**Report on CSCI3280 Assignment 1**

**Basic requirement**

Some clarification of the file submitted:

1. The program should be run on Windows
2. In the both compress() and decompress() function, the current dictionary is deleted after the 4095th entry is used, not include the reserved EOF. A new dictionary would be created immediately. This algorithm matches the example file lzw\_example\_win. Since the entry just entered dictionary must not be read immediately, this algorithm will not contradict the reserved EOF and is able to run smoothly.

**Enhanced requirement**

1. In function compress(), I implement an AVL binary search tree as the dictionary. The ordering keys are the strings of each entries. Strings are compared by the standard function strcmp(). Most of the codes are written by myself, and some of them are modified from the teaching materials in other class (CSCI2100A 20/21 sem2), which are not online resources so I cannot provide references.  
     
   Code (part of):

Time complexity: ,   
*N* is total character count of the input files and *n* is the number of the patterns of the repeated strings on the input files.

All operation of AVL BST have the same time complexity , while reading each character from input, call search(), *N* times in total. If not found, call insert(), *n* times in total.

It also shows that the running speed not only depends on the input size, but also depends on how the strings pattern repeated.

Space complexity:

Each entry of dictionary stored in a node in AVL BST.

1. In function decompress(), I implemented a simple array of strings as the dictionary. Since the program only required to find the strings by code keys, and code keys can directly be the index of array, to achieve time complexity for searching.  
     
   Time complexity: ,  
   *N* is the numbers for code from compressed file.  
     
   Since the insertion of dictionary in this function is just append one more entry to the array, the time complexity is also . While reading each code from the input, call insert() and search(), *n* times in total.

Space complexity:

*n* is the number of the patterns of the repeated strings on the output files.

Each entry of dictionary stored in an element of array. The space complexity does not depend on input size but on the number of entries in the dictionary, which depends on the output file.