I implemented the Burrows-Wheeler Data Compression Algorithm. The Burrows Wheeler Data Compression is a lossless algorithm which means that it allows the original data from the user to be perfectly reconstructed from the compressed data. The three steps for the BWDCA include: completing the Burrows Wheeler Transform, performing move-to-front encoding, and then performing Huffman compression. The Burrows Wheeler Transform rearranges a string into runs of similar characters by taking all possible suffixes of that string and using the last character from each of them to generate a new, usually more easily compressed string because more characters repeat more often. Move-to-front encoding then compresses the string further. It does this by repeatedly reading a character from the input message, printing the position in a given alphabet in which that character appears, and moving that character to the front of the sequence. Finally, Huffman compression returns the final compressed string. To decompress this text, the program simply runs the encoding processes in reverse. I made this because I wanted to expand upon my knowledge of data compression from the last project and this seemed like a perfect algorithm with real-world applications to do that. Bzip2 is a free file compression program that uses the BWDCA exactly as I've described it above. Besides lossless data compression, the Burrows-Wheeler transform itself (not including MTF encoding or Huffman compression), has been used in bioinformatics. The Burrows Wheeler Transform has a "last-first" mapping property, which means that the jth occurrence of a symbol in the BWT corresponds to its jth occurrence in its suffix array. For example, the 1st "a" in a Burrows Wheeler Transform will also be the first "a" in its given suffix array. Using this property, an FM-Index can be used to efficiently find the number of occurrences of a pattern within the compressed text, as well as locate the position of each occurrence. It keeps track of how many of each symbol have been seen in the BWT prior to its ith symbol, and at the end it determines the BWT index belonging to the first of each symbol in its suffix array. Using an FM-Index, you can determine all occurrences of a string in the BWT. This same logic has been applied on a more massive scale to genome sequence alignment. A human genome contains 3 billion base pairs and instead of storing all 3 billion, storing just the Burrows Wheeler Transform of the string significantly reduces the memory needed to complete the alignment searches. Researchers can then use the FM-index searching method to locate identical strings within the BWT. Several programs like Bowtie and BWA were developed with the Burrows Wheeler transform with the goal of decreasing the memory required for alignment. The BWT has also been implemented in image compression and data compression, however, much less commonly than its use in genome alignment sequencing.
One downside of BWT compression is that most often, there is often no improvement in compression size compared to Huffman Encoding alone until a certain length of string is used.

The "$" acts as an end of file character. Since '$' occurs only once in the BWT and rotations are formed using cyclic wrap around, we can deduce that the string with "$" at the end of it is the correct inverse string to find. It is lexicographically before any other character in the string so

that it does not interfere with any processes. The inverse BWT method will have errors if multiple end of file characters are included.

Takes all possible suffixes and their location, adds them to a list as a tuple with their original indexes, and sorts based on first character.

Using this sorted suffix array, the program constructs the BWT by looking at the original index of the ith value, subtracting one, and adding this to the BWT string.

In many cases, the output array gives frequently repeated characters' lower indexes which is useful in data compression. Initialize an alphabet, here it's all ascii characters. Read one character at a time from the input string and print out the position at which that character appears in the alphabet list. Move that character to the front of the list and repeat the process until indexes for all input characters are obtained.

Use positions in the list to index the original alphabet, move that character to front, and undo the move to front transform.

Creates a table of len(BWT) and sets the first column to the BWT. Include each sorted rotation starting from the original BWT and you will eventually find the original message. This is the most expensive part of the program taking O(n^2logn) time because it runs through double for loops and a sort function to compile the entire table.

Uses binary search to find the index at which a pattern occurs first and last and then prints the indexes at which these matches occur as well as how many total matches occur in the string. This takes O(nlogm) time where n is the length of the pattern and m is the length of the text to be searched.

The BWT way goes about things a bit differently. Using the FM-Index, you can determine all occurrences of a string in the BWT. It searches characters in reverse order to narrow down the range of possibilities for an occurrence of the string, and then returns all occurrences. Only takes O(n) time because the FM Index can be calculated in one scan of the BWT.