CSCI311-DNAProject-Team9

James Howe, Jules Ward, Minh Anh Phan, Brian Richardson

October 2021

1 Introduction

Find our code here Github Link

2 Longest Common Substring

This algorithm will return the longest common substring. It does this by checking whether our first inputted string and second inputted string both have the same consecutive characters, this is a substring. Then the largest substring, that is common between the two strings, is the longest common substring.

The run-time is going to be $m{*}n$, and this gives us an $O(m{*}n)$. This indicates length of our DNA sequence string times the length of the query sequence string. What gives us this run-time is the fact that we have nested for loops that iterate based on the length of the strings inputted. We have multiple constant operations but these all cost $O(1)$.

3 Longest Common Subsequence

This algorithm is implemented in Python based on our class discussion. The algorithm will return the longest sequence of characters from the original string in the same order but not necessarily contiguous. Let m be the length of the first string and n be the length of the second string. For the algorithm, we create 2 $m * n$ tables, one to store the length of the longest subsequence and one to store the direction that we have iterated through. Once the table are filled, we iterate through the latter table to rebuild the string. Since the string is reverse, we have an $O(k)$ operation to reverse the string with k being the length of the longest subsequence, but we know $k \leq min(m, n)$ so it is absorbed in the bigger runtime. Therefore, the final run time of the algorithm will be $O(m * n)$.

4 Edit Distance

The edit-distance algorithm intuition lies within the longest-common-subsequence algorithm. However, the edit-distance algorithm will be making three recursive calls (or memoized look-ups), rather than the two from the longest-common-subsequence algorithm.

The base cases is that if one string is empty, and the other string has n characters, there are n operations required. After that, each comparison is made at the character level. If the characters are equal ignore them and check the next characters, if they are not equal, perform 3 operations: an insertion of the character from the second string to the first, a replacement of the character from the first string with the second, and a deletion of the character from the first string. The algorithm then ignores the character, and moves onto a smaller substring. Each operation, deletion, replacement, and insertion is counted as one operation. This is recursively computed until the first string equals the second string.

Since the problem can be recursively broken down, there is optimal substructure. Since, there are overlapping subproblems, and optimal substructure, this algorithm can be solved with dynamic programming. Given two words with length n and length m, the dp-table indexed as $dp[i,j]$, with $0 \leq i \leq n$, and $0 \leq j \leq m$, can be setup with base cases $dp[n,j] = m - j$, and $dp[i,m] = n - i$. This table has dimensions $m \times n$, thus to bottom-up compute the table will have algorithm complexity of $O(m \times n)$.

5 Needleman-Wunsch

The Needleman-Wunsch algorithm follows the same concept as edit distance, except we assign different scores for different operations on the string, the 3 operations we score are a match, a mismatch, and an insert/delete(indel), typically the basic scoring setup is +1 for every match -1 for every mismatch, and -1 for every indel. Varying the values allows us to emphisise or deemphisis specific operations, for example if we cared heavily about having certain a sequence match a lot of the query, but didn't care if we had to add characters to make it happen, then we could score +1 for every match, -1 for every mismatch, and 0 for every indel. The Dynamic programming approach to the solution uses a table to memoize optimal sub solutions. The table is the length of string 1 X length of string 2 and each cell indexed (i, j) represents the score of the substrings of string 1 from 0 to i and string 2 from 0 to j. The paths to the left and one up represent and indel operation, and the path to the upper left diagonal represents a match/mismatch, depending on if the base letters for cell (i, j) match. When constructing the table start at the upper left corner and then go on the diagonal which goes left to right, bottom to top, and use the previous entries as described previously to contruct the score for the current cell, using the smallest value with the appropriate score added. at the end of the algorithm the bottom right corner is the total score of the algorithm.
The Needleman-Wunsch algorithm with my implementation goes through every cell in the table once doing 3 conditional checks for each cell, which is constant, so the runtime of the algorithm is $O(|S_1||S_2|)$ with S_1 and S_2 being the input strings.