1. Benchmark your code on the given calgary and waterloo directories. Develop a hypothesis from your code and empirical data for how the *compression rate* and *time* depend on *file length* and *alphabet size*. Note that you will have to add a line or two of code to determine the size of the alphabet.

Hypothesis:

The file length will increase the runtime by O(n) because every character of the file has to be iterated over and stored. The increase in alphabet should increase by O(log n) this is because the data is stored in a tree and does not have to be iterated over necessarily.

Compressing calgary directory:
Time taken: .321 seconds
Original length: 3226253
New length: 1814541
Percentage of space saved: 43.76%

Compressing waterloo directory:
first folder (Color):
Time taken: 1.474 seconds
Original length: 14761384
New length: 11666269
Percentage of space saved: 20.97%

bridge.tif -> bridge.tf.hf:
Original length: 65666
New length: 63542
Time taken: 0.023 seconds
Alphabet Size: 256
Percentage of space saved: 8.73%

circles.tif->circles.tif.hf:
Original length: 65666
New length: 15902
Time taken: .004 seconds
Alphabet Size: 256
Percentage of space saved: 14.32%

clegg.tf -> clegg.tif.hf
Original length: 2149096
New length: 2033920
Alphabet size: 256
Run time: 0.218 s
Percentage of space saved: 7.76%

france.tf -> france.tif.hf
Original length: 333442
New length: 263360
Alphabet size: 91

Run time: 0.029 s
Percentage of space saved: 10.65%

Based on this data from compressing both the Calgary and waterloo folders, one can see that the length of the file affects the run time. When the file is twice as big, the runtime becomes twice as big. This backs up my hypothesis that the runtime of the compression increases O(n) with n being the length of the file. This is because it needs to run through every single additional character and find its value in order to add it to the map. However, this only affects the runtime in the creation of the map. After that, just the number of characters used affects the runtime.

The big O relationship between the size of the alphabet and the runtime actually appears to be O(n log n). This is because when the processor runs it needs to loop through n times where n is the number of codes representing a letter. Then it needs to retrieve these codes from the tree that you create which takes O(log n) time. In addition, as alphabet size increases, the compression rate decreases. This is because it is easier to compress a file with many occurrence and the resulting file ends up being smaller.

2. Do text files or binary (image) files compress more (compare the calgary (text) and waterloo (image) folders)? Explain why.

As shown by the data above, the image files which are stored in the waterloo folder took longer to compress than the text files did which were contained within the Calgary folder. This is because not every ASCII character is used in most text files. Thus, the alphabet size is relatively low, increasing the compression rate. There are more repeated characters in a text file too. In an image, almost every ASCII value is used, and they are used multiple times. Thus, these images compress a lot less because each pixel is used more frequently and almost every color of pixel is used in every image.

3. How much additional compression can be achieved by compressing an already compressed file? Explain why Huffman coding is or is not effective after the first compression.

Kjv10.txt.compressed -> kjv.txt.compressed.hf

Time: 0.361s

Original length: 2488864 bytes

New length: 2451324 bytes

Percent space saved 1.15%

Kjv10.txt.compressed.hf-> Kjv10.txt.compressed.hf.hf

Time: 0.281s

Original length: 2451324

New length: 2452387

Percent space saved -0.04%

Melville.txt.compressed->thMelville.txt.compressed.hf

Time: 0.021s

Original length: 57199 bytes

New length: 56828 bytes

Percent space saved 0.65%

The percentage of space saved after compressing an already compressed file was minimal in all of the cases that I tested. For both of the already compressed text files that I tested, the percentage of space saved was under 2%. In addition, when I compressed a file that had been already compressed and compressed by Huffman, I actually made the fire larger by .04%. Huffman coding is not effective after the first Huffman encoding because it is lossless and fully compresses the file to the extent that it can. The algorithm compresses every letter the same amount, and in the same way. Thus, compressing an already compressed will just generate the same result with a few added bytes for the pseudo_eofs.

Devise another way to store the header so that the Huffman tree can be recreated (note: you do not have to store the tree directly, just whatever information you need to build the same tree again).

You could store the header in a global array. This would allow the tree to still be recreated. In addition, it would be just as efficient as the way that it is currently done.