Bea Casey, Sam Citron, John Hunt, Wenyi Qian
Talmage
311 Project Write-Up
November 19, 2020

Intro/Project Description:

Our task was to implement a spell check system from scratch. Provided a list of words, our goal was to use a Longest-Common Subsequence detector to solve this problem. We chose to implement this project in Java. We used Java's built in ArrayList data structure to hold a given list of words as well as to hold a list of possible correct spellings per the user input.

Our interface consists of the program prompting the user to enter a word on the command line. It will then return a correct spelling of that word. Below we will describe and analyze our code in more sufficient detail.

Algorithm Description:

We used a Longest-Common Subsequence detector in order to find certain sequences in a given input. This will allow us to cross-check an input's subsequences against a .txt file containing correctly spelled words and return what is deemed the most likely desired spelling.

We achieve this by first reading in a .txt file containing correctly spelled words so we have something to cross-reference the inputs we receive against. Next order of business is to receive and record the user's input to be spell-checked. After we've received user input, we'll want to check to see if the given input closely matches any of the correctly spelled words from our .txt file. We do this by finding the length of the Longest-Common Subsequence in the input and rating how closely this Longest-Common Subsequence matches the words in the .txt file by finding the words with the greatest score. Once we have rated the matches, we can then add likely possibilities of the correct spelling of the input into a list. From there we can print the list which will give the user the most likely desired results from the original input, and finally choose one of the words as the correct or most likely output.

Algorithm Runtime Analysis:

A spell-check program could potentially be a very expensive program if not implemented efficiently and effectively. Implementations of a spell-check could have a runtime of $O(2^n)$ which would be certainly devastating and extremely inefficient because as the number of operations increases, the runtime increases exponentially which is never desired. However, our group was able to recognize that Dynamic Programming can be applied to this problem which reduced our runtime from what would have been $O(2^n)$ to a runtime of either $O(mn)$ or $O(n^2)$ if $m == n$.

Finding the Longest-Common Subsequence is a problem that has optimal substructure. If we are comparing two strings to find the Longest-Common Subsequence,

(L=(x[0,...m] and y[0...n])

and we find that the two strings have the same final character in their respective strings,

x[0,...m-1] == y[0,...n-1]

then we can rejigger the problem to look like:

L = (1 + x[0,...m-2] and y[0...n-2])

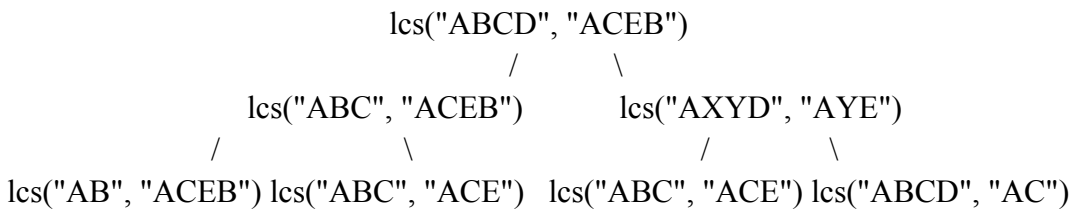
We can do this until we find a position where the final character of the respective strings don't match at which we would want to do something like

If x[0,...m-1] != y[0,...n-1]

L = max(L(x[0,...m-2], y[0...n-1]), L(x[0,...m-1], y[0...n-2]))

Both of these processes in conjunction will give us our optimal substructure because we can navigate our main problem using solutions from subproblems.

Finding the Longest-Common Subsequence also has overlapping subproblems which is the second requisite for dynamic programming. If we were to look at a recursion tree for finding the Longest-Common Subsequence of two similar strings, we would actually see that there would be a good amount of subproblems we'd be solving are subproblems that appear more than once on different sides of our recursion tree. For example:



As we can see, lcs("ABC", "ACE") would be getting solved twice which is overlapping. This allows us to use memoization so we can avoid solving problems we already have the answer to. Using both optimal substructure and overlapping subproblems we can drastically optimize this problem so we can be as efficient as possible.

In our solution, we do a lot of iterations through lists, which slows down the program. In future iterations of this project, it would be best to find a way to minimize the number of times and the size of the lists we iterate through.

Conclusion:

At surface value, a spell checker seemed like a difficult problem that would have a large runtime and would be difficult to optimize, however after we found that dynamic programming could be applied to this problem, we were able to solve and implement this program effectively and efficiently. We found that at the beginning of the project we had slight issues finding a place to start, but after we got the train rolling, putting the pieces of this program together actually became relatively simple. After we finished our implementation, we started looking at this program in a broader sense and we were curious how this program would perform if instead of a .txt file containing select words, we used a dictionary with every word in it. Obviously our implementation would be horrifically slow, but this further made us think about ways we could improve our solution. We were immediately thankful that our phones and computers are able to do spell checks and even grammar checks so quickly!