

Programming Project Report: Autocorrect

By: Amanda Christy, John Colbert, Marcus McCallum, and Hannah Shin

Overview of Approach

For our project, we used the LCS pseudocode from class in order to compare the word we input to a correctly spelled word on our word list. The words on the word list represent correctly spelled words that we would autocorrect our input word to if the input word was spelled wrong. For our word list, we used the Excel sheet, allWords.xlsx, and copied all the words from it into our own .txt file. This made it easier to search for the words that were correctly spelled versions of our misspelled words because they were all in alphabetical order. In order to decrease runtime, we used the hashmap data structure along with other algorithms that will be described in our “Algorithms” section. When we ran our code, we found that it autocorrected the words properly but we still play on adding extra features such as a GUI later on.

Algorithms

For the createDict function what we had done was read each line of the txt file containing all the words and then one by one adding each word to its respective key (big O should be $O(n)$ since we go through the whole word list).

The longestCommonSub function creates a matrix (we shall call it the c matrix) that will be the size of the $\text{len}(\text{word1}) \times \text{len}(\text{word2})$. We first loop through the indices of the first word and compare it to all the indices of the second word. If they match we add a point to the respective area in the matrix. If it does not we adjust our location marker in the c matrix and continue. To get the best score we would need to return $c[m][n]$ ($m = \text{len}(\text{word1})$ and $n = \text{len}(\text{word2})$ so $c[m][n]$ is the top right corner of our matrix).

The searchDict function takes in the user input word and then compares it to every word in its respective key value (the first letter of the user input word) using the longestCommonSub function. It will save the best score returned by the LCS function and its corresponding word.

The runtime for the searchDict function should be $O(nm)$. This is because searching through a hashmap should be $O(1)$ and so the largest contributor to this function's run time would be the LCS function making it $O(nm)$.

Data Structures

For our dictionary, we used a hashmap. Hashmaps map keys to their value pairs. The keys are the letters of the alphabet and the value pairs are the words we are inputting. This allows for an efficient lookup of the correctly spelled words given what words we input; by using this hashmap, it decreases the runtime drastically.

Other Choices

We plan to talk about extra features as we add them.

Asymptotic Runtimes

The complexity of our program is largely dominated by the LongestCommonSubsequence algorithm that we implemented. The runtime of this algorithm depends on the words that are passed into it, as the words will be compared letter by letter in order to obtain a similarity score. This of course is not the only function in our program, yet the runtimes of the others are relatively less significant by comparison.

The initializer function is straightforward and has a runtime of $O(1)$.

The createDictionary function runs through the entire text file, sectioning off the dictionary by first letter. There are 26 entries in the dictionary based on the 26 letters in the alphabet that could be the first letter of a given word. The runtime ends up being $O(n)$ where n is the number of words in the text file, as each word must be appended to the dictionary one by one.

The longestCommonSubsequence function, as mentioned earlier, provides the bulk of the runtime complexity of our program. The algorithm compares our two inputs letter by letter, which creates a runtime of $O(m*n)$ where m and n are the lengths of the two words that were passed into the function.

The searchDict function runs based on the hashmap that we implemented as our dictionary. Based on the input word, this function finds the corresponding key in the dictionary (the first letter of the word) and cycles through each value, running the LCS function on the original word and each word in the key's values. Since searching in a hashmap has complexity $O(1)$, the resulting complexity is the same as LCS itself, $O(m*n)$.