CSCI311-DNAProject-Team12

Cole Zehe, Cole Hausman, Rob Barlow, Nick Satriano

October 2021

1 Longest Common Substring

- Brief Description: Finds the longest common suffix amongst all possible substrings of both sequence strings inputted to the algorithm. The common suffix with the maximum length is the longest common substring.
- Asymptotic Runtime: $O(m \cdot n)$
- Runtime Analysis: The main section of this algorithm has two nested loops. One for loop goes from 1 to m, and the other goes from 1 to n. The work done within those loops is constant each time through. Therefore, we can conclude that this algoirithm is $O(m \cdot n)$

2 Longest Common Subsequence

- Brief Description: Creates a table in which to be filled out by the algorithm. If a character matches with the current table cell, then add one to the diagonal; Otherwise, take the largest value from the last column and row for the current cell. When the table is full, the cell of the last row and last column contains length for the longest common subsequence.
- Asymptotic Runtime: $O(m \cdot n)$
- Runtime Analysis: The major part of this algorithm is two nested loops. One of the for loops goes from 0 to n, and the other for loop goes from 0 to m. Each time through the loops, a constant amount of work is done. We can conclude that this leads to a runtime of $O(m \cdot n)$

3 Edit Distance

- Brief Description: Similar to the longest common subsequence method, it creates an $m+1 \times n+1$ table. It checks if two letters in the sequence are the same. If there are not equal, it takes the minimum of substituting, inserting, and deleting the current character to match the sequences. The edits to transform the *unknown* sequence to the *known* are stored in table b. When using this method, the user is given the option to see the edits needed to convert one string to the other. White characters mean no changes were made, blue means the original character was substituted for the one currently there, green means the character was inserted, and red means the character was deleted.
- Asymptotic Runtime: $O(mn)$
- Runtime Analysis: In the code, the tables to store solutions to the subproblems (table c) and the edits (table b) need to be created and initallized. Since these are both $m+1 \times n+1$ tables, this takes $O(mn)$ time. To solve the problem, each cell in the table needs to be filled out. Filling out a cell can be done in $O(1)$ time. The runtime will be the number of cells in the table times the work per cell. Since there are $(m+1)(n+1)$ cells in the table and it takes $O(1)$ time to fill in a cell, the runtime is $O(mn)$. To print the edits, no cell is visited more than once, so in the worst case (each cell takes $O(1)$ time again) is $O(mn)$. Since the largest term is $O(mn)$, the runtime of this method is $O(mn)$.

4 Needleman-Wunsch

- Brief Description: This algorithm is used to align two strings. This is done through three key parameters: match, mismatch, and gap. Match and mismatch are fairly straightforward, they simply represent the values which are added or subtracted when values match or do not. Next, the gap parameter comes into play, the gap represents how we align the strings in our output. A negative gap will prioritize larger gaps with more letters of each original string input being concurrent. Whereas a gap of say 0 will prioritize a larger amount of smaller gaps with fewer letters of each original string being concurrent. Generally speaking, its better to have fewer, larger gaps.
- Asymptotic Runtime: $O(mn)$
- Runtime Analysis: First, an mXn table is initialized where m is the length of the first string and n is the length of the second string. We then iterate through this entire table which takes $O(mn)$ time. For each iteration we create an array of length 3 with looks at the three neighbors of each cell (left, up, diagonal). This is a constant $O(3)$ time per iteration. Finally, we traverse back through our second table, which keeps track of our gaps which takes theoretically $O(mn)$ in worst case but never quite reaches this time complexity. Therefore the runtime of this algorithm can be simplified to $O(mn)$.

5 Zehe's Algorithm

- Brief Description: This algorithm aims to find the closest DNA sequence in a unique way. It does so by comparing how many of each DNA nucleotide letter (A,T,C,G) there are relative to the total count, and comparing those ratios.
- Asymptotic Runtime: $O(n)$
- Runtime Analysis: First, it goes through each character in the target sequence t and adds to letter counts in a dictionary. After this for loop through t's characters, it divides each value in the character count dictionary by the total length of t, in order to get ratios to total. Then, a for loop goes through each sequence to be compared with t. Each character in each of the sequences is counted and ratios are found in the same process as for t. If the ratios are closer to the target sequence than the previous closest, then the best sequence is updated. Because this entire process increases linearly by the number of sequences being compared and the length of the sequences, we can say that this algorithm is $O(n)$.

6 Triples Algorithm

- Brief Description: This algorithm finds the most common gene triplet within the target sequence. It then compares the number of occurrences of that triplet within each of the gene sequences passed to the function.
- Asymptotic Runtime: $O(n^2)$
- Runtime Analysis: First, the algorithm iterates through each character in the target sequence and keeps count of each triplet occurrence in a dictionary, finding the max. Then, the algorithm iterates through every key in the sequence dictionary that is passed to it and then iterates through each key's value. During this nested loop, the algorithm keeps track of the number of times the max triplet occurs for each key. The sequence with the highest number of matches is the output. Since the process will take n times longer per sequence in the sequences dictionary, we can say that runtime will increase n times for each sequence added, making this algorithm have a runtime of $O(n^2)$.