #build a codec from the only tree that's left
    survivor = hq.dequeue()
    return survivor.build_codec()
```

The nice level of abstraction is made possible by a class called HuffmanHeap, which I showed in class. It hides all the work of ensuring that the tree nodes are dequeued in the right order (the smallest first).

There are perfectly acceptable alternative solutions. The algorithm requires the two smallest tree-nodes. These can be obtained by a linear search, or by re-sorting the list every time a new tree is added to the list. Ideally, some of the steps will be delegated to a separate function

The method `encode` should create the data to be sent to the output file, as a list of strings. A model solution for the method is given below. It doesn't have a lot of surprises. It has create the right strings and put them in the list in the right order.

```
def encode(strings, codec):
    """
    Purpose:
        Use the codec to create the data to be sent to the output file.
    Pre-conditions:
        :param strings: A list of strings to encode.
        :param codec: A dictionary containing character-code pairs
    Return:
        :return: a list of strings including:
            the number of codes the number of coded lines,
            the codes
            the coded lines
    """
    output = []
    output.append(str(len(codec)) + ' ' + str(len(strings)))
    for char in codec:
        output.append(codec[char] + ':' + "'" + char + "'")
    for s in strings:
        encoded = []
        for char in s:
            encoded.append(codec[char])
        output.append(''.join(encoded))
    return output
```

Notes to markers:

- Very few students will have implemented the Huffman Heap, I presume. It is not required for full marks!
- Most will either re-sort the list every time, or will use linear search.
- For full marks for correctness of `encode`, it is enough that the strings are put into a list in the correct order. Marks should be deducted for algorithmic errors (e.g., omitting some information, or using the wrong format).
- For full marks for correctness of `build_codec()`, it is enough that the algorithm be implemented. Marks should be deducted for algorithmic errors (e.g., not keeping the list sorted).

- For full marks for design and style, the method needs to be documented fully, as shown.
- It's okay if marks are allocated generously here. I expect most submissions will get the file format right, since they have a Decoder program (A8Q3) for testing.
- Note that any code constructed by building a binary tree will lead to output that can be coded and decoded. The Huffman algorithm guarantees the best code, not the only code!

Question 3 (10 points):

Purpose: To practice the skill of reading and modifying someone else's code. To recognize that efficiency of memory use is sometimes as important as computational efficiency.

Degree of Difficulty: Moderate

On Moodle, you'll find a program named `Decoder.py`. It is a fully functional program that is able to decode files that have the expected file format (as outlined in Question 2).

The program follows a simple design, and is well-documented. All the pieces should be familiar to you, as Assignment 7 had a question very much like this.

However, there is a serious efficiency problem with this program. As you can see, the design requires that the entire file be read at once, and that all the data get passed from one function to another. For example, the function `read_file()` reads the entire file's contents, and stores it all in a single list of strings. From there, the lines of encoded text are decoded one at a time, but all the encoded text is stored in a list.

For small files, this is okay, since modern computers are fairly big. However, for very large files, bigger than what we're working with, it is very inefficient to store the whole file's worth of data in memory all at once. It is better, when it is possible, avoid this situation.

Your job is to take the given code, and rewrite parts of it, so that the program does not store all the data in the file all at once. This task will require some analysis of the current code, and some redesign. You do not need to write new code, or new algorithms. You just need to arrange it so that the program is not storing all file's data at the same time.

For this question, the use of Version Control may be especially useful, as you will want to save your program at various stages of development while you rewrite it. Whether you actually use the history of your development or not, ensuring that you could is a professional attitude.

What to Hand In

A file named `a8q3.py` containing the revised program. Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 5 marks: Your program correctly decodes encoded files, and does not store the entire contents of the encoded file in memory at the same time.
- 5 marks: Your program is well-designed, documented, and shows evidence of defensive programming.

Solution:

A model solution for for this problem can be found in the Python file `Decoder.py`. Some of the salient points are shown below.

The key issue is that in the original program, a string, named `s` (rather unhelpfully), containing all the data in the named file, is passed around from function to function. For really big files, this is not the best way to program an application like this. Fortunately, this kind of application can do much better. We only need to read the file one line at a time. This allows us to process the whole file, without having it all in memory.

In order to ensure that the bare minimum of data is kept in memory, it is enough to re-arrange the pieces given in the original. In my solution, I've removed the function `read_file()` completely. Instead

of looking at the list for the data, I use calls to the file method `read_line()`, and I never accumulate any more than one line from the data at any time.

Here's what the `main()` function looks like:

```
def main():
    """
    Purpose
        The main program.
        Usage: python3 Decoder.py <filename>
        Sends output to DEFAULT_OUTPUT_FILE
    Return:
        :return: None
    """
    if len(sys.argv) != 2:
        print('Usage: python3', sys.argv[0], '<filename>')
        print('-- sends output to', DEFAULT_OUTPUT_FILE, '-- ')
        return

    the_file = open(sys.argv[1])
    assert the_file, 'Could not open file'

    line = the_file.readline().rstrip()
    assert line, 'File had no data'

    sizes = line.split()
    assert len(sizes) == 2, 'First line of file corrupted'

    code_size = int(sizes[0])
    assert code_size > 0, 'Unexpected code size value'

    message_size = int(sizes[1])
    assert message_size > 0, 'Unexpected message size value'

    dc = build_decoder(the_file, code_size)

    for l in range(message_size):
        line = the_file.readline().rstrip()
        message = decode_message(line, dc)
        print(message)

    the_file.close()
    return
```

Note the use of `assert`. This is optional, in the sense that it is not required by the question description. However, its use halts the program at the points where things could possibly go wrong. There is a saying in software developer circles: "All input files are evil until proven otherwise."

The `build_decoder()` function also changed. Instead of taking a list of strings as an argument, it takes a file object. Instead of accessing the list for data, it reads the file by calling the `read_line()` method.

```
def build_decoder(afile, code_size):
    """
    Purpose:
        Build the dictionary by reading some lines from the given file
```

```
    Preconditions:
        :param afile: an open file, ready for reading
        :param code_size: the number of lines of code
    Return:
        :return: a dictionary whose keys are 'CODE' and values are <char>
    """
    codec = {}
    for l in range(code_size):
        line = afile.readline().rstrip()
        assert line, 'Expected a code line at line '+str(l)
        cchr = line.split(':')
        code = cchr[0]
        # the normal case: the split created 2 strings
        if len(cchr) == 2:
            char = cchr[1]
        else:
            # if the char is ':' itself, the split will create more than 2
            char = "':'"

        codec[code] = char[1] # The input file has ' ' around the character
    return codec
```

I've added a bit of code that allows the processing of the `':'`, which is a little tricky because I was using it to separate the code-character lines in the file. This is entirely optional, but perhaps educational.

The function `decode_message()` has not changed at all.

Notes to markers:

- For full marks for correctness, it is enough that the program does the decoding correctly without accumulating strings in lists. Marks should be deducted for any accumulation.
- The dictionary of code-char data (the *decoder*) must accumulate strings. There is no choice here.
- It is completely up to the student which parts of the program are broken into functions. It is not incorrect to have the whole program as one long script (though I'd suggest they lose style points for that).
- For full marks for design and style, the program should be well documented, as usual. Some functions might have disappeared, which is fine, though I think `decode_message()`, since it hasn't changed, should not have been modified.