Genes tree

1)

The code uses a bottom-up dynamic programming approach to calculate the length of the LCS. This involves filling a 2D list, lookup, with the lengths of the LCSs of all substrings of X and Y. Once the lookup table is filled, the code uses it to find and return all LCSs of the full strings X and Y.

The python code that returns all the longest common subsequences between two strings and their length is in code cell 2 in the appendix. Code cell 1 has the genes input, and code cell 3 has the assertion test cases to demonstrate the code's functionality.

2)

The matrix in figure 1 shows the number of unique LCSs between each two strings. Using a set data structure ensured that thy're unique combinations as a set doesn't allow for duplicates.

	a	b	c	d	e	f	g
a	1	1	48	2	1	2	28
b	1	1	220	2	32	824	12
c	48	220	1	144	4	2	512
d	2	2	144	1	16	12	2
e	1	32	4	16	1	1	120
f	2	824	2	12	1	1	36
g	28	12	512	2	120	36	1

Figure 1. Matrix of the number of unique LCSs between each two strings

Total sum of all numbers= 4049

total sum of all numbers without considering each string with itself = 4042

total number of unique LCSs = 2028

total number of unique LCSs without considering each string with itself = 2021

To compute the LCS of two strings, the code creates a two-dimensional array lcs_matrix of size (n+1) x (m+1), where n and m are the lengths of the strings. It then iterates over the characters in each string, comparing them to find matching characters. If two characters match, the code adds 1 to the value in the lcs_matrix at the indices corresponding to the characters. If the characters don't match, the code takes the maximum values in lcs_matrix at the previous indices. Once the lcs_matrix has been populated, the code stores the length of the LCS for the two strings in len_lcs_matrix. Finally, it creates a Pandas data frame to display the matrix in a more readable format.

The output of code cell 4 in the appendix shows the 2D LCSs lengths matrix of dimensions(7,7).

C- The matrix shows the length of the longest common subsequent between two strings; thus, the greater this length is, the stronger “closer” the relation between the two strings compared. For example, the length of the LCS between gene a and gene d is 73 (the longest LCS between a and any other gene). That means that gene d is the most related to gene a. However, examining the matrix that way doesn’t give any inferences about parental relationships. It only shows how close the connections are.

3)

A- The local greedy strategy would use the percentage of LCS length relative to the length of the original string to infer parental relationships. For this purpose, I created a matrix of the

percentage of LCS length relative to the row string shown in the output of code cell 5. For example, the value in location (q,k) is the percentage of LCS between strings q and k relative to string q. The value in location (k,q) is the percentage of LCS between strings q and k relative to string k.

This can be utilized to infer parental relationships because the parent gene contains more of the child gene than the child gene contains the parent gene. Thus, the higher percentage value between the LCS and the two genes tells the parental relationship. For example, in the generated matrix, the highest percentage value for row a is with d (a,d), then we check (d, a); if (d, a) is higher, then d is the parent of a, and if (a,d) is higher, then a is the parent of d, which is true for this matrix.

Following this greedy strategy, finding the highest percentage for each row and comparing it with the corresponding value in columns, then identifying the parent based on the highest, we'll obtain the tree shown in figure 2.

a → d

d → b

e → c

d → g

e → f

e → f

d → g

This strategy is greedy because it only uses direct comparisons between two values and infers a parental relationship.

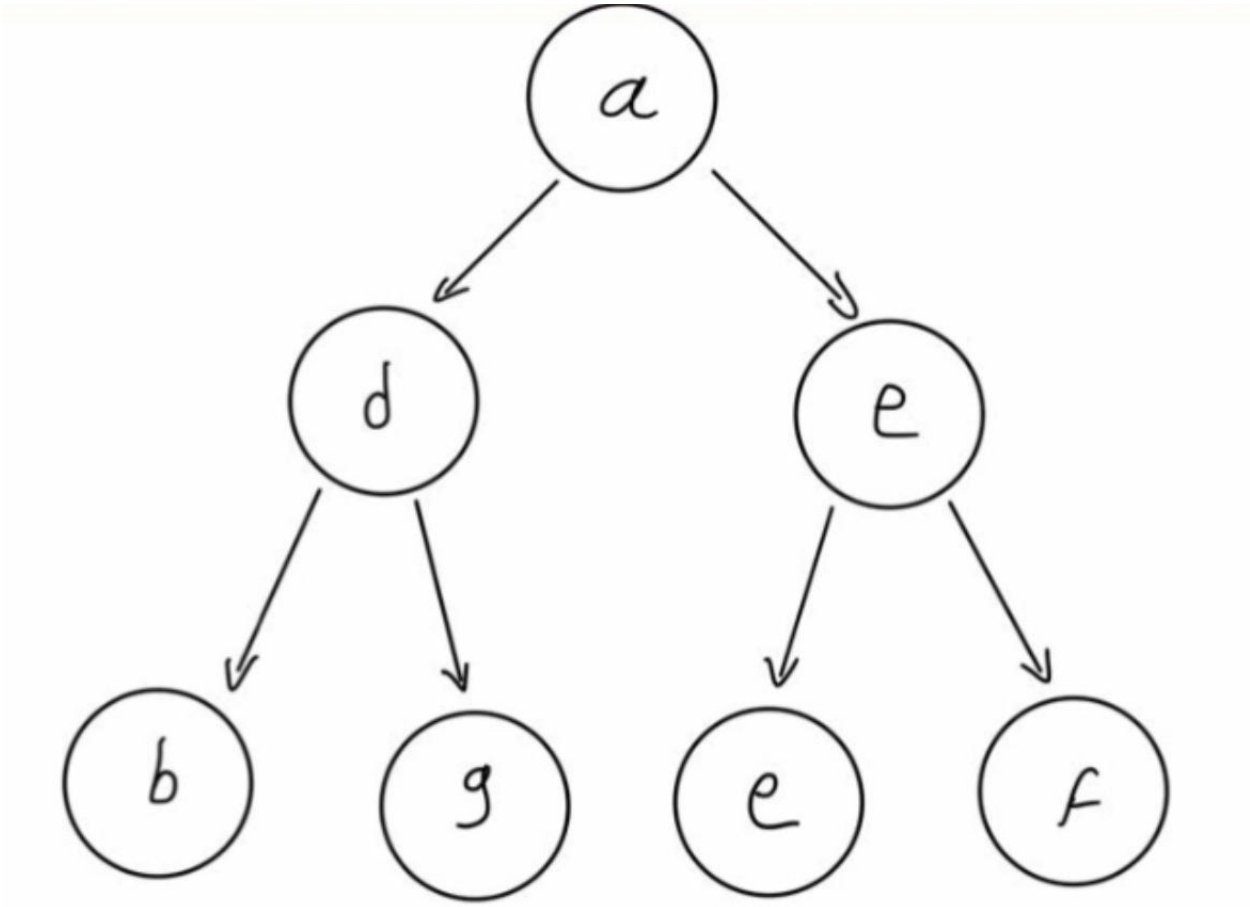

Figure 2. The genealogy tree of both greedy and global approaches

B- The global strategy would find the relationships using a dynamic programming approach of finding the shortest edit distance between two strings. This approach involves finding the minimum number of operations needed to transform one string into another. The operations are insertion, deletion, and mutation of a single character. The approach uses dynamic programming as it has an optimal substructure and overlapping subproblems. The code is in code cell 6

- Optimal substructure: to find the shortest edit distance between two strings, we can break down the problem into finding the shortest edit distance between the first k characters of

the first string and the first l characters of the second string. Once we have found the optimal solution for this subproblem, we can use it to solve the main problem.

- Overlapping subproblems: to find the shortest edit distance between two words, we may need to find the shortest edit distance between the first k characters of the first string and the first l characters of the second string multiple times. This repetition of subproblems leads to overlapping subproblems that the approach uses memoization to solve only once.

After getting all the distances between every two strings, we add all the distances from each string to the others (for example, the distance from a to b, a to c, a to d… etc.) and save them as the sum of distances from a. Since we do the same for all the strings, we obtain a list of all the sums. Those sums show how connected each string is to all other strings. The grandparent string is the most connected (having the shortest distances) to all other strings. Thus, the string with the lowest distances sum value is the grandparent.

After identifying the grandparent by comparing it to all the values, we identify the parents by finding the most connected strings “shortest distance” to the grandparent. Then, we identify the children by finding the most connected strings “shortest distance” to the parents. This is done by the tree_maker() function in code cell 7, which constructs the genealogy tree using the aforementioned strategy.

The resulting tree is the same tree shown in Figure 2.

C- After using the greedy approach and the global approach, the two tree predictions turned out to be similar.

The first approach used a greedy strategy of finding the highest percentage value, comparing two strings at a time to establish parental relationships, which turned out to be a valid strategy for constructing the genealogy tree.

The second approach used a global dynamic programming approach to find the global strongest relation value to all other strings “lowest sum of distances,” and consider it the grandparent and then build the tree based on the strongest relations to each member of a certain generation. Thus, it guaranteed an overall optimal solution to the problem of reconstructing the genealogy tree.

4)

Greedy:-

- The complexity of the LCS algorithm depends on the lengths of the two input strings, X and Y. The LCSLength() function has a time complexity of O(mn) and a space complexity of O(mn), where m and n are the lengths of the two input strings. The findLCS() function has a time complexity of O(2^(m+n)), where m and n are the lengths of the two input strings. This is because the function has to consider all possible combinations of subsequences of the input strings, which can be very large in number. The space complexity of the findLCS() function is O(m+n) since it only stores the LCS and its length in memory. Thus, the overall time complexity of the LCS algorithm is O(mn + 2^(m+n))
- The time complexity for building the matrix is O(n^2 * m^2), where n and m are the lengths of the two strings being compared. This is because the main loop iterates over each pair of strings in Set_Strings, and for each pair, it creates a two-dimensional array

lcs_matrix of size (n+1) x (m+1) and iterates over the characters in the two strings, resulting in a nested loop with a total of n * m iterations. The space complexity of this algorithm is also O(n^2 * m^2) because it creates a two-dimensional array lcs_matrix to store the lengths of the LCSs for prefixes of the input strings. This array takes up n * m space in memory. In addition to this, the algorithm also creates a matrix len_lcs_matrix to store the lengths of the LCSs for each pair of strings, which takes up n^2 space in memory. Therefore, the total space complexity is O(n^2 * m^2).

It is worth noting that the time and space complexities of this algorithm can be improved by using a different approach to compute the lengths of the LCSs. For example, instead of using a two-dimensional array, it is possible to use a one-dimensional array and some clever pointer manipulation to compute the lengths of the LCSs in O(n * m) time and space. This would make the overall time and space complexity of the algorithm O(n^2 * m) instead of O(n^2 * m^2).

So, since we're only using the matrix to infer the genealogy tree, the computational complexity for the greedy approach would be O(n^2 * m^2).

Global Dynamic Programming:

- The time complexity of the shortest edit distance algorithm is O(mn), where m and n are the lengths of str1 and str2, respectively. This is because the function uses a two-dimensional array of size m x n to store the results of subproblems, and the time taken to fill in the values of each element in the array is proportional to m and n. The

space complexity of this algorithm is also O(mn) since the two-dimensional array used to store the results of subproblems takes up m x n units of space.

- Since distances() function iterates over strings twice, and for each pair of strings, it uses the editDistDP() function to calculate the distance between the two strings, the overall time complexity of the combined functions is O(mn^2).

Experimental plots:

- The experimental plots produced for both algorithms show similar behavior to the theoretically driven complexity analysis as they're both scaling quadratically.
- The greedy approach graph is produced in code cell 10 in the appendix. It shows sharp rises and quadratic growth as the number of strings increases. This is expected because its complexity is O(n^2 * m^2), which has two quadratic quantities multiplied.
- The global approach graph is produced in code cell 12 in the appendix. It shows quadratic growth as the number of strings increases. This is expected because its complexity is O(mn^2), which is a quadratic complexity.

5)

A- To find the probabilities of each possible operation, I used the Needleman-Wunsch algorithm, which is a dynamic programming algorithm used for sequence alignment in bioinformatics. It is used to align two sequences of nucleotides or amino acids and find the region of maximum similarity between them. The algorithm uses a scoring system to determine the similarity

between the two sequences, with a higher score indicating a greater degree of similarity. The algorithm begins by creating a matrix of scores, with each element in the matrix representing the alignment score for a pair of nucleotides or amino acids from the two sequences. The algorithm then processes the matrix from left to right and top to bottom, filling in the scores for each element based on the scores of the elements around it. The final alignment is then determined by tracing a path through the matrix that corresponds to the highest overall alignment score. The output is the two strings aligned with differences indicated by dashes or unmatching characters.
I then passed all the parental relationships from the genealogy tree found before to a function that generates all alignments between parents and children.
After that, I created a function that counts each type of operation (deletion, insertion, mutation) between each parent and child separately using the following logic:

- If a character in the parent became a dash "-", then a deletion happened.
- If a dash "-" in the parent became a character, then an insertion happened
- If two corresponding characters in the parent and child do not match, then a mutation has happened.

An aspect that supports this logic is that the sum of deletions, insertions, and mutations from a certain parent to its child is the same as the shortest edit distance between them. This confirms that those are the actual operations that happened to generate the child from the parent.

I then passed all the parental relationships to this function to get all the numbers of operations to convert each parent to its child.

Finally, I calculated the probability of each operation by dividing the frequency of this operation by the length of the parent string. And to get the closest estimate for the actual probability, I added each operation type probabilities and divided them by the number of parental relationships (same as the number of separate probabilities) to get the average.

B- The python probability calculator is in code cell 17 in the appendix. After passing the parental relationships, the probabilities turned out to be as follows:

- Deletion: 4.1%
- Insertion: 9.3%
- Mutation: 4.9%

Those represent the chances for each character of the parent string to change in order to produce the child string. This also indicates that the probability of no change is 81.7%.

Those probabilities seem very reasonable because they're considered relatively rare, as stated in the prompt. Also, the algorithmic strategy used to obtain those probabilities is solid and justified properly. Thus, the probability estimation is a strong one.