Project 1
Huffman coding
TA : Sanketh Indarapu

1 Objective
Given a frequency distribution of symbols, the Huffman algorithm assigns codes to each symbol so as to minimize the size of the message. The message is then encoded using this symbol-to-code mapping and transmitted to the receiver. For the receiver to decode the encoded message, it is necessary for the receiver to have some knowledge of the symbol-to-code mapping. This information is generally transmitted as the header part of the encoded file. In this project, you will implement techniques to minimize the size of this header.

2 Description
In this section, we informally describe the details involved in the implementation. Later, we will actually lay down the specifications of the project.

Before reading further, you are advised to first refer to the class notes for Huffman encoding. Understanding those portions is crucial to what follows.

Consider different ways of informing the receiver of the symbol-to-code mapping. Most common implementations transmit the frequency distribution of the symbols in the header part. The idea behind this approach is that the receiver can then build the Huffman tree and thereby, learn the codes for the symbols. However, this approach needs additional guarantees to succeed. For a given frequency distribution, different Huffman trees are possible - all of them are optimal in the sense that they have the same codeword lengths for all the symbols, but the actual codes for the symbols differ. For the sender and the receiver to build the Huffman tree, there must be a guarantee that both of them build the same Huffman tree. This is generally guaranteed in practice by specifying that both the algorithms break ties in the same way - that is, they handle cases when there are equal contenders for choices in the same way.

Note that this frequency information can be possibly large, because there is one count per symbol. The count could possibly be a large value. Nevertheless, this is what is used in practice because, in general, the size of the data far overwhelms the size of the header. However, in this project, we will try and optimize this header size, mainly because it is instructive to do so.

2.1 The first optimization
The first way to try and improve on the simplistic approach outlined before is to somehow pass a description of the tree to the receiver (Another option is to list the codes corresponding to the symbols, but it can be checked that this too takes $O(|A| \cdot \text{max\_depth})$, where $A$ is the alphabet to which the symbols belong). Before proceeding further, let us simplify things by assuming that we shall be operating over the 256 ASCII characters from 0 to 255.

Consider the issue of what is the essential information in a Huffman tree. The essential information in the tree is the codeword lengths of the symbols. Any other tree that satisfies the prefix code requirement and has the same codeword lengths is equivalent to the Huffman tree for the purposes of compression. We call any such tree an optimal tree. One way of transmitting the header information with very little information
is to send just the codeword lengths of the symbols over to the receiver. The receiver and the sender should then agree on a common optimal tree to build from the codeword length information to determining the actual codes. Then, they both will have the same codewords for all the symbols, thus achieving the original purpose. Note that this approach differs from the one described in the previous subsection in agreeing on the definition of a common optimal tree, whereas the previous approach agrees on a common algorithm to build the optimal tree. A quick order estimate shows that this would cost about $O(|A| \ast \log(\text{max_depth}))$, that is, in our case, $256\ast 8$ bits. This is especially neat, because we would then be using a byte for each codeword length value (which can be anywhere between 1 and 255 in the worst cases).

The approach of the previous paragraph therefore seems promising. The sender would send the receiver the codeword lengths as a list of 256 values. Then both would construct a canonical optimal tree (that we shall call the canonical optimal tree now on) and use this tree for the encoding and the decoding purposes. Note that this seems to imply that the sender constructs two trees - one for finding the codeword lengths themselves and one for the canonical optimal tree. However, this is not necessarily the case and the sender could execute a minimal version of the Huffman algorithm that does not involve building the tree, but instead relies on simpler data structures.

2.2 The canonical optimal tree

The canonical optimal tree for a given set of codeword lengths is a full binary tree with leaves corresponding to the symbols and with the depths of the leaves corresponding to the codeword lengths and also satisfying the prefix code property. Before defining the canonical optimal tree, we define a function \textit{in-order} which takes such a tree as input and returns the string formed by concatenating the symbols at the leaves of the tree visited in an in-order traversal of the tree. The canonical optimal tree for a given set of depths is defined as the tree with the lexicographically smallest value of the in-order function.

To summarize, the header consists of 256 codeword lengths, one byte per value, and there is agreement between the sender and the receiver on the canonical optimal tree.

2.3 Further optimization

Is there any way in which we can further optimize the size of the header information? It is helpful to consider the typical case in such situations.

In a typical execution of the Huffman encoding algorithm, for example, over a file where the frequencies of the symbols are almost the same, or where the frequencies of many symbols are close, the codeword lengths are not going to take all the values from 0 to 255. It is more likely that the codeword lengths are chosen from a small number of values. Suppose, for example, that the codeword lengths are mostly 7, 8 and 9. Then, the list of codeword lengths in the header would look like \{ 7 7 8 9 9 8 7 7 7 7 9 9 9 8 8 8 9 9 9 7 8 \ldots \}. This sequence seems like the ideal candidate for further compression. And while we are on the topic of Huffman compression, we can as well try and use the Huffman encoding to compress this sequence of values.

Thus, the sender, instead of sending over a list of 256 bytes, is now going to send the Huffman encoding of this list. This seems fine, except for one detail. It is necessary to send a header in this case too, to specify the symbol-code mapping for this second-level Huffman encoding (we shall use the term second-level encoding to refer to this compression). We could again try and send the header information using the approaches we are using. But this it is not likely to save that many bits at this level, when we are just compressing 256 bytes of data. So, for sending over the second-level Huffman encoding header information, we shall use the (inefficient) approach we mentioned in the very beginning. We shall just send over the frequency count of the second-level symbols, which are the codeword lengths. In this case, the symbols refer to the codeword lengths and so, the frequency distribution of the lengths is the number of codewords with a particular length \( i \), for all \( i \). The lengths can possibly be anywhere between 1 and 255 (with 0 being used to denote the length of a symbol which is not in the tree) and so, we would need to send, 256 values for the count of the number of codewords with length as this value.

It is useful to rephrase the whole discussion in terms of the Huffman tree. Codeword lengths correspond to depths of leaves in the tree and the frequency information we are sending over is basically the number of
leaves at depth \( i \).

At this stage, therefore, we are going to send, as part of the header, two sets of data. First, the frequency distribution of the codeword lengths, that is, the number of codewords with a given codeword length. There are going to be 256 bytes for this set (with one byte for each value). The next set is the encoding of the 256 codeword lengths of the 256 symbols using the canonical optimal tree obtained from the frequency distribution of the first set. These two sets complete the header portion.

2.4 The last optimization

The approach of the previous section can be further improved by realizing that the first part of the header itself is going to be fairly redundant and compressible. At this stage, instead of using any of the standard compression techniques like the Huffman encoding again, we specify a compression algorithm of our own, that performs well on such data. The pseudocode for this algorithm is specified below. You are encouraged to try and understand the idea of this algorithm before implementing it. In what follows, the list of frequencies of the codewords is denoted by \( A \).

We now describe a method for encoding an array \( A \) of 256 nonnegative integers summing to 256. The method achieves particularly good compression when the following conditions hold: (1) the highest index \( i \) for which \( A[i] \neq 0 \) is relatively low; (2) \( A[i + 1] \) is similar in magnitude to \( A[i] \) for most values of \( i \). Conditions (1) and (2) are typically satisfied in the application of interest in this project, i.e., where \( A[i] \) denotes the number of nodes of codewords of length \( i \) in a Huffman tree with at most 256 leaves. (Note: In this application, \( A[0] \) plays a special role. It represents the number of zero-frequency symbols, i.e., symbols for which no codeword has been assigned. Thus the number of leaves in the Huffman tree is 256 – \( A[0] \).)

Below we provide pseudocode for the encoder. The pseudocode makes use of a couple of auxiliary definitions. Let \( s(0) \) denote the empty string and for \( i > 0 \) let \( s(i) \) denote the binary representation of \( i \) (with no leading zeros). For example, \( s(4) \) is equal to the binary string 100. For any nonnegative integer \( i \), let \( w(i) \) denote the length of the string \( s(i) \). For any \( i > 0 \), let \( t(i) \) denote \( s(i) \) with the leading bit (which is a 1) switched to a zero. For example, \( t(4) \) is equal to 000.

```java
for (int a=0, b=0, i=0; a<256; a+=A[i], b=w(A[i]), i++)
    if (w(A[i])>b)
        transmit w(A[i])-b 1's followed by t(A[i]);
    else
        transmit b-w(A[i])+1 0's followed by s(A[i]);
```


```
01100101100111000111110110000
```

2.5 Encoding and decoding of the input data

The actual encoding and decoding are fairly simple. The encoding process is simply replacing the bytes of the input data stream by the corresponding codewords. The decoding process consists of reading bits from the stream till a codeword is identified and then writing the corresponding symbol to the output stream.

3 Specification

In order to help organize the coding involved in this project, we now lay down the interfaces between the different parts of the assignment and the classes in which the code is supposed to be written.

Most of the data in this project is handled as a stream of bits. There is no corresponding abstraction in the common programming languages. We therefore, provide a class called “BitStream” to help support this
abstraction. This class supports the reading and writing of bits and bytes and should serve as the medium of exchange of data across interfaces.

The BitStream class has the following specifications -

**BitStream()** - the null constructor.

**BitStream(byte [] b)** - initializes the bit stream to the bytes in the array b.

**BitStream(String inputFileName, boolean type)** - initializes the bit stream from a file. The type field determines the type of the file. If it is true, the file is assumed to be a byte file and is read the usual way. If the type field is false, the file is assumed to be a bit file written earlier using the toBitFile method.

**boolean readBit()** - reads one bit from the stream. The return value is the truth value of the next bit in the stream. It throws java.io.EOFException if the stream is empty or if there are problems associated with it. You are advised to check the stream using isEmpty() before each read to recognize EOF instead of relying on this exception.

**byte readByte()** - reads one byte from the stream. This method too throws java.io.EOFException on empty stream or if there are problems with the stream.

**boolean isEmpty()** - returns true if the stream is empty.

**int writeBit(boolean bit)** - writes the specified bit to the stream. It returns non-zero on success and 0 on failure.

**int writeByte(byte b)** - writes the specified byte to the stream and returns the success status as above.

**int toBitFile(String outputFileName)** - writes the bits in the stream to a file. The file is specially marked to handle the conversion from bits to bytes. It returns 0 if the operation fails.

**int toByteFile(String outputFileName)** - writes this stream to a file as if it were a byte stream. Any extra bits that do not form a complete byte will be discarded. It returns 0 if the operation fails.

**byte [] toBytes()** - converts the stream to an array of bytes. Any extra bits that do not form a complete byte will be discarded.

There should be one main class called “CanonicalTree”. This class encapsulates the canonical optimal tree described in Section 2.2. Note that while in reality, the sender and the receiver would be using two different copies of the CanonicalTree class and agreeing on the specification of the tree, in this project, we have only one class for both these abstractions. **However, your code for the decoder (resp. encoder) should work correctly with any other encoder (resp. decoder) that meets the specifications.** This class must have two constructors to specify how a tree can be built -

**CanonicalTree(byte [] codeword_lengths)**;
which builds a canonical optimal tree given the array of 256 codeword lengths. A codeword length of 0 means that the corresponding symbol does not occur in the input data. This method is used by the receiver to construct a canonical optimal tree from the codeword lengths sent by the sender.

**CanonicalTree(int [] frequencies)**;
which builds a canonical optimal tree from the frequencies of the 256 symbols. A 0 frequency means that
the symbol is not present.

This constructor is used by the sender to construct a canonical optimal tree from the input data and by the receiver to construct the tree for the second-level coding of the codeword lengths. The canonical optimal tree is however, defined in terms of the codeword lengths alone. Hence, in this constructor, you will have to get the codeword lengths first from the frequencies by executing a simplified version of the Huffman algorithm (you could always execute the Huffman algorithm completely and get the codeword lengths from the tree thus built, but this is not necessary and it is possible to do this using simpler data structures) and then use the code of the previous constructor to build the tree. This simplified version of the Huffman algorithm must be the same with respect to non-deterministic choices for both the sender and the receiver. This means that ties must be broken in exactly the same manner. We therefore require that the following choices be made in the implementation of the Huffman-like algorithm:

1. Each node is associated with a symbol - an internal node of the tree is associated with the smaller of the symbols of its children and a leaf node is associated with the symbol it represents.

2. Whenever there is a tie among nodes with equal frequencies, the algorithm selects the nodes in the lexicographic order of their corresponding symbols - that is, in the order of their ASCII values. Specifically, it chooses the two lexicographically smallest values.

This class also should contain four other methods -

```java
int encode(BitStream input, BitStream encoded) ;
```
which takes a stream of bytes in the form of a BitStream and outputs the encoded BitStream (just the encoded data, without the header). The method returns 0 on failure and non-zero on success.

```java
int decode(BitStream encoded, BitStream decoded, int n) ;
```
This takes a stream of bits representing the encoded data without the header information and outputs the decoded BitStream till at most n symbols are decoded. If n is negative, then it decodes till the input stream returns EOF. The method returns 0 on failure and non-zero otherwise.

```java
byte [] codewordLengths() ;
```
return an array of 256 bytes, with byte i being the length of the codeword of the ASCII character with value i. The length should be 0 if the corresponding symbol does not occur in the input data.

```java
byte [] codewordLengthFrequencies() ;
```
returns the frequency of distribution of the codeword lengths as a 256 byte array. Byte i is the number of codewords with codeword length i, with codeword length 0 interpreted as above.

```java
static int [] frequencies(BitStream input) ;
```
The code to calculate the frequencies of occurrences of the symbols, that is, the 256 ASCII characters, in the input data stream should be encapsulated in this static method in the CanonicalTree class. This method returns the frequencies as an int array of size 256. The BitStream input is made up of bytes only and can be read using the readByte method of the BitStream class.

The last optimization in Section 2.4 involving the special encoding scheme should be encapsulated in a single class named “SpecialEncoding”. This contains two methods:

```java
int specialEncode(byte [] input, BitStream encoded) ;
```
which takes an array of 256 bytes through the array “input” and outputs the encoded data (where the encoding is performed according to the pseudocode in Section 2.4) to the BitStream “encoded”. The return code follows the usual convention.

```java
byte [] specialDecode(BitStream encoded) ;
```
which outputs the 256 byte array formed by decoding the bitstream given. This method should terminate when it recovers 256 bytes. No other indication of termination will be found.
The code that ties together all these individual bits of code is encapsulated in the classes “Sender” and “Receiver”, which represent the sending and the receiving process respectively. These classes contain functions send and receive respectively, whose code is provided by us. This code uses the interfaces specified so far and it is necessary for you to stick to the specifications closely to get your final code to run. These two classes also have main methods to take parameters from the command line. The encoding process is all put together in the send function and the decoding process in the receive function. The transmission of the message is simulated by writing the encoded message to a file through the BitStream class and then reading the file in the Receiver class using the BitStream class.

The code for the BitStream, Sender and Receiver classes will be put up on the web page.

4 Suggestions for Debugging

One simple way to check that the code works is to run the encode and the decode methods of the CanonicalTree class within a single class and check that they return the original data. You may need to write a new driver class for this or you may just modify the Sender class. You might want to read the Sender class code for this.

Another way to debug and understand the code is to print out the tree in some format. It is then possible to check for small examples that the CanonicalTree class works as expected and that it constructs the right tree.

5 Deliverables

You are required to turn in the source code files for CanonicalTree.java, SpecialEncoding.java and any other auxiliary source code files to “project_1”.

6