- “readme.pdf” that describes the design and implementation of your fileCompressor - include an analysis of the time and space usage and complexity of your program. You do not need to analyze every size function, but you should analyze the overall program's time and space used as a function of the size of the input.

- Recursive algorithm:
    - Given file path: create a dirent structure and read through directory
    - Using a while loop, traverse each entry in the directory
    - If the entry is a file → call the appropriate method (build, compress, decompress) and pass in the the new file path for the file
    - If the entry is a directory → recurse, passing in the new file path of the directory

- Recursive algorithm ensures that every file in a directory will be passed to the appropriate method

Our overall program was divided into three separate parts:

1. fileCompressor.c
    a. Contained the main, as well as handling command line arguments and dictating which functions to execute given the flags.
    b. Contained build/compress/decompress functions
        i. Build: invoking build calls a function addTokensToHashTable (also inside fileCompressor.c) which populates a hash table (holding words and frequencies) from a certain file. We implemented it this way so that when we were given a recursive build case, build() would be called on every single file in all sub directories so only one hash table would accumulate all words and frequencies. Then, after build is called, other helper functions were called (defined in huffmantree.c and hashtable.c) that built the minheap and huffman tree structures from the hash table.
        ii. Compress: Initialize a hash table that is responsible for mapping the words to its respective bitstring code. Read the given codebook and tokenize both the words and the bitstring code. For each line, we call the put(char* code,char* word) function from the hashtable library to find the index (hash function is just the sum of ASCII values of the word) to store the bitstring code in. After the codebook is read and the hashtable is populated with all mappings, read in the main file and tokenize the words/delimiters. Every time we see a delimiter, we call the get(HashTableCode**, char* word) function to get the bitstring codes of the corresponding word (preceding the delimiter) and delimiter and write it into the compressed file.
        iii. Decompress: Initialize the hashtable and populate it similarly, except the mapping is now from bitstring code to word. When reading in the compressed file, we increase our token bit by bit. Each time a bit is read and added to our token, we call the get() function to see if the

accumulated token currently has the corresponding word. If it does, we write in the word to a new decompressed file and we empty our token. If the get() function does not return a corresponding word, we continue to add to our token until it finds a word, at which we write to the decompressed file.

   c. Handled opening all files/subdirectories

2. hashtable.c
   a. Contained a global variable elements that held the number of all unique tokens that were read from all files, that was incremented every time a unique token was inserted
   b. Implements a hash function that returns a key to index the hashtable in
   c. Implements an insert() function that is used to map words to their frequencies
   d. Implements a put() function that is used to map words to their bitstring code (or vice versa)
   e. Implements a get() function that is used to fetch a certain value given the key
   f. Implements various print
3. Huffmantree.c
   a. Contained all functions required to create a minheap array of structures (struct item contained int frequency, char* word, item* left, item* right). Essentially minheap was an array of pointers to tree nodes.
      i. Function createArray() is invoked after the hash table is populated from all files. Instantiates an array of size (elements) and updates each array element to hold a word and its frequency.
      ii. Standard functions for heap were siftUp, siftDown, buildMinHeap, push, pop, all self explanatory. These functions were done in place.
   b. Contained all functions required to create a huffman tree from the minheap array. The way we implemented the huffman tree is that we did not create a new tree structure, rather, we combined elements from the array.
      i. Function combine_two creates a new item node with frequency (item1+item2), and a word (“subtree”) to denote that it is not a leaf node. This item node is then pushed back into the minheap.
      ii. Function buildHuffmanTree invokes combine_two until only one element is left in the minheap array, the head node of the resultant huffman tree.
      iii. Function buildCodebook recursively traverses the head node of huffman tree using inorder traversal. A parameter was an accumulator string called bitstring that had 0’s or 1’s depending on if it was the left or right path being taken, this being held in a char variable called mode. If a leaf node is the current node of the function, this is then printed to the HuffmanCodebook along with its code.

Worst case (Big O) time analysis

Recursive mode:

- Build:
    - Let n be the number of total files to build
    - Let m be the longest file out of the n total files
    - For every file, we read in one file: the file to build the codebook from
    - Reading the file takes O(m) time
    - Adding tokens to the hash table takes O(m) time
    - Creating a minheap array from searching every element in hash table takes O(1) time
    - Big O: O(nm)
- Compression:
    - Let n be the number of total files to compress
    - Let m be the longest file out of the n total files
    - For each file, we read in two files: the uncompressed file and the codebook
    - Reading the codebook takes in O(m) time
    - Storing mappings in hashtable takes O(m) time
    - Reading the file takes in O(m) time
    - For each word in a file, searching the corresponding code in the hashtable takes O(1) time
    - Big O for all would be O(nm)
- Decompression:
    - Same process for compression applies
    - Big O would be O(nm)

Big O for entire program would be O(nm)

Non-recursive mode:

- Build:
    - Let n be the file being read
    - Reading file takes O(n) time
    - Adding token to hash table takes O(n) time
    - Creating minheap array, searching for each word in a file in hash table, takes O(1) time
    - Big O: O(n)
- Compression:
    - Let n be size of file being read
    - Reading takes O(n)
    - Adding to hashtable takes O(n)
    - Accessing hashtable takes O(1)
      Big O: O(n)

- Decompression:
    - Similar processes to Compression
    - BIg O: O(n)

Space analysis

Our program creates:

- Build:
    - a hash table for word/frequency pairs, a minHeap array of structs
    - The hash table is initialized to hold 100 buckets, and handles collisions through linked list chaining. Space taken would be the number of unique words in a file.
    - The minheap array of structs is initialized to hold (element) # of structs. Space taken would be sizeof(struct item) * elements.

- Compress/Decompress:
    - At worst, the codebook size is the same size as the file being read
    - Thus, in the worst case, the hash table will take O(n) space, n being the size of the file read