1. **messageFind(self, filename):**

Algorithm:

Open the file that is passed and read the characters line by line. A check to include only characters between a to z and A to Z, considering the sensitivity. When a new line is found using ascii code the word id concatenated to string 1 and the rest of the characters in the new line are concat to string 2. A maximum number function is created to find the max between two numbers. Then the function to find the lcs is created where first the m and n are set to the length of the strings. Then a loop is started to fill up the dynamic programming table, matrix is set using the length of the two string. Then the base case is set where the first row and column is set to 0. Then there is a check to see whether the characters are the same, if same then the table is filled with 1 + the value at the cross, and if the characters do not match then add the maximum of the value above or to the left. Then the sequence itself is reached by backtracking through the table. The i and j are set to the lengths of the strings, then a while loop is executed, and checks if the two characters are the same then, concat it to the empty string and decrement the i and j counter to go cross, else choose a path where the value is larger. The message is stored in the instance variable created.

precondition: a text file

post-condition: message stored in an instance variable

Time Complexity: O(nm) worst case, where m is the length of string 1 and n is the length of string 2

Short summary:

Preprocessing of the file takes O(nm) where the N are the words and M is the maximum length of the strings.
The function lcs takes O(nm) time where m is the length of string 1 and n is the length of string 2, for a 2d matrix and backtracking on the same matrix. Backtracking runs in time complexity O(n+m) because in the worst case we backtrack on the entire column then the row to reach the end from the bottom
So the total time complexity assumed here is O(nm) + O(nm) + O(n+m) so which eventually is O(nm)
basic backtracking idea only referenced from https://www.geeksforgeeks.org/printing-longest-common-subsequence/
Space complexity: O(nm) worst case, where m is the length of string 1 and n is the length of string 2. The matrix itself is used to backtrack to find the message so another

matrix is not needed to back track so space is reduced. Hence using only O(nm) time on columns and rows.

2. **wordBreak(self, filename):**

Algorithm:

The dictionary file is first read and then the words are concatenated as a main string. It is pre processed according to the characters from A to Z and a to z. Then the words are concat to the temp string. From the temp string, it checks if there is a new line if a new line is found it appends the word to the empty list, the temp list is initialised to empty. Then the loop goes through till the last word is appended to the list. The length of the message and and the longest word in the dictionary is assigned. The matrix is created where the rows are the length of the message and the columns are the length of the longest word in the dictionary. The loop is started on the length of message and the length of the longest word in dictionary. On every iteration a set of word is checked through slicing to the dictionary file if there is a match "1" is set to the matching position else a "0" is set if no match is found. The negative start is possible when extracting the start of the message to avoid it a "0' is set to the matrix. The backtracking basically looks from the last row if there is a "1" initialised it looks for the start and stop index and concats that word to the final string, then it initialises the temp string back to empty for the new word. If the word is not matching to the dictionary the extra words are added after the flag to the final string, since the algorithm was concatenating empty string at the end so I did length -1 to avoid the empty string

Precondition: reads in the dictionary text file

Post – condition stores the message in the instance variable

Time complexity: O(kM.NM) worst case, where k is the length of the message, M is the length of the longest word in dictionary and N are the words in the dictionary.

Short Summary:

The first loop runs O(k) times where k is the length of the message, and the nested loop runs O(M) times where M is the length of the longest word in dictionary and then the dictionary words are checked by the message which takes O(N) time to run along the length of the longest word in dictionary. So overall it takes O(kM.NM) worst time complexity.

Space complexity: O(kM) where k is the length of the message and M is the length of the longest word in the dictionary. So the tracking has to go through every column and row in worst case.