FIT2004 – Assignment 2 – Analysis

James Schubach – 29743338

Task 1

This DP algorithm makes a table with the length of the longest string. It does this by looping through the first and second word and checking if each letter is each to each other at different indices. E.g.: At index 1 of the first word and index 3 of the second word both have the same letter it will increase the value by one from index 0 and index 3 from the table. This in term makes the last value of the table the length of the longest word. From here a while loop will run at from the end of the word /DP table and backtrack through the table and find the string. It checks if both first_word[n] and second_word[m] are equal and if they are they add it to the list at the end and then decrease n and m and the idx. Otherwise it checks if the DP table at [n-1][m] > [n][m-1] and if it is it decrease the value of n. It will then check if the words are the same. It does this until either n or m are 0. The reason why this works is it uses the data in the DP table to back track through the string comparing values.
Time Complexity: O(nm) this is because at the start of the algorithm it loops through both words n and m respectfully. Making the outer loop n and inner m thus n * m. The while loop will also be at worst nm it has to decrement n or m each loop and will exact once one value reaches 0. Thus, our complexity is O(nm + nm) which = O(nm)
Space Complexity: O(nm), Our input is O(n + m), we then create a table of size n*m making it O(n*m) , we then create a list the length of the maximum subsequence of the string. This makes it O(nm + M) which m >= M, thus O(nm)

Task 2

This function breaks a string into words, based on the words in a given dictionary. It does this by starting at the end of a string and incrementing towards the start and checking string[i:] and string[i:j]. The algorithm implements DP by remembering words that it has checked. It does this by checking the DP table at j and if it's in the table at that index it will add that to the current index, thus allowing us to not check a subsequence of a given string if we have already searched the subsequence before. If it hasn't searched it before and it's a valid word, it adds it to the table at the given index. After it has gone through the whole string, it will loop through the DP table and take the first element at the DP index and then increase i depending on the length of that word, if there isn't a value at the DP[index] there it will at the index of the string to the word and continue until DP{index] has a value. Finally, it sets self.message as the string with spaces.
Note: where N be the number of words in dictionary, M be the maximal size of the words, k is the size of input string
Time Complexity: O(kM * NM), this is because the first part of the algorithm loops through the string from the end to the beginning making it k, while the inner loop will break every time string[i:j] > M. This allows the first part of the algorithm to operate in O(kM), then we do a lookup which is NM. If you work this out, it becomes O(kM * NM)
Space Complexity: The input is NM, we make a table the length of the string which is less then NM. The rest of the algorithm only uses variables, thus its O(NM)