**Due : December 13, 2016**

# Updates

- Updated the comment for the method *shortestPathLength* in Graph.java. You should return -1 if there are negative edges (since you cannot run Dijkstra in this case).
- Updated the comment for the method *compressionRatio* in HuffmanEncoder.java. The parameter $s$ will always be the decoded string.
- Updated the provided skeleton code for the *HuffmanEncoder*(String filePath) constructor. Changed $scan.hasNext()$ to $scan.hasNextLine()$ and $String\ s = scan.next()$; to $String\ s = scan.nextLine()$;. It now properly differentiates between spaces and newlines.
- Updated the comment for the *HuffmanEncoder*(String filePath) constructor. It now clarifies that there will never be trailing newlines.
- Added a new test file, "read_from_file_test.txt", so that you can check your implementation of the *HuffmanEncoder*(String filePath) constructor. The file "read_from_file_test_results.txt" contains a list of the frequencies for each character so that you can compare your output without having to manually count it. Note that characters like '!' that do not appear in the text also do not appear in the frequency table. Pay close attention to the frequency of spaces and newlines.
- Added *java.io.BufferedWriter* and *java.io.FileWriter* as imports to the HuffmanEncoder files.

# Overview

This assignment consists of two parts: implement a Huffman Encoder and implement an undirected, weighted graph.

# General Notes

- Do not change any method or class signature. You should only edit inside of the functions. If your code changes any class or method name or signature, you will receive an automatic 0. You should not implement any other function or instance variable besides the ones that are provided, unless explicitly allowed.
- If you are using Eclipse, be sure to remove the line declaring your code a package when you submit your code. Failure to do so could result in you receiving a 0 if the autograder fails.
- Make sure your code compiles. Non-compiling code will automatically receive a 0. If you have a problem that is causing you to not be able to compile, it may be better to just comment out the incorrect code and return a dummy value (something like null or -1) so the rest can compile.
- Make sure that your code does not print out anything (there should be no System.out.println in your code). You will receive an automatic 0 if your code outputs something to STDOUT during the tests.
- To ensure that your code will be accepted by the autograder, you should submit your code on YSCEC, download it again, unzip it, recompile it and check the provided test suite. This way, you know that the file you are submitting is the correct one.

# Huffman Encoding

Huffman Encoding is an optimal, prefix-free binary encoding scheme. In this homework, you will implement a Huffman Encoder with several features. For instantiation, your encoder can take in a pre-defined frequency table or the path to a file, from which it will infer a frequency table (meaning it should construct a frequency table from the characters appearing in the file). Once it has constructed a frequency table (and by extension, the encoding tree), it should be able to encode and decode text in both string and file form. Refer to the lecture for how to construct the encoding tree.

# Implementation Notes

- For this assignment, you can assume that no characters outside of the given alphabet will occur in a string to be encoded, and no characters besides 0 and 1 (and possibly new-lines, which should be skipped) will appear in a string to be decoded.
- In order to use a PriorityQueue on a defined class in Java, your class must implement *Comparable*, specifically it must have a compareTo method. We have done this for you for the HuffmanNode class and it should **not** be modified.
- We will not be grading the toString() method from this class, but you may still implement it in a way that helps you debug your code.
- We have provided some skeleton code for reading in from a file. Note that this "skips" newline characters, represented in Java as '\n'. A newline character appears in the text directly after the string returned by call to "scan.next();" since that method splits a string at newline characters by default. You can assume that the last line in a file will not have a trailing newline character. So, for example, a file with 4 lines of text will only have 3 newline characters, one at the end of each of the first 3 lines. When completing this homework, be sure to check that your code is properly counting the number of newlines (and other special characters).

# Directions

- Write your name and student ID number in the comment at the top of the HuffmanEncoder.java file. Please write your name in English letters, not 한글.

- Implement all of the required functions in the HuffmanEncoder class.
- Do not import anything other than what is provided. You may add private instance variables and methods to the Huffman Encoder class but do not change any preexisting method names. You should not modify the HuffmanNode class in any way.

# Graphs

The second part of this assignment will be implementing undirected, weighted graphs.

An undirected, weighted graph is a set of vertices, $V$, and a set of edges, $E = \{(u, v) : \text{s.t. } u, v \in V\}$. Each edge is also given a weight, which is some integer. Note that since this is an undirected graph, the edge $(u, v)$ is the same as the edge $(v, u)$.

In this homework you will be implementing code to construct undirected graphs as well as code to find the minimum spanning tree of the graph and shortest path between pairs of nodes. For the minimum spanning tree, you should implement Prim's algorithm. You should use Dijkstra's algorithm for the shortest path problem. Both of these can be found in the lecture notes.

# Implementation Notes

- If there is more than one minimum spanning tree, you can output any of them. You only need to find one, not all of them.
- You can assume that any graphs we test will be connected.
- You may represent the graph in any way you wish and use any of the imported data structures to implement the algorithms.

# Directions

- Write your name and student ID number in the comment at the top of the Graph.java file. Please write your name in English letters, not 한글.
- Implement all of the required functions in the Graph class.
- You are allowed to add any helper functions and variables that you would like, as long as you do not delete or modify any of the existing ones. We will only be testing the ones that are provided, so make sure your code runs through the correct methods.
- Pay careful attention to the required return types.

# Deliverables

**PLEASE READ CAREFULLY!**

Submit **only** the following files in a directory named PHW4. This directory should be zipped inside a zip file called PHW4.zip. Be sure your code compiles and does not include the "package" line so that it will work with the autograder. This is your responsibility, not the TA's, so we will not check or correct for this error. It is a good idea to submit your code to YSCEC, download it again, unzip it, and verify the contents (that it compiles, passes the tests, etc). That way you know you are submitting the correct version of your code.

- HuffmanEncoder.java
- Graph.java

To clarify, your file structure should look like PHW4.zip → PHW4 → HuffmanEncoder.java, Graph.java. It should **not** look like PHW4.zip → HuffmanEncoder.java, Graph.java. This means when you unzip the zip file, the first thing you should see is another directory named PHW4. This directory should contain HuffmanEncoder.java and Graph.java.

# Testing

As before, we have provided a small test suite for you to check your code. The directions for how to use it can be found on the YSCEC notice board. You are provided with the following files to help you test your code.

- HuffmanEncoderTest.java
- HuffmanEncoderTestRunner.java
- GraphTest.java
- GraphTestRunner.java

You will need to download the junit-4.10.jar file from the YSCEC Notice Board.
You should not include any of these files in your submission.
Note that the test suite we will use to grade your code will be much more rigorous than the one provided here (and not necessarily a superset of the provided tests). You should consider making your own test cases to check your code more thoroughly.