CIS 121—Data Structures and Algorithms with Java—Spring 2018

Homework 6—Compression

Due: Monday, March 12, 11:59pm online

2 Required Problems (45 points), Qualitative Questions (10 points), and Style and Tests (10 points), Written (35 points)

DO NOT modify methods or method headers that were already provided to you.
DO NOT add public or protected methods.
DO introduce helper methods and inner classes that are **package-private** (i.e. do not have a privacy modifier) as you deem necessary.

Motivation: Compression Algorithms

Today's computers do not exist in a black box. Machines are constantly communicating over the network, transmitting both raw application information and large files. We are familiar with the notion that faster programs are better. This also holds true over the network; high-latency networks affect everything from multiplayer video games to standard Internet access to mission- and life-critical applications.

Faster network access is always better. However, we cannot always make an Internet link faster (the networked equivalent of throwing more hardware at the problem). Therefore, we must use intelligent algorithms to reduce our load on the network. Some of these are built directly into the internals of our computers, like TCP congestion control. (Less is more, sometimes!) Others are optionally implemented at the application level, such as compression algorithms.

The goal of a compression algorithm is to take a sequence of bytes and transform it into a different sequence of fewer bytes, such that the original can be recovered. Because compression algorithms reduce the size of a file, they allow quicker transmission of files over the network, benefitting everyone on that link.

Compression algorithms can be lossless (like ZIP) or lossy (like JPEG). Lossy compression is useful for images, music, and video because multimedia is an emergent property of its file formats; in other words, we don't care if there are slight imperfections in a JPEG image, as long as we can still tell it's a LOLcat. On the other hand, text-based files must be compressed without any data loss; what good is a source code file if it cannot be compiled because a few of its bits switched from 1 to 0? We will focus on lossless compression.

In this assignment, you will implement several *lossless* compression algorithms that are used as parts of larger compression schemes, such as ZIP.

`Compressor` interface

We have provided a `Compressor` interface that represents any kind of (lossless) compression algorithm. The interface contains two methods, `compress`, which takes a plaintext and outputs a compressed version, and `decompress`, which reverses a compression.

NOTE: The `Compressor` and `AbstractHuffman` interfaces cannot be modified in any way. Failure to abide by this will prevent your code from compiling and will lose you credit on the assignment.

Part One: Huffman coding (20 points)

Files to submit: `Huffman.java`, `HuffmanTest.java`

The first compression algorithm is known as Huffman coding. The idea behind Huffman coding, and an idea common to a lot of concepts and techniques in computer science, is that common activities should have a lower cost than rarer activities. For example, we encode ASCII characters with eight bits each, but we see the character `e` much more often than the character `x`, which we see much more often than the `BEL` character (ASCII code 7).[1] Therefore, why not represent those more frequent characters with fewer bits and those rarer characters with more bits?

First, assume the existence of an alphabet, whose composing characters abide by some positive-valued probability mass function. In other words, each character in the alphabet has a probability of showing up, these probabilities are each greater than zero, and these probabilities sum to one. Your Huffman implementation will permit alphabet specification by either passing in a map from `char`s to `int`s or by providing a `String` from which the alphabet and probability mass function will be determined. Here, each `int` represents its corresponding `char`'s count, not its probability.

Why is this the case? We use `int`s instead of `double`s or `float`s because of the inherent imprecision of the floating-point standard; it is impossible to accurately represent a number as simple as $\frac{1}{3}$. Instead, we determine the alphabet's probability mass function by taking each `Character`'s corresponding count and dividing it by the sum of all `Character`s' counts.

Here is a (simple) example alphabet, according to *our* specification:

a	b	c
1	2	2

Here, the character `a` shows up $\frac{1}{5}$ of the time, and the characters `b` and `c` each show up $\frac{2}{5}$ of the time.

Huffman coding uses a binary tree to encode character information, where every leaf represents a character in the alphabet and where each edge represents the state of a bit in the compression. Starting from the tree's root, traveling to a node's left child indicates appending a `0`-valued bit to the compressed representation of a character, and traveling to a node's right child indicates appending a `1`-valued bit to the compressed representation of a character. Note that because of this, Huffman coding requires the alphabet to have no fewer than two characters.

The generation of the Huffman tree is the crux of the algorithm. Create a lone leaf node for each character in the alphabet (i.e. you should have n discrete leaf nodes). Add each node into a min-heap whose key is character frequency. While the min-heap has more than one node in it, follow the following process:

1. Remove the two nodes of lowest frequency from the min-heap.
2. Create a new node. Assign its left child to be the first removed node (the one with lower frequency) and its right child to be the second removed node (the one with relative higher frequency). **NOTE**: While this directional invariant is arbitrary, failure to abide by this standard will result in lost points, as our unit tests will expect this invariant.
3. Assign the new node's frequency to be the sum of its left child's frequency and its right child's frequency. This new node is an internal node and no longer corresponds to a character in the alphabet.
4. Add the new node to the min-heap.

When there is only one node left in the min-heap, it is our final Huffman tree.
Note: You may use `java.util.PriorityQueue` as the min-heap for this part of the assignment.

[1] http://www.asciitable.com

We have provided five method and constructor stubs for you. Do *NOT* modify the headers of these methods and constructors, only the bodies. Failure to abide by this will prevent your code from compiling, and you will lose credit on the assignment.

Of note regarding the implementation:

1. Decompression is straightforward; given a sequence of `0`s and `1`s, start from the root and go left or right as determined. When you get to a leaf, you've translated a character; record that, and go back to the root.

2. Compressing is less straightforward because you cannot use the `0`s and `1`s to find a character; you want to use the character to find the `0`s and `1`s.

3. Your `compress` and `decompress` methods should be as asymptotically efficient as possible. Part of this includes using the `StringBuilder` class to create your compressed and decompressed output instead of using `String` concatenation; the former can concatenate two `String`s in $\Theta(1)$, whereas the latter can only concatenate two `String`s in $\Theta(n)$. **We will stress test your code against massive inputs to make sure you used `StringBuilder`.**

4. The `expectedEncodingLength` method is nothing more than the expectation of a discrete random variable.[2] We are using this method not to test your knowledge of basic probability, but instead to test the optimality of the encoding that your algorithm generates.

5. You will need to use a private inner class for the Huffman tree. This class will need to implement the `Comparable<T>` interface so you can use a min-heap to store the incomplete Huffman trees. You are strongly encouraged to override the `toString` method to print a tree representation for ease of debugging.

6. Because we will be unit testing your output, you must implement your `compareTo` function of the Huffman tree with the following procedure for breaking frequency ties: if two leaf nodes have the same frequency, the node with the "smaller" character (use ASCII value to represent "smaller") should be the "smaller" node. If the two nodes being compared aren't both leaf nodes, the node that was created *first* should be the "smaller" node. (How might you go about implementing this?)

7. Our implementation uses `String`s of `0`s and `1`s to represent a stream of bits. This is convenient for a program written in Java, but it is not what would happen in the real world. If we were to implement this algorithm in production software, we would post-process each compressed answer by converting each character into its corresponding bit.

8. We will compute a compression ratio that quantifies how well our compression performs. It should return the average compression ratio for all strings you have compressed over the lifetime of the object instance. We define compression ratio as the length of all outputs from compression over the length of all inputs to the compression. This length should be calculated in terms of "bits". We recognize that our output, although a binary string, isn't actually a sequence of bits. However, you can pretend that it is just some binary representation of the original input. You do not have to explicitly turn the input into binary, but treat the "length" as the number of bits. You can treat the input length as the number of characters times the size of a Java char, which is 16 bits. The ratio is maintained and modified as you continue compressing strings using the same Huffman instance, since it is an aggregate ratio.

[2] Wikipedia

Part Two: LZW Compression (25 points)

Files to submit: `LZWCompressor.java`, `LZWCompressorTest.java`

The next compression algorithm to implement is the Lempel-Ziv-Welch compression scheme. You may notice that the Huffman scheme requires use of a reasonable seed string in order to get good results. We can move away from this by using an adaptive model which learns and updates the compression model as text is read in. This allows you to do one-pass compression on a string, and the adaptive nature of the model should produce better compression.

We will follow similar assumptions to Huffman encoding. We will assume the existence of an alphabet, which is a set of allowed chars in the compression. We also assume a codeword length w in order to put a limit to the number of entries we will put in our symbol table. The values in the table should correspond to w-bit binary strings, hence how we know there should be a maximum of 2^w key-value pairs in the table.

LZW compression uses a symbol table to store compression codes. Keys are substrings that you want to compress, and the values are the compression codes that you will output. The idea is to build up the symbol table to better compress substrings that you have already seen. For example, compressing "NANANANA BATMAN!!!! NANANANA BATMANNNN!!!" would be good because there are many repeating substrings that can be better compressed each time they appear. The symbol table learns throughout the string.

The generation of the underlying symbol table is the crux of the algorithm. There are multiple data structures you may choose from to implement the symbol table, and you should consider the tradeoffs of each of them. Initialize the symbol table with the correct size depending on the length of the codeword. Add codewords for all of the single characters included in the alphabet. In this initialization, please put the characters **in sorted order**. Notice that `char`s are Comparable, so you can take advantage of their built-in ordering. Process your string input with this process:

1. Find the longest string in the symbol table that is a prefix of the unscanned part of input. This is another way of formulating the longest prefix match problem.
2. Output the codeword (value) that was stored in that location in the symbol table.
3. If the number of entries in the table is less than 2^w, then add the prefix + next character to the symbol table. If the table is full, do nothing.

The stub file for this portion is very similar to the Huffman file, except you do not have to implement the expected encoding length function since that doesn't apply to this algorithm.

Here is an example to illustrate the input and outputs to this compression algorithm:

Create my compressor with `codeLength` of 3 and `alphabet` as the set of $[a, b, n, d]$.

The input "banana" should output "001000011101000", which is 15 bits long.

The state of my symbol table should look something like this:

a	000
b	001
d	010
n	011
ba	100
an	101
na	110
ana	111

The compression ratio will be $\frac{15}{length(banana)*16} = \frac{15}{96} = .156$.

DO NOT save the symbol table you build inside of `compress` in your class.
DO NOT try to decompress by using a saved symbol table (you must rebuild it!).

The benefit of using this method is that you can decompress by reconstructing the symbol table. This is useful in real life since you no longer have to send a seed along with the encoding to have another party be able to decompress it. Other than this, you are free to set up any inner classes or instance variables, helper methods that you want.

Part Three: Conceptual Questions (10 points)

Files to submit: `questions.txt`

Please answer the following questions in a text file called `questions.txt`:

1. Analyze the running time of generating the full Huffman tree for standard Huffman coding.
2. Analyze the running time and space usage of your LZW algorithm. Specifically, think about the size of the symbol table. Could you use another data structure to improve it? If not, say why. State any assumptions you make about the inputs and parameters for the data structures.
3. Compare the compression ratios between Huffman and LZW for a variety of strings. Can you make Huffman perform better than LZW on a specific string? If yes, state the string and parameters used and why they worked. If not, state why this is the case.
4. When might it be better to use Huffman coding instead of LZW Compression coding? Why? (Please give a specific example scenario or use case that is not strictly data dependent like in the above question.)
5. The size of the fixed code length, and therefore the size of the symbol table, has implications on the performance of LZW. If you increase the size of the symbol table, what effect will it have on runtime, space usage, and compression ratio? (Feel free to provide any data you test on) State a scenario where you might want a small symbol table, and a scenario where you would use a large symbol table.

Style & Tests (10 points)

The above parts together are worth a total of 55 points. The remaining 10 points are awarded for code style, documentation and sensible tests. Style is worth 5 of the 10 points, and you will be graded according to the 121 style guide.

You will need to write comprehensive test cases for each of the methods you implement. Make sure you consider edge cases, i.e. the more "interesting" inputs and situations. Use multiple methods instead of cramming a bunch of asserts into a single test method. Be sure to demonstrate that you considered "bad" inputs such as null inputs. Be sure to also demonstrate that you've tested for inputs to methods that should throw exceptions! Your test cases will be graded for code coverage and are worth 5 of the 10 points. You will have to thoroughly test your code to get full points! This includes testing any helper methods you have written. **Note**: you will not be able to write JUnit test cases for any private methods. Instead, make them package-private by leaving off any privacy modifier (i.e. do not write public or private).

Finally, we encourage you to work out small examples of Huffman and LZW coding by hand in order to write unit tests. We have not included sample test cases.