Homework 7—Compression

Due: Friday, March 4, 10:30am online

3 Required Problems (75 points), Qualitative Questions (10 points), and Style and Tests (15 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). 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 chars to ints 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 ints instead of doubles or floats 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 Characters’ counts.

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

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

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:** Once a Huffman tree has been fully generated, compression and decompression are both *idempotent*. This means that encoding and decoding do not have side effects on the world that affect their results. Much like how $2+15$ always equals 17, no matter how many times we perform it, calling `huff.compress(str)` for some Huffman object `huff` and some String `str` will always return the same answer, no matter how many times we perform it. This will *not* be the case for part 3 of the assignment, adaptive Huffman.

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 0s and 1s, 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 0s and 1s to find a character; you want to use the character to find the 0s and 1s.

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 `Strings` in $\Theta(1)$, whereas the latter can only concatenate two `Strings` 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. 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 Strings of 0s and 1s 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 the 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 the output although a binary string isn’t actually number of bits, but you can pretend that it is just some binary representation of the original input. You do not have to explicitly turn the input into

---

2 Wikipedia
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.

**Part Two: k-ary Huffman coding (25 points)**

Files to submit: KHuffman.java, KHuffmanTest.java

Now that you’ve implemented binary Huffman coding, implement k-ary Huffman coding, where \( k \geq 2 \). k-ary Huffman is relevant for mediums of transmission that have more than just “high” (1) and “low” (0) voltage state. For example, setting \( k = 3 \) would represent encoding state in “trits” (2, 1, and 0).

Notice that at each iteration of the algorithm, you are removing the \( k \) nodes of lowest frequency from the min-heap and adding a new node in, thereby reducing the size of the min-heap by \( k - 1 \) nodes each iteration. Consequently, when \( k > 2 \), we may not have enough nodes to create a full tree. To avoid this, you must insert zero-probability placeholder nodes (that don’t correspond to any characters) into your min-heap at the beginning of the algorithm. How many zero-probability placeholder nodes must we insert? It is up to you to determine this. As a hint, consider cases for when \( k = 2, m < k \), and \( m \geq k \) (where \( m \) is the size of the alphabet).

\( k \) can range from 2 to 36. Your encoding characters should be 0, 1, ..., 9, a, b, ..., y, z.

You should maintain a similar directional invariant that you did in standard Huffman coding: that is, for any node, its leftmost child should have the lowest frequency, followed by its next-leftmost child having the second-lowest frequency, etc. This should be a natural extension of moving from trees with two children to trees with \( k \) children. You should also use the same `compareTo` invariant to break frequency ties that you did on standard Huffman coding.

**Part Three: Adaptive Huffman coding (30 points)**

Files to submit: AdaptiveHuffman.java, AdaptiveHuffmanTest.java

Now that we understand the basic idea of Huffman coding, we can change the algorithm to learn new frequencies and new characters. Jeffrey Vitter of Duke\(^3\) developed an algorithm to do just this.

First, let us define a few additional terms and concepts. A **block** is a collection of nodes in a Huffman tree that all have the same frequency. We will want to assign each node a decreasing **ID** at the time of creation. Within a block, the node of highest **priority** is the node with the highest ID. Finally, we will use an **NYT node**, or **not yet transmitted** node, to catch new additions to the alphabet.

Adaptive Huffman does not take an input alphabet or input set of frequencies. Therefore, the very first character is encoded solely by its eight-bit ASCII code in binary. After that, characters in the tree are transmitted by their tree-based encoding, as done in standard Huffman. Characters **not** in the tree are transmitted by the tree-based encoding of the NYT node **and** the new character’s eight-bit ASCII code in binary.

The sender will compress each character by the above paragraph, **after which** it will perform a tree update (after each character is compressed). The receiver will decompress each character by the above paragraph, **after which** it will perform a tree update (after each character is decompressed). Therefore, assuming that the receiver decompresses all messages in the order that the sender compressed them, the sender and receiver’s Huffman trees will always be synchronized. In this light, the added complexity of adaptive Huffman is already visible compared to standard Huffman.

---

\(^3\) http://www.cs.duke.edu/csed/curious/compression/adaptivehuff.html
When writing your adaptive Huffman implementation, please make sure you can do multiple compressions and decompressions without losing internal state. Internal state should not reset after calling compress or decompress; that would defeat the purpose of learning from the input fed in.

Tree updates are the hard part of adaptive Huffman coding. Let this be fair warning: adaptive Huffman tree updates are trickier pointer manipulation algorithms. Your best bet on implementation would be to study the algorithm and then write it out in very high-level pseudocode, where you use as many helper methods as possible. Then, write the helper methods. Our internal solution is only about twenty lines of code (disregarding comments) because we have delegated functionality to many helper methods. If you are writing significantly more lines in your tree update function, you need to break your function down into more helper methods. This is a complex algorithm that code readers will not be able to understand if it is not broken down as such.

1. A tree update indicates that we have encoded or decoded a specific character in the alphabet. If this character is new (i.e. not in the tree), go to the NYT node and create two children. Its left child should be a new NYT leaf node (the original NYT node is no longer NYT, but rather a standard internal node). Its right child should be a leaf node for the new character. Increase the weight of the new character node and the old NYT node. The current node should still be the old NYT node. Jump to step 5.

2. If the character has already been seen, find it in the tree.

3. Does the current node have priority in the block? If not, swap the associated character and children of the current node and in its block with priority. Do NOT swap the IDs. Do NOT perform this swap if the node with highest priority (highest ID) is the current node’s parent. The current node pointer should point to its new position, not its old position.

4. Increment the frequency of the current node (whether or not a swap occurred).

5. If the current node is the root, we’re done with the tree update. Otherwise, set the current node to the current node’s parent, and jump to step 3.

Why is step 3 so complicated? During the process of a tree update, the tree may violate the sibling property, which is that parents have a higher frequency than their children. The node swapping up the tree ensures that at the end of the update procedure, the tree is in a valid state.

We have provided four 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.

Because the adaptive Huffman coding algorithm performs a tree update on both compression and decompression, you will not be able to use just one AdaptiveHuffman object when testing your code. That is because in practice, this algorithm would run across the network. When testing, you will want one AdaptiveHuffman object for a “client” computer, and one for a “server” computer. The client and server can communicate, so both can compress and decompress. However, it is critical that the server decompresses each client message in order, and vice versa. Otherwise, you will see incorrect results.

While we encourage you to work out your own test cases by hand, it will likely help you understand the algorithm by working through a known example step-by-step. A very thorough example is available [here]. Do note that this example doesn’t use ASCII for new character transmission, but rather some proprietary format. You’ll need to adjust accordingly.

Once you’ve finished this problem, you’ll have written a compression scheme that remains optimal, even if character frequencies fundamentally change. That’s pretty cool!
Part Four: 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 of generating the full Huffman tree for k-ary Huffman coding.

3. What do you think is a real-world benefit of an idempotent algorithm? (Hint: are messages sent over the Internet always reliably delivered?)

4. When might it be better to use Huffman coding instead of adaptive Huffman coding? Why? (Please give a specific example scenario or use case.)

Style & Tests (15 points)

The above parts together are worth a total of 85 points. The remaining 15 points are awarded for code style, documentation and sensible tests. Style is worth 5 of the 15 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 manually graded by the TAs, and are worth 10 of the 15 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, k-ary Huffman, and Adaptive Huffman coding by hand in order to write unit tests. We have not included sample test cases.