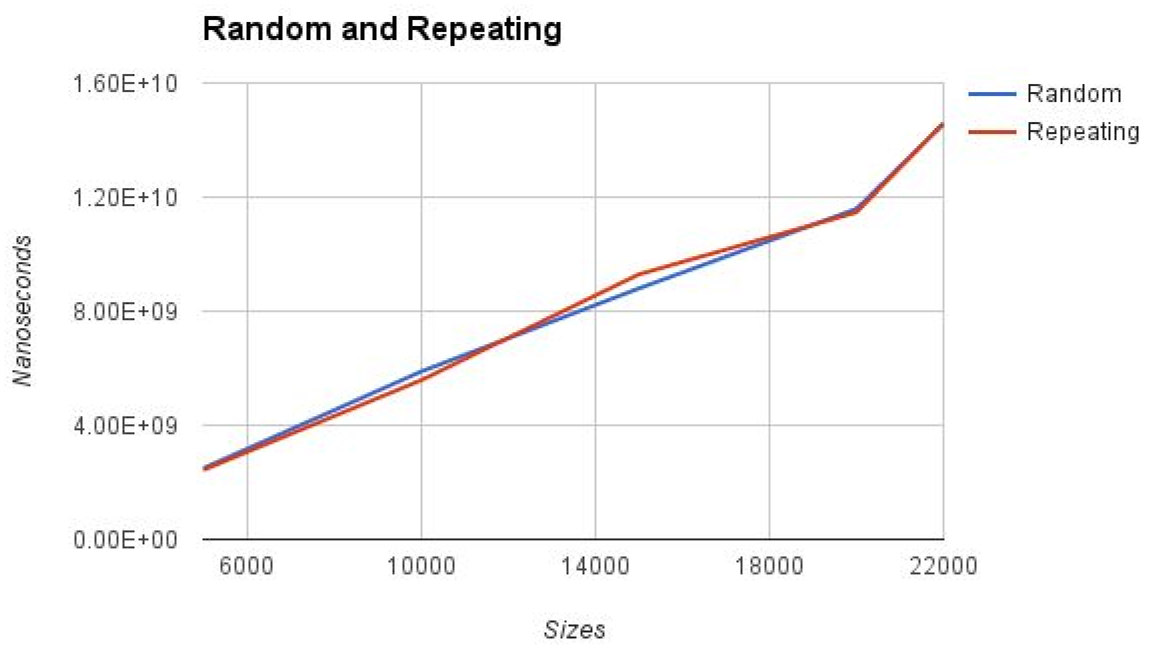
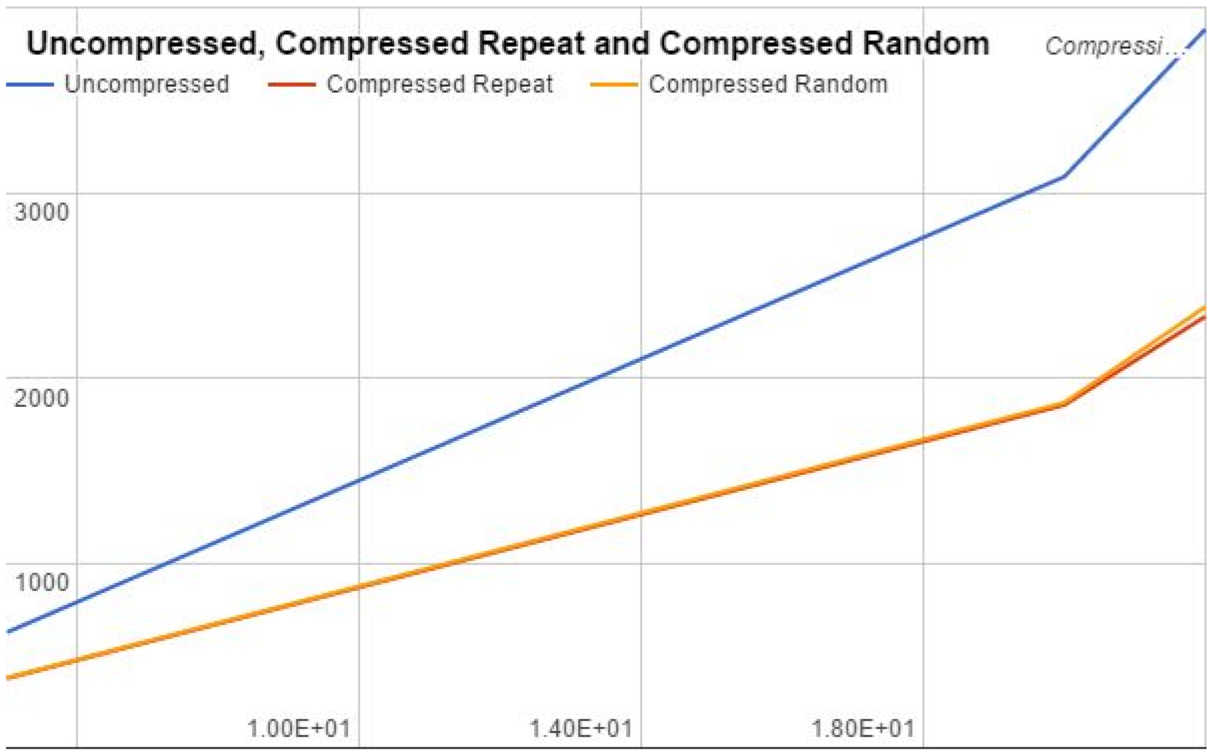
Travis Taylor Cassity
11/23/2016
Assignment 12

1. I created several text files. One batch had random letters, where the other was the same letter repeating itself throughout the whole file. The sizes ranged from 5000 characters to 22000 characters, and it took quite a while for it to run. As such, it was iterated only 5 times before it was averaged, and even then, took about 20 minutes for both batches to run.



Random vs Repeating Number Efficiency
Although this wasn't explicitly requested in the question, I found it to be an interesting question, and have included the data above. It appears that the efficiency is the same for randomly generated words, and words that are just a repeating character. I would imagine at significantly larger files sizes (as n -> infinity), the Repeating efficiency would be much higher, as if there is only one character to add, the tree is completed instantly, and traversing a tree of size 1 has an efficiency of O(C).



Compression
The file size for the number of characters, and the number of repeating characters, was virtually unchanged for my sample sizes. Logically, I would assume that the more variety of characters there are, the larger the file size, as the binary compression will have more numbers as it has to progress further down the list. (010, vs 111000100011 per char, if there were a large number of different strange characters)

2. For what input files will using Huffman's algorithm result in a significantly reduced number of bits in the compressed file? For what input files can you expect little or no savings?
Although my graphs in question #1 do not show it, I would presume that there would be larger file size saving in a file with very similar and repeating numbers, such as a text files of integers, as their range is only 0-9. This is due to the tree being significantly smaller, with a smaller header as well, and thus quicker traversal and smaller bits/bytes per char (or in this case, integers)
I would also presume that files with a wide variety of characters - such as a text file containing different language characters and various symbols (Arabic, Japanese, ^#$%!* etc) would have a much larger file size (that is, the compressed file will not provide as much saved space), due to a larger tree, and by extension, larger binary representation per character. Such as 000 for A vs 0010111010111001 for a complex Japanese character.

3. There's a number of reasons. Firstly, it was pretty easy to program it this way. Secondly, all of the 'mini-root' nodes, the non-leaf ones, already had the appropriate weights. This made calculating weights for branches closer to the root easier. It also means we do not have to adjust these weights as more leaves are added while the tree is being built.

4. Huffman's algorithm is lossless. Meaning it does not lose any data while being compressed and decompressed. You can compress/decompress a file a million times, and it will not lose (or gain) any data. Lossy data means you lose data per compression, which this algorithm does not.

5. About 9.