Project 2: File and Folder Compression

"The volume of data managed by enterprises grows by 2x every 18 months"
–Sign outside Sandisk Computing Lab, Cory Hall

Background Information

In this project, you will be implementing variants of Huffman encoding, a lossless data compression algorithm that is used in encoding schemes such as JPEG and MP3 (MPEG-1).

Before we discuss Huffman encoding, you have to understand bits. From Wikipedia:

> A bit is the basic unit of information in computing and digital communications. A bit can have only one of two values, [...] most commonly represented as 0 and 1.

Basically, all information sent over the internet, pixels on your computer monitor, and files stored on your hard drive are represented by long sequences of zeros and ones. When you compress a file, you're replacing a series of zeros and ones with a (hopefully) shorter sequence of zeros and ones that represents the same information.

In text files using ASCII encoding[1], characters take up 1 byte (8 bits) each. The main idea behind Huffman encoding is to represent the most commonly occurring characters using codewords that are less than 8 bits in the compressed file at the cost of representing the less common characters with more than 8 bits. In practice, this reduces a text file's size.

From *Algorithms* by Dasgupta, Papadimitriou & Vazirani:

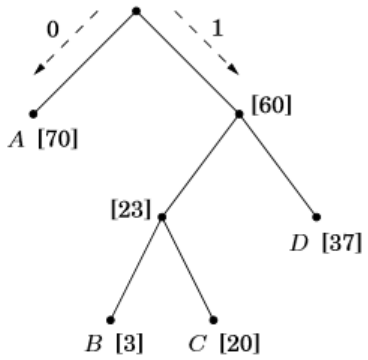
> A danger with having codewords of different lengths is that the resulting encoding may not be uniquely decipherable. For instance, if the codewords are {0, 01, 11, 001}, the decoding of strings like 001 is ambiguous. We will avoid this problem by insisting on the *prefix-free* property: no codeword can be a prefix of another codeword.
>
> Any prefix-free encoding can be represented by a *full* binary tree – that is, a binary tree in which every node has either zero or two children – where the symbols are at the leaves, and where each code word is generated by a path from root to leaf, interpreting left as 0 and right as 1. Decoding is unique: a string of bits is decrypted by starting at the root, reading the string from left to right to move downward, and, whenever a leaf is reached, outputting the corresponding symbol and returning to the root.

We can construct an optimal encoding tree greedily from the bottom up. Begin with the set of leaf nodes, with each leaf node having a character and a character frequency (e.g. character: 'a', frequency: 99). Find the two nodes in your set with the smallest frequencies (say, nodes n_1 and n_2 with frequencies f_1 and f_2) and create a new node with n_1 and n_2 as the left and right children, and $f_1 + f_2$ as its "frequency". Add this new node to your working set of nodes and remove n_1 and n_2 from that set. Repeat this process until there is only one node left in your working set: the root node of the Huffman encoding tree.

[1] http://en.wikipedia.org/wiki/ASCII

Symbol	Frequency
A	70
B	3
C	20
D	37

Symbol	Codeword
A	0
B	100
C	101
D	11



A prefix-free encoding example with one possible codeword mapping. Frequencies are shown in square brackets. (image from *Algorithms* by Dasgupta, Papadimitriou & Vazirani)

This gives us the following encoding algorithm:

1. Count characters' frequencies
2. Construct a Huffman encoding tree
3. Construct a table mapping characters to codewords
4. Encode a sequence of characters by following the character-to-codeword table

The decoding procedure was explained earlier in this document.

End of File Considerations

Let's say we have the character-to-codeword mapping {'a': 0, 'b': 10, 'c': 11}. We would encode the string "abc" as the binary string 01011. However, modern computers store data in bytes (blocks of 8 bits), not individual bits, so we would have to pad our binary string 01011 with three arbitrary garbage bits at the end in order for us to store the encoding of "abc". If we used three 0's, we would decode 01011000 as "abcaa", not our original string "abc". If we used three 1's instead, we would decode 01011111 as "abcc" and then our program would crash – the trailing 1 does not match a valid character with our prefix-free encoding! Regardless of which bits we choose to fill up these remaining three spaces, our program won't be able to return the original "abc".

We can solve this problem by adding an "end of file" character to our set of valid characters. This "end of file" character would be treated as any other character when building the Huffman encoding tree or encoding a file's content. Now let's say we have the character-to-codeword mapping {'a': 0, 'b': 10, 'EOF': 11}. We would encode the string "ab" as 01011000 with the EOF codeword 11 following the b codeword 10 (it doesn't matter what the last three bits are), then correctly decode 01011000 as "ab", so long as we stop the decoding process after decoding the "end of file" character.

Supplementary Readings

If you would like a more detailed explanation of Huffman encoding, you should read these notes from Stanford University.

Equivalent Codemaps

Given an input file, there are multiple codemaps / isomorphic Huffman encoding trees that all have the same compression ratio. For example, given a file containing the text "aba", we could end up with any of the following codemaps:

Symbol	Codeword
a	0
b	11
EOF	10

Symbol	Codeword
a	0
b	10
EOF	11

Symbol	Codeword
a	1
b	00
EOF	01

Symbol	Codeword
a	1
b	01
EOF	00

Thus, given any input file, there will be multiple valid compressed versions of the file, depending on your implementation of `HuffmanTree.java` and how you build your Huffman tree. However, given any compressed file, there will only be one valid decompressed version of the file.

Helper Files

We've provided two helper files for you to use, which you can download here:
`FileCharIterator.java`
`FileOutputHelper.java`.

`FileCharIterator.java` is an iterator that outputs the content of a file one character at a time in binary string format. Sample usage:

```
// Assume "Test.txt" contains the string "abc"
FileCharIterator myIter = new FileCharIterator("Test.txt");
myIter.next(); // returns String "01100001", the ASCII for "a"
myIter.next(); // returns String "01100010", the ASCII for "b"
myIter.next(); // returns String "01100011", the ASCII for "c"
myIter.hasNext() // returns false
```

`FileOutHelper.java` has a method `writeBinStrToFile` that takes in a binary string (e.g. 01100001) and appends the corresponding character(s) to the end of a specified file (and creates the file if it does not already exist). The input binary string's length must be a multiple of 8.

You are not required to use these files – what matters is that your project successfully implements huffman encoding using our specified format – however, we strongly recommend that you use them if you aren't familiar with reading and writing bytes to files in Java. In particular, we recommend that any time you need to read or write from a file in this project you should just use these two files, unless you know for sure that you are reading/writing ascii text.

Disclaimer

Using Strings of 0's and 1's is somewhat inefficient. In practice, we would read/write bits one at a time using a language that has better low-level file I/O support such as C or C++. However, this class uses Java, Java only allows for the reading/writing of individual bytes, and we don't want you to worry too much about bit/byte operation details. Thus, we'll be using Strings of 0's and 1's for intermediary bit representation. It might be a good idea to use Java's `StringBuilder` object for efficiency reasons.

1 Huffman Encoding

Write a program `HuffmanEncoding.java` that compresses and decompresses files using Huffman encoding. Your program must run via command line and support encoding and decoding:

`encode`

Takes in a file name, compresses the file, and outputs the compressed file with a codemap header.

A codemap header is a table in text format that lists binary strings and their encodings. Each line of the codemap header should contain a binary string, followed by a comma, followed by another binary string (the encoding of the initial binary string), followed by a newline character (`\n`), and nothing else. Since files don't actually have an "end of file" character, one line of the codemap file will contain "EOF", followed by a comma, followed by a binary string (the encoding of "EOF").

Suppose we had the codeword mapping {'a': 0, 'b': 10, 'EOF': 11}. Then a printed codemap header would look like this:

```
01100001,0
01100010,10
EOF,11
```

For reference, here are some characters and their binary string representations:

`a:01100001`
`b:01100010`
`␣:00100000` (space character)
`\n:00001010` (newline character)

Given an input file to be compressed, the output file your program creates must contain the codemap header, followed by an additional newline character, followed by the contents of the compressed file. For example, with the codeword mapping {'a': 0, 'b': 10, 'EOF': 11}, the output file would contain:

```
01100001,0
01100010,10
EOF,11

[Contents of compressed input file]
```

Here are some details about this representation you should take careful note of:

- There is no space between the comma and the encoding
- An empty line is nothing more than two newline characters in a row
- The code map itself is not compressed. Only the actual contents of the file below it are
- The compressed file below the code map is NOT made up of the characters `'1'` and `'0'` concatenated together; it is made up the bits 1 and 0. If you open up the compressed file in a text editor, you won't

see the characters `'1'` and `'0'` making up the majority of the compressed files. Open up one of our examples to see for yourself. Compare this with the code map, which has lines that do contain the characters `'1'` and `'0'`. Note that each of these characters actually consist of 8 bits.

Usage:

```
java HuffmanEncoding encode target destination
```

- `target`: The name of the file to be compressed
- `destination`: The name of the output file

`decode`

Decoding should reverse the `encode` operation. Given an output file from `encode`, `decode` should reproduce the original file.

Usage:

```
java HuffmanEncoding decode target destination
```

- `target`: The name of the file to be decompressed
- `destination`: The name of the decompressed file (to be created)

We've provided some example files here, including both uncompressed and compressed versions (labeled as .huffman files). There is a small text file, a couple of stories from Project Gutenberg, and a small image (that doesn't compress very well). Note that when you compress one of these example files, the resulting compressed file might not look exactly like ours because it might have a slightly different code map. However, you should be able to decompress our .huffman files (as well as your own, of course!) to obtain something that exactly matches the original file.

Try opening one of our .huffman files in a text editor to confirm that it matches the format we described. You'll notice that the compressed file looks like garbage text. This is normal. Once you decompress it, it will become the actual contents of the file.

2 Huffman Encoding Variant

Write `FileFreqWordsIterator.java`, an iterator that iterates through a file's contents and returns the next word of the file (only if it is one of the most frequent words in the file), or the next character of the file in binary string format. For this part of the project, we will define words as strings of length ≥ 2 that do not contain any space or newline characters ("\n" is the only newline character we will be concerned with for the purposes of this project). For example, if we had the input "it was the best of times, it was the worst of times" and the number of most frequent words to keep track of (`n`) was equal to 4, then your codemap would contain entries for `{"it", "was", "the", "of", "␣", "b", "e", "s", "t", "i", "m", ",", "w", "o", "r"}`. If there are multiple words tied for the n^{th} most frequent word, you can break ties arbitrarily so that you only keep `n` words in your codemap.

Now, your program should support the command:

```
java HuffmanEncoding encode2 target destination n
```

where `n` is the number of frequent words to keep track of.

`decode` should work the same as it did in part 1. In fact, the only part of your code that should change for part 2 is the use of a different iterator.

Note: Part 2 will only work properly on text files.

Kitten Example

Here is an example. Let's say Kim likes kittens. She has a file with many kittens, and her name only once. It looks like this:

```
kitten kitten kitten kitten kitten Kim kitten kitten kitten kitten
```

Converting this text from ASCII to binary:

- `kitten` maps to `011010110110100101110100011101000110010101101110`
- `K` maps to `01001011`
- `i` maps to `01101001`
- `m` maps to `01101101`
- ␣ maps to `00100000`

Munim wants to compress this file. He specifies that his iterator frequency calculator finds two frequent words ($n = 2$). This is one possible codemap he could generate:

```
011010110110100101110100011101000110010101101110,0
00100000,10
010010110110100101101101,110
EOF,111
```

Munim then wonders what will happen if $n = 1$. This is one possible codemap he could generate:

```
011010110110100101110100011101000110010101101110,0
00100000,10
01001011,1100
01101001,1101
01101101,1110
EOF,1111
```

3 Zipper

Congratulations! If your project works so far, you should now be able to compress an individual file.

Let's try a new challenge: compress a whole directory of files into a single file. This single file is called an archive file – you've probably come across these in the form of .ZIP files. Our project will NOT be exactly like .ZIP – it will be a simplified version that gets at some of the same core functionality. We will call it Zipper.

Your task will be to write a Java program named Zipper.java that corresponds to the following spec:

After compiling Zipper.java, if you use the command:

```
java Zipper zipper [input directory name] [output file name]
```

it will create a single new file named [output file name] that essentially consists of all the of the files in the original directory compressed (using `encode` from part 1) and concatenated together (details on its exact format on the following page).

Then, you can use the command:

```
java Zipper unzipper [input file name] [output directory name]
```

where [input file name] is a file that was created after calling zipper on a directory. This command will decompress every file that was added to [input file name] and restore the original directory structure inside the directory [output directory name].

Let's give a bit more detail on what the file that is the output of the zip command should look like. For convenience, we'll call this file a .zipper file, though you aren't required to name your files like this.

The majority of a .zipper file will simply be a block of bytes comprised of each individually compressed file concatenated together (along with their codemap headers). In this block of the .zipper file you do NOT need to include compressed versions of the directories themselves – only the compressed actual files.

Above the block of compressed files in the .zipper file should be a *table of contents* structure. The table of contents lists each file name (including the whole path to the file – that is, the file name and all of the folders above it), followed by a comma, followed by an integer that indicates the byte number that the file starts at in the .zipper file (NOT including the number of bytes in the table of contents itself). For example, the number associated with the first file that appears in the .zipper files should be 0, indicating that it starts at the 0th byte of the .zipper file, not counting the table of contents. Further, the number associated with the second file should be the number of bytes in the first compressed file, since the second file appears directly after the first file. And the number associated with the third file should be the number of bytes in the first file and the second file added together, since it appears after both of them. These numbers might change slightly if you decide to insert newline characters between the files for readability's sake.

The table of contents should list BOTH actual files and the directories themselves. However, since directories do not appear in the chunk of compressed files, the table of contents should list their byte position as -1.

The order that file names appear in the table of contents does not necessarily have to reflect the order that the files actually appear in the .zipper file, because the position number contains this information.

There should be an empty line between the table of contents and the remainder of the .zipper file which contains all the other files concatenated together. This empty line tells you where the table of contents ends and the other files begin.

In summary, suppose we have the following directory structure that we want to use Zipper on:

```
dir1/
    file1.txt
    file2.txt
    dir2/
        file3.txt
```

Once we zipper it up, we should get one file that looks like this:

```
dir1/file1.txt,1 // 1 indicates file1.txt is the second of compressed files
dir1/file2.txt,0 // 0 indicates file2.txt is first file in chunk of compressed files
dir1/,-1         // -1 indicates dir1 is a directory, not a file
dir1/dir2/file3.txt,2
dir1/dir2,-1
[blank line]      // separates table of contents from actual compressed files
[compressed dir1/file2.txt, with code map in front]
[compressed dir1/file1.txt, with code map in front]
[compressed dir1/dir2/file3.txt, with code map in front]
```

Again, the compressed files can appear in any order, and the file names in the directory can appear in any order.

Testing

You should write tests for all nontrivial methods you implement.

Feel free to look for Java libraries that test whether two files contain the same contents, but otherwise you can just check equality by iterating through them byte-by-byte.

At the terminal, a fast way to tell if two files are different is to call

```
diff [file1] [file2]
```

Provided you have `diff` installed, if this produces no output, it means the files are the same. Windows users, look into using `fc` (file compare) instead of `diff`.

We also provided you with an example directory to try `zipper` on, and our our zipper of it. You should be able to unzipper it a copy of the example directory at whatever location you decide to unzipper it to.

Submission Information

Create a `readme.pdf / readme.txt` file that includes:

- the names and logins of all group members
- a description of each of your test's purpose
- a short summary of each member's contribution to the project
- the running times of each step of your encoding and decoding algorithms from part 1

Submit your solution to this project by Saturday August 2, 10pm PDT with the command `submit proj2`.

Your submission should include the following files:

- `FileFreqWordsIterator.java`
- `HuffmanEncoding.java`
- `Zipper.java`
- `FileFreqWordsIteratorTest.java`
- `HuffmanEncodingTest.java`
- `ZipperTest.java`
- `readme.pdf / readme.txt`

in addition to all other Java files (including test files) you wrote for this project. Only one group member needs to submit per group. If you modified `FileCharIterator.java`, you must submit your version as well. Otherwise, we will add the default one to your submission.

Grading

This project is worth 30 course points. The grade breakdown is as follows:

- Code correctness: 20
  - Part 1: 10
  - Part 2: 4
  - Part 3: 6
- Comprehensive testing: 4
- Proper Java conventions and code style: 3
- `readme.txt`: 3

Project 2 Checkoff

There will be a progress checkoff on Friday July 25 in lab. To get full points for this checkoff, you need to show your TA two things:

- Completed and working code for part 1 (encode and decode using `FileCharIterator`)
- A document that describes all of the methods you implemented and plan to implement for the entire project. Essentially you are writing your own skeleton for this project, since we didn't give you one. The document should list each method *with its complete method header*, along with a description of its input arguments and return value, and any side effects it has (what kinds of files it creates or writes, what kinds of instance variables it sets, etc).

Your TA will check encode and decode for correctness. Your TA will also look over your methods and let you know if it looks like your group is on the right track.

Appendix: Basic file I/O in Java

We're going to be reading and writing files in this project. There are lots of ways to go about doing this – the following will be our suggestions for what to do, but if you have ways of going about it that you prefer, you can use that instead. To use the classes we suggest, make sure you `import java.io.*;` at the top of your file – `java.io` contains common classes for dealing with input and output.

The File object

You'll find it convenient represent files in Java using the File class. Here's how to use it: Suppose you have a file on your computer named `"HelloWorld.txt"`. You can make an object representation of it in Java using the following syntax:

```
File f = new File("HelloWorld.txt");
```

You can do this for both files and directories.

Here are a couple of methods a File object has that you might find particularly helpful:

- `isDirectory()`: returns a `boolean` indicating whether or not the file is a directory
- `listFiles()`: if this file is a directory, returns an array of File objects contained in this directory
- `createNewFile()`: if a file with this name does not yet exist, creates an empty file with this name
- `mkDir()`: if a directory with this name does not yet exist, creates a directory with this name

There might be more helpful methods you can use too! Explore the Java API on the File object to see all of its methods:

http://docs.oracle.com/javase/7/docs/api/java/io/File.html

Streams

In order to read and write characters from a file, we'll use the concept of a stream. A stream is simply a source of data. For example, we can make a stream out of a file if we want to use it as a source of data in our program.

We can have input streams or output streams. If we want to read in a file into our program, we'll create an input stream from it. This means that the file will be treated as an input source of data. If, instead, we want to write to a file, we'll create an output stream for it.

Reading from a file

In Java, there are a lot of different input streams dealing with different kinds of objects. To create an input stream for a file, we'll use the file input stream. Here's how we create an input stream for the file f we created above:

```
FileInputStream fis = new FileInputStream(f);
```

Now we will be able to read bytes of data from this file by calling `fis.read()` repeatedly. Each time you call it, you will get the next byte from the file, until you run out of bytes. After you have read all of the bytes in the file, `fis.read()` will return `null`.

Reading bytes from the file, however, may not be that convenient for you. If you want to read whole lines of text from the file as Strings, you'll want to use something a bit more sophisticated than the `FileInputStream` – we suggest you use something called the `BufferedReader`. A `BufferedReader` can be created from a `FileInputStream`:

```
BufferedReader br = new BufferedReader(fis);
```

Then you can read lines from the file as strings by calling

```
br.readLine();
```

repeatedly. Each time you call it, you get the next line in the file, until you run out of lines. When you run out of lines, `br.readLine()` will return null.

When you're done with a `BufferedReader`, you should close it with

```
br.close();
```

Using a `BufferedReader` is only appropriate if your file is a text file organized into lines. Otherwise, for the purposes of this project you'll want to read and individual bytes from the file. It is possible to do this with `FileInputStream`, but we recommend you use our `FileCharIterator` instead if you don't want to work with the `byte` type.

Writing to a file

Writing to a file is mostly symmetric to reading from one. If you want to write a byte to a file, create an output stream from the file:

```
FileOutputStream fos = new FileOutputStream(f);
```

Then you can write a byte to it with

```
fos.write(//some byte);
```

If you want to write a string to a file, you'll want to use the `BufferedWriter`. However, a `BufferedWriter` cannot be created from a `FileOutputStream`. Instead it is created from a `FileWriter`:

```
FileWriter fw = new FileWriter(f, true);
BufferedWriter bw = new BufferedWriter(fw);
```

(the `true` indicates you want to append to the file, as opposed to overwrite it)

Then you can do

```
bw.write(//some String);
```

Both the `FileWriter` and the `BufferedWriter` should be closed when you're done with them.

IMPORTANT: If you're not already familiar with Java readers and writers, do not attempt to read and write the compressed portions of your file (or any part of a non-text file) using them. We only recommend you use them for writing lines of text, such as the code maps at the top of .huffman files, or the table of contents at the top of .zipper files. To read and write non-text, you'll want to deal with individual bytes. We just told you how to do this using streams, but we don't recommend you do this either – it will probably be easiest for you if you use our `FileCharIterator` and `FileOutputHelper`. If you use those two files, you'll never have to deal with the `byte` type. The only reason we told you how to read and write bytes using streams was for completion's sake, since reading and writing lines as Strings is built upon reading and writing bytes.

In summary, we think you'll find it easiest if you use `BufferedReader`'s `readLine` method and `BufferedWriter`'s `write` method for dealing with the code map and table of contents specifically, and use `FileCharIterator` and `FileOutputHelper` for everything else.

Appendix: FAQ

Q: Can I create temporary files in the middle of my code?

A: Yes, but make sure to delete them after your code finishes running.

Q: Can my program print output to `stdout`?

A: No, your program shouldn't print out anything.

Q: Do I need to be able to handle improperly formatted input files?

A: No, if you are trying to decode or unzipper a file you can assume that it follows the specifications.

Q: Can I use external libraries in my code?

A: You can use anything in the Java standard library, but you shouldn't use any external libraries (other than JUnit). Basically, your code should not require us to download extra JAR files in order for it to work. That said, we definitely encourage you to explore the Java standard library to look for useful utilities so you don't have to write them yourself. Taking a little time to explore might make the project a lot easier!

Q: Should I handle the case where I try to zipper up a regular file (not a directory)?

A: Yes. But you might not have to make a special case out of it, because well-written code should handle this case automatically.

Q: Help! I'm getting weird/inexplicable errors on this project and I'm not sure what to do!

A: First of all, read over the project specs again really carefully to check if there's anything you missed. The specs contain a lot of pretty important details about how to read and write files, and how to format your input/output. If you still don't know how to fix your error, you should learn how to use a search engine to find solutions to your program's errors.

Q: Can I modify `FileCharIterator.java` and/or `FileOutputHelper.java`?

A: Unless you know what you're doing, this is discouraged. If you do change either of these files, you must submit them along with your other project files.