Assignment 1 – Huffman Coding

For this assignment, you will write a decoder and corresponding encoder in Java capable of compressing arbitrary input data using Huffman codes. To test your decoder, an already-compressed binary file is provided. Your decoder algorithm should be able to correctly decompress the given binary file to produce an uncompressed text file containing English text. Your encoder should be able to take any raw text file and produce a compressed version that is decodable by your decoder. More detailed specifications for the compressed format, encoder algorithm, and decoder algorithm are given in the sections below.

One obstacle faced by programmers of data compression algorithms is that operating systems and CPUs typically restrict I/O operations to only allow manipulation of files in byte-sized increments. Variable length codes, including Huffman codes, are often not exactly 8 bits long and therefore often do not line up on a byte boundary. To deal with this problem, the starter code for this assignment includes prewritten classes that wrap Java's built-in `InputStream` and `OutputStream` interfaces, enabling bit-sized read and write functionality by buffering bits in memory until enough bits are accumulated to perform a byte operation on the underlying stream. Use the `read()` method provided by the `InputStreamBitSource` class to read individual bits from an `InputStream`, and use the `write()` method from the `OutputStreamBitSink` class to write bits to a `OutputStream`.

The starter code for this assignment can be found at the following GitHub repository: https://github.com/onsmith/comp590sp19-a1

When you are finished with this assignment, upload your encoder/decoder code to a public GitHub repository. Then, submit the following information as text to the Sakai assignment:

1. Answers to the questions in Part 3 of the assignment.
2. A link to the public GitHub repository hosting your code.

Part 1 – Decoding Huffman Codes

The first part of this assignment is to write a Huffman decoding algorithm. This algorithm should be given a filename pointing to a compressed, binary file—for example, the binary file provided with the assignment.

The data within a compressed file represents a sequence of Huffman-coded bytes. Each of the 256 unique 8-bit patterns (i.e. `00000000`, `00000001`, `00000010`, …, `11111111`) that a byte may represent is assigned its own prefix-free Huffman code. Thus, there are 256 different Huffman codewords referring to the 256 unique byte symbols.

In the compressed file, bytes are represented by their corresponding Huffman codeword. The first step to decoding the file is therefore to construct a table which maps the Huffman codewords to the bytes they represent. This mapping is signaled at the very beginning of the compressed file using the canonical Huffman tree method described in class (see book for details). The 256 lengths of the 256 Huffman codewords are therefore signaled at the beginning of the compressed file (this is described more precisely below). Knowing these code lengths, one can construct the canonical Huffman tree using the deterministic algorithm from class. The canonical tree then provides the mapping between the codewords used in the compressed file and their corresponding byte symbols.

In summary, your decoder should assume that the compressed file is split into the following three sections.

1. *First 256 bytes* – The first 256 bytes of the compressed file represent the lengths of the codewords for each symbol. For example, the very first byte read from the file represents the length of the Huffman codeword for output symbol `00000000`. The second byte represents the length of the codeword for symbol `00000001` and so on. The 256th byte represents the length of the codeword for symbol `11111111`.
2. *Next 4 bytes* – Directly following the 256 code lengths, the subsequent four raw bytes represent an `int` value corresponding to the total number of symbols coded in the file (not including the first 256 + 4 = 260 header bytes). This value can be used to determine when to terminate your decoding loop.
3. *The rest of the file* – After this point, all subsequent bits in the file represent prefix-free Huffman codewords as defined by the canonical Huffman tree. If the very last codeword at the end of the file does not fall on a byte boundary, then the file is padded with `0`s accordingly so that it does fall on a byte boundary.

The example compressed file provided with this assignment represents ASCII English text. Therefore, after successful decoding, you should be able to open it with a text editor to see English words. If you see gibberish, it is likely that something in your canonical Huffman tree construction or decoding loop is incorrect.

Part 2 – Encoding Huffman Codes

The next part of the assignment is to write an encoder capable of compressing raw input files and producing encoded files that can be decoded by the decoder from Part 1. Thus, your encoder should output a file with the same structure as in Part 1; the first 256 bytes written should represent the codeword lengths in canonical tree form, the next 4 bytes should represent the number of encoded symbols, and the rest of the bytes should be filled with codewords.

The encoding process is essentially the same as the decoding process, only in reverse – instead of reading bytes, you are writing them. Instead of transforming Huffman codewords into bytes, you will be transforming bytes into Huffman codewords.

When you sit down to start writing your encoder, you'll notice that the first thing you have to output to the encoded file is a list of 256 codeword lengths. While one possibility is to simply reuse the same codeword lengths (and therefore the same codewords) from the encoded file given with this assignment, **you should not do this**. Instead, your encoder should generate optimal Huffman codeword lengths for the specific content that is being encoded, according to the symbol frequencies present in the input data. To do this, you will need to scan the entire source file and track how many of each symbol is encountered. These values can be used to calculate the average probabilities of each symbol across the entire input file. Next, use this information to construct an **optimal, minimum variance Huffman tree** according to the algorithm described in class (and also explained in the book). This tree will provide the correct lengths for each codeword according to their probabilities in the source input. Finally, throw away the generated Huffman tree and use the codeword lengths alone to re-generate a new Huffman tree that represents the canonical tree for those lengths. You should be able to re-use some code from Part 1 to do this.

Part 3 – Entropy Calculation

For the last part of the assignment, perform the following steps and answer the following questions.

1. Use your decoder to decode the provided compressed file, producing ASCII English text.
2. Calculate the probability of each symbol in the source ASCII English text file by dividing the number of occurrences for each symbol by the total symbol count.
3. Calculate the theoretical entropy of the source message in bits per symbol using the symbol probabilities from part 2. What is this value?
4. What is the compressed entropy achieved by the provided compressed file in bits per symbol? (Note: Do not include the overhead incurred by the 260 header bytes.)
5. Use your encoder to re-encode the raw English text.
6. What is the compressed entropy achieved by the Huffman code produced by your encoder?
7. Using your entropy value calculations as evidence, does your encoder achieve better compression or worse compression then the original compressed file?