Homework Week 5: Huffman Coding

(Last Submission Date: Tuesday, Oct. 21st at 11:59pm)
Working alone? Extra credit for early submission!

Goal: Practice implementation of an algorithm using techniques learned in CS3xx courses.

Activity: Given text to encode, build a Huffman tree to assist in data compression.

Teamwork: Pairs within the same section of CS483 are permitted, however ***you cannot get the extra credit*** for early submission this way. *One set of code should be produced and uploaded to GradeScope as a GROUP submission.*

Details

Motivation

Understanding an algorithm on paper and implementing an algorithm using a programming language can feel like two very different processes. Because of this, there are many algorithm "visualizers" which show students how an algorithm works, printing out the "state" of the algorithm after certain lines of code. These visualizations, in turn, are simply programs that you can write yourself.

Visualizations are actually very common, and not just for learning algorithms. Sometimes you need to demonstrate to yourself and others that your software is working, and visualizing can help you wrap your head around a complicated code scenario. Visualizing is also helpful for debugging – many of you who were in CS112 at GMU probably started learning to code by looking at a code "visualizer".

You are going to use your CS2xx and CS3xx skills in this assignment, while reinforcing your understanding of greedy algorithms we've learned (building Huffman trees), to produce a visualization. Below are some additional industry skills you will practice as well:

1. **Code Augmentation** – For this assignment you aren't writing the entire program. Instead, you need to fit your code into a larger code base. So, you'll have to read/understand other people's code.
2. **Code Reuse** – No rerolling your own tree data structures for this project, there's a library that does it for you. That means you have to read/understand a library you haven't used before.

This is probably 80-85% of what you do in industry… read a big code base, figure out where you need to make modifications/additions, add your code, and test with the larger code base. Another 10-15% of the time you might be writing code "from scratch" but as part of a team, so you'ss still need to read everyone else's code and perform edits *only* to your part of the code. In both these scenarios you'll often be using libraries written by third parties.

(Yes, there is another 5-10% of your time when you might "hack something together" from scratch by yourself, but that's usually just test-snippets for larger projects.)

IMPORTANT: There are 0 things in the next few pages you should skip. If you decide not to read the documentation, I have only one thing to say:

May the computer have mercy on you.

The Simulator

The folder called "`codeGoesHere`" is your "***code*** directory", the folder above this is your "***project*** directory". This project has a large code base where you will be inserting your contribution. The majority of this code is there to provide you with a visual display of your work once you are done. As it is, the code base can be compiled and run with the following commands from your ***CODE*** directory on Windows[1]:

```
javac -cp .;483libs.jar *.java
java -cp .;483libs.jar SimGUI
```

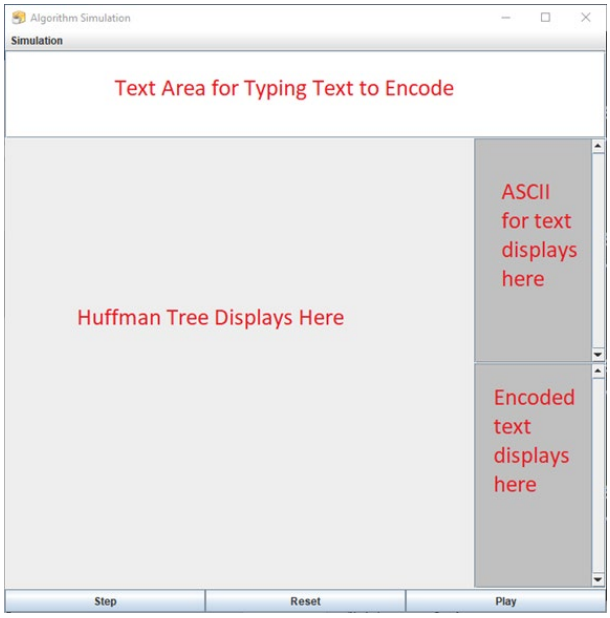
Or the following commands on MacOS/Linux:

```
javac -cp .:483libs.jar *.java
java -cp .:483libs.jar SimGUI
```

Why is there extra stuff in the command? The **`-cp`** is short for **`-classpath`** (meaning "where the class files can be found"). The **`.;483libs.jar`** or **`.:483libs.jar`** has the following components:

`.` the current directory
`;` or `:` the separator for Windows or Linux/MacOS respectively
`483libs.jar` the provided jar file which contains the library code for JUNG (more later)

If you compile and run, you should see the simulator open, even though you haven't written any code yet:



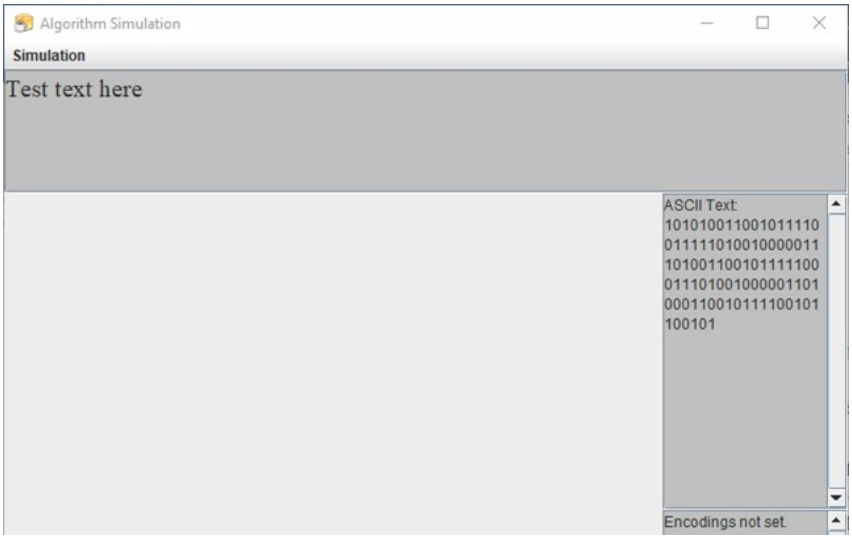
The "Text Area" at the top is where you can type some text to be encoded. The "Huffman Tree" area is where the you will see the tree-building visualization (but only after you write it!). The "ASCII" area shows what the text would look like encoded with simple ASCII. The "Encoded Text" area is for output from the encoding part of the project.

The buttons "step" (take one step of the algorithm), "play" (takes steps with a timed delay in between), and "reset" (start over) are along the bottom and should work if you click them.

---

[1] **Note:** There is a "commands.txt" file provided which contains all the commands for both Windows and Linux/Mac.

Out of the box, nothing will display in the "Huffman Tree" area or the "Encoded Text" area, but if you type something into the top text area and hit "play" you will see each character highlighted one at a time, and finally the ASCII text display at the end:



The following two commands will also work to run the simulator with more parameters:

```
java -cp .;483libs.jar SimGUI [text]
java -cp .;483libs.jar SimGUI [text] [ms-delay]
```

`[text]` can be any string, but you'll need to use quotes around the text if you have spaces, **`[ms-delay]`** can be any integer over 100 and controls the simulation drawing-delay (minimum 100ms). For example, if you want to have "PETER PIPER" pre-loaded and a 200ms animation speed, you'd use:

```
java -cp .;483libs.jar SimGUI "PETER PIPER" 200
```

On Mac/Linux, don't forget to replace the **`;`** with **`:`** in the above commands. Calling with no parameters gives defaults of the empty string and 800 to the above parameters respectively.

The JUNG Library and Provided Code Base

The JUNG library (Java Universal Network/Graph Framework, http://jung.sourceforge.net) provides a lot of cool visualization tools for graphs (and trees!), such as automatic layouts, animations, and much more. The remainder of the code base was written by me (Dr. Russell) utilizing the JUNG library heavily.

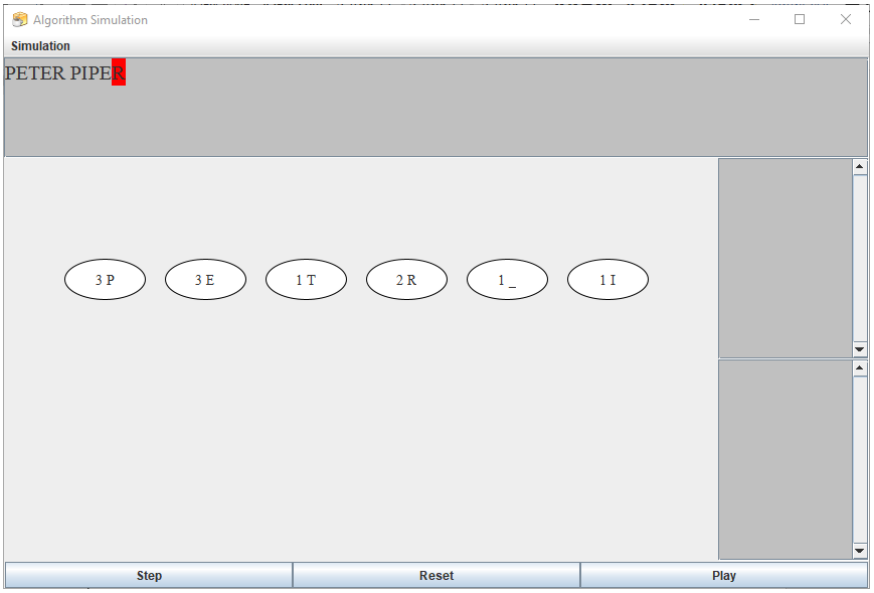
Below is an overview of the code base provided:

- **`SimGUI`** – The simulator itself.
- **`TreeNode`** – a general-purpose tree node class for the simulator
- **`HuffmanTreeNode`** – extends tree node class for Huffman-specific things (like counts)
- **`TreeEdge`** – a general-purpose tree edge class for the simulator
- **`FourEightyThreeForestAlg`** – an interface for CS483 forest algorithms
- **`HuffmanAlg`** – code to display the Huffman encoding tree building algorithm step-by-step
- **`OrderedDelegateForest`** – forest that returns tree nodes in a predictable order and labels edges between tree nodes based on that order (extends JUNG's **`DelegateForest`**)

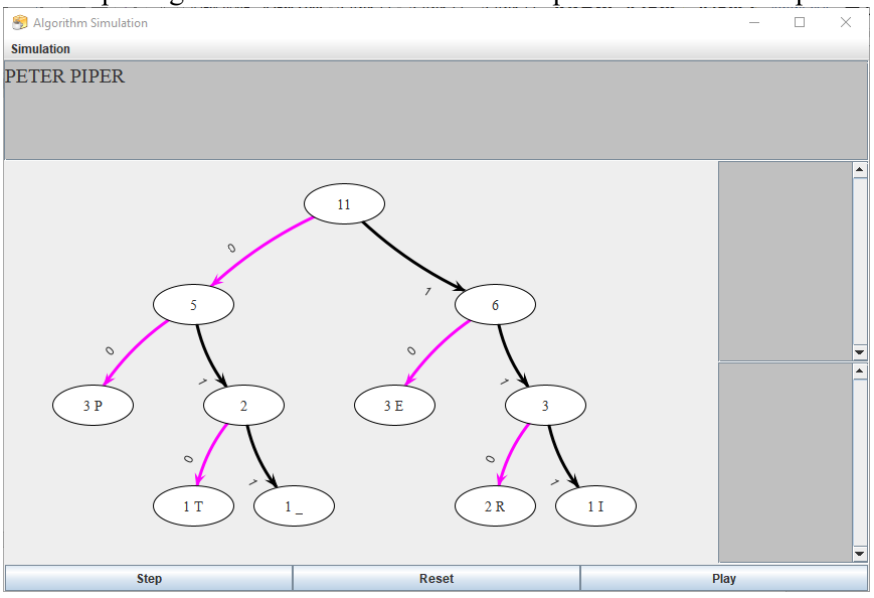
The Algorithm (Your Code)

Your task is to complete the **`step()`** method in **`HuffmanAlg`**. Each call to **`step()`** progresses the visualization forward one "step". **`HuffmanAlg`** has "**`YOUR CODE HERE`**" comments which indicate where you need to put your work, and there are detailed instructions in that file. However, below is a quick summary of the types of steps you need to do:

1. **Multiple Steps to Count the Characters** – Create nodes for every unique character in the input text and sets their counts. Example after *all* counting steps on "**`PETER PIPER`**":

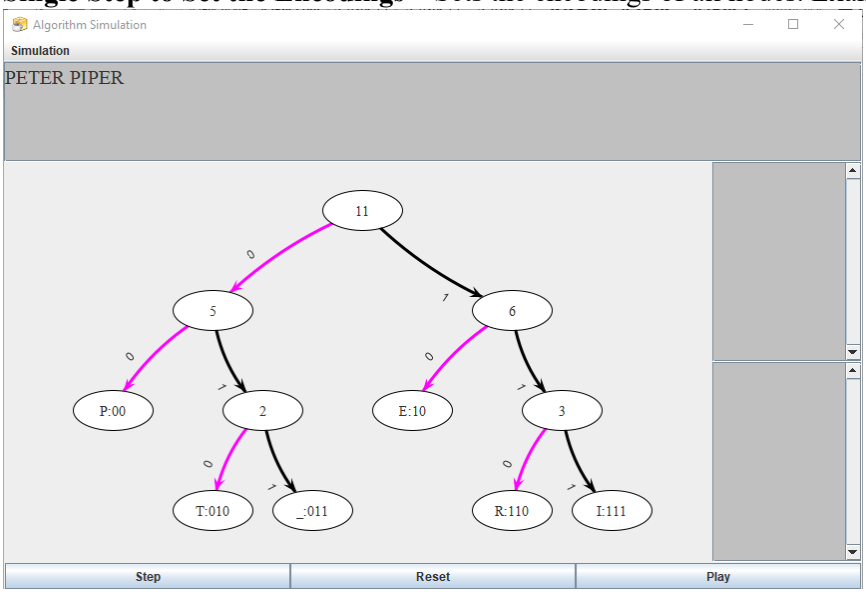


2. **Multiple Steps to Build the Tree** – Construct a Huffman tree using the (correct) greedy algorithm. Each step merges two trees in the forest. Example after *all* build steps:

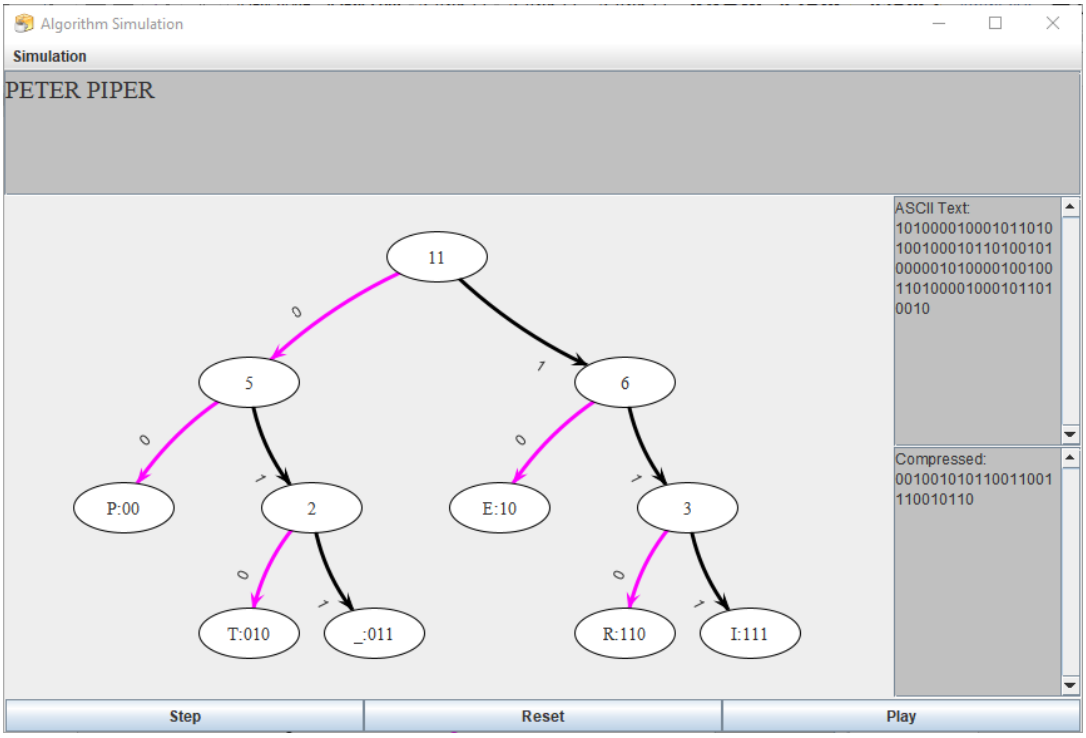


IMPORTANT: Read over the provided code and ***all*** the comments in **`HuffmanAlg`** ***before*** starting, there is a lot of work you are ***not*** doing. My implementation of everything you need to do up to here is ~24 lines of code + braces and comments.

3. **Single Step to Set the Encodings** – Sets the encodings of all nodes. Example after this step:



If you complete this stage, the next "step" of the algorithm will display the encoded text below the ASCII text in the simulator (you don't have to do this, it's automatic, you just set the encodings):



IMPORTANT: My implementation of this part is ~8 lines of code + braces and comments. If you think/design before you write the code, you'll probably have a much easier time.

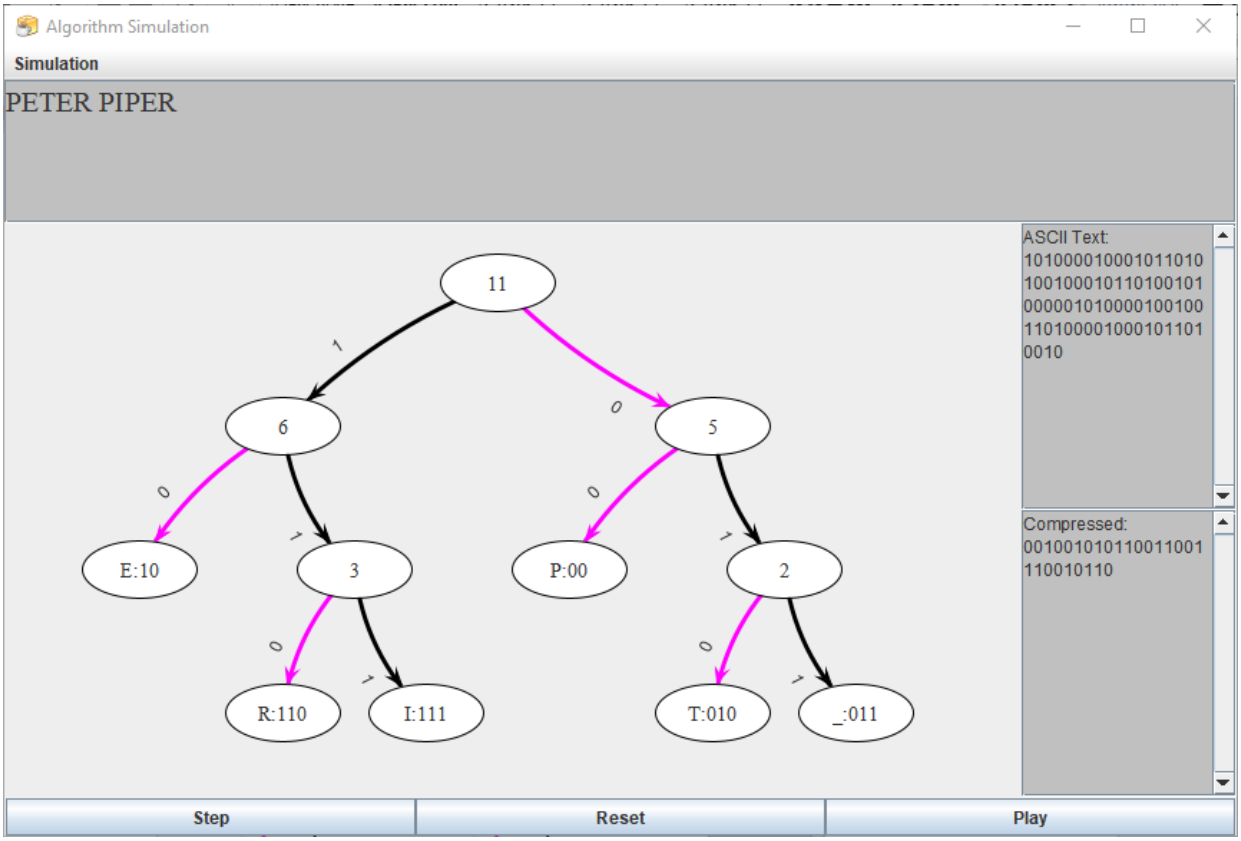
Important Behavior Information/Oddities

Below are a few items which may annoy/bother you, but are expected from this simulation:

- For visualization purposes, a node for the character space *displays* as _ (an underscore). The character *stored* in the node should still be a space, it's just "overridden" when displaying in the GUI (see **`HuffmanTreeNode:toString()`** and **`SimGUI`** lines 25-259). Underscores also display as _ and other whitespace characters (such as tabs) display as whitespace (so you can't see them). Just don't use text with underscores and non-space whitespace characters if this annoys you.

- If have not done the step to set encodings, there will be one "no action" step performed by the simulator before the algorithm displays the ASCII text. That is expected.

- As shown in the examples above, most of the time the 0-child will be displayed on the left and the 1-child on the right. These are the order the children are returned by **`OrderedDelegateForest`**. However, if you click the "step" button very fast or you set the animation speed higher, you may see some flipped edges where the "1" is on the left (as shown below). This is just a *visual* glitch; all the tree information should be identical if you are following the algorithm correctly (i.e. the encodings will be the same, the "1" edge will go to the same child node, etc.). So, while "annoying" it shouldn't interfere with anything.



Provided Testing Code

You can test your code with the GUI (obviously) but there is also some provided sample testing code for you, provided as both a main-method tester (**`MainMethodExampleTest.java`**) and as a JUnit tester (**`JUnitExample.java`**). These can help you verify before submission that your code is producing the correct output for the grader.

How to run the main method tester (from your ***CODE*** directory):

```
java -cp .;483libs.jar MainMethodExampleTest
```

This should produce "yays" if the simple tests are ok:

```
Character Counting Yay!
Tree Building Yay!
Encodings Yay!
```

How to compile and run the JUnit tests (from your ***PROJECT*** directory – above your code directory):

```
javac -cp .;codeGoesHere/483libs.jar;junit-4.11.jar;codeGoesHere *.java
java -cp .;codeGoesHere/483libs.jar;junit-4.11.jar;codeGoesHere JUnitExample
```

On Mac/Linux, don't forget to replace the `;` with `:`.

The output of a successful "everyone" JUnit run would look like this:

```
JUnit version 4.11
..........
Time: 0.095

OK (11 tests)
```

IMPORTANT: You need more tests than this. Feel free to add to the JUnit tests and the **`MainMethodExampleTest.java`** and as much as you'd like to do more thorough testing or invent your own JUnit tests. Test code is just code calling your code, you can do that!

Submission

Go to GradeScope and submit ***just the Java files from your CODE directory***. The GradeScope autograder will give you some immediate feedback to ensure you have submitted everything correctly:

- Cannot Find Java Files
  If your results say anything like:

  ```
  **********************
  Results
  **********************
  Cannot find java files Issue: cannot locate java files in
  submission folder
  ```

  Then something is wrong with your submission, and you should re-check what you've uploaded. Common mistakes: uploading a folder rather than just the Java files, uploading a zip, rar, or other compressed file, uploading .class files.

- Fail "Compile Alone"
  You are probably missing a file, make sure you submit ***all*** your Java files from your ***CODE*** directory. To make sure you've submitted your latest working version, upload all the Java files in the code directory, not just the **`HuffmanAlg`**.

- Fail "Compile With Tests (Preliminary Check)"
  Somehow you are not compiling with the JUnit tests. Check your own JUnit tests to see why this might be.

Grading Rubric

No credit will be given for non-submitted assignments, assignments submitted after the last submission date, non-compiling assignments, or non-independent work ("pairs" are still required to work independently of other students/pairs). Otherwise your score will be as follows:

For all students:

- **Full credit:** 4pts for passing the provided JUnit tests in **`JUnitExample.java`**, 6pts for passing additional JUnit grading tests.
- **Breakdown by task:** 3pts for counting characters (like provided JUnit tests 1-3), 5pts for building trees (like provided JUnit tests 4-10), and 2pts for encoding (like provided JUnit test 11).

For individual students:

- **Extra credit for early submissions:** 1% extra credit rewarded for every 24 hours your submission made before the last submission date (up to 5% extra credit). Your latest submission before the due time will be used for grading and extra credit checking. You CANNOT choose which one counts.

For teams:

- **NO extra credit option.** ***I don't care who is "responsible" for what part of the project. If you're working as a pair, you are both responsible.*** You may NOT start working as a pair and then split to try to get the extra credit. All cases of similar code being submitted by two students *not claiming to be on a team* will be considered an honor code violation.

5% will be subtracted if you don't submit correctly to GradeScope in format described in the submission instructions (because it will slow down grading for the entire class).

Common Questions / Problems / Concerns

- *Why are there so many weeks for this homework?*
    - This is a ***two***-week code homework assigned over a longer period. It's larger than a one week mini-code homework, but it is certainly not 3-4x bigger. It's maybe 1.5x or 2x depending on your comfort reading other people's code.

      So why are there so many weeks? Well, I don't know your mid-semester schedule: what classes you have midterms in, what you'd like to prioritize preparing for the midterm in this class, whether you want to (or don't want to) work during the break, etc. Since I do need you to finish another pencil-and-paper homework before taking your midterm, I've just given you the time you need to schedule *this* homework however best suits you.

      That said, please, for your own sanity, don't **unintentionally** ignore this project for several weeks before starting. Get as much done as early as you can.
- *What version of Java will you be testing with?*
    - JDK 17
- Do I have to write this in 32 lines of code?
    - Nope. My solution is around there (plus comments, whitespace, and braces), but I'd expect everyone's implementation to be different. I'm only telling you how long my code is to let you know that it ***can*** be done in that many lines. So, if you're writing hundreds of lines of code, something is probably wrong. Also, I did ***not*** try to write efficiently, so you may be able to do better than me!
- *I'm having trouble compiling out of the box*.
    - Check your Java version number (and your Javac version number). Java 9 or later should be fine. If you have an older version of Java it may complain about <> and anonymous classes.
    - Check that you're on the actual command line or terminal. Many IDE terminals are not the "real" terminals. For Windows, Powershell is not the same as the command line.
- *Can I add additional instance variables? Can I add additional non-static methods?*
    - Yep. That's something you can (and are encouraged to do) in **`HuffmanAlg`**.
- *Can I add additional static variables? Can I add additional static methods?*
    - Can... yes... but there's probably something wrong, so ask if you think you need to do this, and I will try to help you brainstorm other ways to do what you're trying to do.
- *Can I edit any of the files other than HuffmanAlg?*
    - Technically, yes, but those edits will only affect you. The graders will be using the **provided** versions of all files except **`HuffmanAlg`**. So, if you want to make the background green and default to 100ms animations, that's totally cool, but the TAs will be using what was provided to you, and if your code doesn't work with the provided files ... :(
- *Can I edit parts of HuffmanAlg that don't say "YOUR CODE HERE"?*
    - I can't stop you (technically) but you're very likely to produce non-working code if you do this. Ask if you think you **need** to and I will disabuse you of this notion.

- *Can I import additional code (from JUNG, from Java, etc.) in HuffmanAlg?*
    - **`java.util.*`**, JUNG's Forest Interface, and JUNG's Tree interface are already imported for you, and **`java.lang`** does not require an import. Please ask (on Piazza) if there is something you want to import other than these and the TAs will tag me. You shouldn't need anything that's not already imported, but I'll consider requests individually for this assignment. Importing non-approved code may result in a 0 if I believe it is being used to bypass the assignment goals (so ask!).
- *I've never worked with code written by others :(How do I even start looking at this?*
    - If you've read through the project description and don't know where to start, here is my advice:

      Identify the "structure" and "where to find things":

      The best way to start reading other people's code is to begin by getting a sense of the file structure (what files are there? what do they contain? etc.).

      At this point, you can try to locate the part where you need to do the editing. For this project, it's labelled, so that should help.

      Take a break + connect to what you're doing:

      Next, remind yourself what you're trying to do (in this case simulate the Huffman Tree building algorithm) and review how that should work on paper and pencil. This forced break will start to connect the code base to things you're working with in the actual algorithm.

      Identify the "tools" you can use to build your part:

      After your break, browse around the files you found in #1 and look for anything you might find useful. For this project, that would be things like "tree nodes", "tree edges", "Huffman tree nodes", etc.

      Look at any other documentation. For this project, that's JUNG forests and JUNG Trees. You're not trying to memorize anything, just look around for anything that seems useful.

      Next, look at the things that might be already done for you (such as in **`HuffmanAlg`**).

      Tiny, miniscule, edits:

      Now it's time to make some tiny edits and see what happens. This is a good way to "break the ice" before you start coding.

      You can print things, such as what mode you're in, how many times you've executed the **`doNextStep()`** method, etc. to get you started.

      The goal isn't to start solving the "big problem" it's to get you over the initial fear of "starting wrong" (the writer's block of coding).

      At this point you'll probably feel better. If not, stop by office hours!