Decoding the Relationships Between Genes: Final Computational Application

Minerva University

Problem Solving With Data Structures and Algorithms

December 19, 2024

Decoding the Relationships Between Genes: Final Computational Application

Introduction

This assignment revolves around analyzing and reconstructing genealogical relationships between a set of given gene sequences. Each gene is represented as a string of characters, and a series of mutations (insertion, deletion, or alteration of characters) leads from an ancestral “root” gene down to subsequent generations. Our goal is to use computational methods to both understand the similarities between these strings and to reconstruct a binary genealogical tree that represents parent-child relationships.

By the end of this assignment, we will have a fully implemented solution that computes all LCSs for any two strings, provides a matrix of LCS lengths for the entire set of given gene strings, and generates binary genealogical trees using both local and global inference strategies. We will analyze and compare the results of these strategies, their complexities, and their alignment (or lack thereof) with expected genealogical relationships.

Longest Common Subsequences and Lengths

The Longest Common Subsequence (LCS) problem is about finding sequences that appear in order (not necessarily contiguously) inside two given strings. If there are multiple longest subsequences with the same length, we want to find them all. For example, if we compare the strings "ABCBDAB" and "BDCABA", there are three different subsequences of length 4 that appear in both: "BCAB", "BDAB", and "BCBA". We want to list all of these.

With our code (See Appendix A), we use a dynamic programming approach to fill out a table c that records the length of the LCS for every prefix of x and y. The following test cases—in which all passed—were used to prove the code works:

```
x1, y1 = 'ABCBDAB', 'BDCABA'
x2, y2 = 'abc', ''
x3, y3 = 'abc', 'a'
x4, y4 = 'abc', 'ac'
x6, y6 = "HELLO", "HELLOWORLD"
x7, y7 = "ABC", "ABC"

#Assertions to verify correctness of the implementation
expected_x1_y1 = sorted(['BCAB', 'BDAB', 'BCBA'])
assert longest_common_subsequences(x1, y1) == (expected_x1_y1, 4)  #Test with multiple LCS
assert longest_common_subsequences(x2, y2) == (None, 0)  #Test with an empty string
assert longest_common_subsequences(x3, y3) == (['a'], 1)  #Test with a single matching character
assert longest_common_subsequences(x4, y4) == (['ac'], 2)  #Test with partial matching
assert longest_common_subsequences(x6, y6) == (["HELLO"], 5)  #Test with overlapping subsequences
assert longest_common_subsequences(x7, y7) == (["ABC"], 3)  #Test with identical strings
```

By identifying all the LCS and their lengths, we can measure the degree of similarity between any two gene strings. This LCS-based similarity measure is needed for the final steps of the assignment, where we use these relationships to construct and compare different genealogical trees.

LCS Length Matrix for set_strings

With the code in Appendix B, we produced our 7x7 matrix with the LCS length. Multiplying 7x7, our matrix dimensions, we get 49 LCS lengths in total.

$$\begin{bmatrix} 265 & 235 & 199 & 252 & 214 & 251 & 212 \\ 235 & 250 & 211 & 223 & 227 & 223 & 220 \\ 199 & 211 & 257 & 197 & 234 & 195 & 229 \\ 252 & 223 & 197 & 282 & 207 & 241 & 205 \\ 214 & 227 & 234 & 207 & 252 & 205 & 243 \\ 251 & 223 & 195 & 241 & 205 & 280 & 208 \\ 212 & 220 & 229 & 205 & 243 & 208 & 269 \end{bmatrix}$$

I) Which strings are more strongly related to each other?

If two strings share a very long LCS relative to their lengths, it suggests they are very similar and possibly closely related. In other words, a higher LCS length indicates that the two sequences have retained more ancestral similarity and undergone fewer divergent mutations. So, by examining the LCS length values, larger LCS lengths between a pair of strings indicate a stronger similarity (closer evolutionary relationship), and smaller LCS lengths suggest more distant relationships or more mutation events separating them. One point is that to identify which pair of strings in the LCS matrix is the "strongest" (most closely related), we need to look at the off-diagonal entries (since the diagonal values represent the LCS of each string with itself).

$$\begin{bmatrix} \cancel{265} & 235 & 199 & 252 & 214 & 251 & 212 \\ 235 & \cancel{250} & 211 & 223 & 227 & 223 & 220 \\ 199 & 211 & \cancel{257} & 197 & 234 & 195 & 229 \\ 252 & 223 & 197 & \cancel{282} & 207 & 241 & 205 \\ 214 & 227 & 234 & 207 & \cancel{252} & 205 & 243 \\ 251 & 223 & 195 & 241 & 205 & \cancel{280} & 208 \\ 212 & 220 & 229 & 205 & 243 & 208 & \cancel{269} \end{bmatrix}$$

For instance, when we see our matrix, we notice that the strongest relationship is 252, which occurs at the positions:

- [0, 3] (row 0, column 3)
- [3, 0] (row 3, column 0)

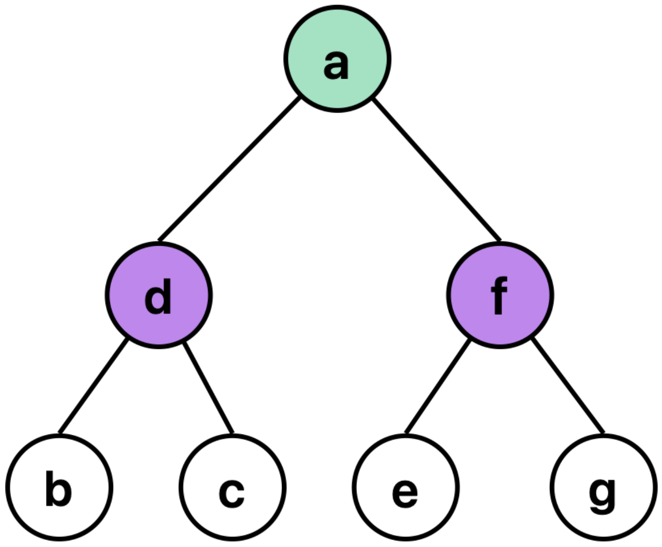
So, the strings a (index 0) and d (index 3) are the most closely related pair, as they share the largest LCS length of 252.

~~265~~	235	199	252	214	251	212
235	~~250~~	211	223	227	223	220
199	211	~~257~~	197	234	195	229
252	223	197	~~282~~	207	241	205
214	227	234	207	~~252~~	205	243
251	223	195	241	205	~~280~~	208
212	220	229	205	243	208	~~269~~

II) Could we infer the resulting genealogy binary tree?

The relationships indicated by the LCS matrix suggest that the strongest connections (highest LCS values) correspond to closer genetic relationships, such as parent-child or sibling pairs, while lower LCS values suggest more distant relationships. This hierarchical grouping based on proximity allows us to construct a plausible binary tree reflecting the evolutionary genealogy of the strings. While this tree is not guaranteed to be unique, it represents one possible arrangement consistent with the LCS matrix.

Starting with the strongest relationship, the pair (a, d)[0, 3] has the highest off-diagonal LCS value of 252, suggesting that these two strings are most closely related. This relationship implies that a and d should occupy adjacent positions in the tree, either as siblings or as parent and child. The next strongest relationship is between a and f [0 5], with an LCS value of 251, indicating that f is closely related to a and can be grouped within the same subtree. Similarly, e and g [4 7] share an LCS value of 243, indicating that these two strings form another cluster in the tree, possibly as siblings. To fully incorporate all strings into the tree, b and c [1 2], with an LCS value of 211, can be grouped together as another distinct cluster since their relationship is stronger than their relationships with other strings. Finally, the tree can be assembled by combining these clusters based on their relative relationships.



- (a) is chosen as the root because it has the strongest connections in the LCS matrix (with d and f being directly related to it).
- (d) and (f) are placed as direct children of (a) because of their high LCS values with (a).
- The remaining strings (b), (c), (e), and (g) are placed as children of (d) and (f) based on

their relative relationships:

- (b, c) Grouping:
    - The average LCS of (b, c) with d is 210.
    - The average LCS of (b, c) with f is 209.
    - Since the average with d is higher, (b, c) should remain grouped under d.
- (e, g) Grouping:
    - The average LCS of (e, g) with d is 206.
    - The average LCS of (e, g) with f is 206.5.
    - Since the average with f is higher, (e, g) should remain grouped under f.

Strategies Explanation Video

Decoding Genetics - CS110 Assignment.mov

Implementation of Local and Global Strategies

Expectations Prior Coding

Before implementing the local (greedy) and global (dynamic programming) strategies, we can anticipate that the trees will probably be different. The greedy approach focuses only on immediate similarities—each node picks the children that yield the highest LCS at that particular step. It does not reconsider these choices in light of future assignments. On the other hand, the global approach simultaneously considers all nodes and attempts to arrange them to maximize the overall total LCS, even if that means temporarily choosing children that might not yield the highest immediate LCS score.

As a result, we expect the greedy tree and the global optimization tree to differ, because one optimizes locally at each step while the other optimizes the entire structure holistically.

Local Strategy - Greedy Approach

With the greedy approach, at each level of the tree, we assign the parent-child relationships by selecting the strongest immediate relationship (highest LCS value). This means a string is assigned as a child to the parent string with which it has the highest LCS value among its potential parents at that level. It focuses only on the most local and immediate relationships at each step, without considering the global tree structure. Also, it is greedy because the algorithm commits to decisions immediately, without backtracking or reconsideration.

For example, we start at the root (e.g., **a**) and determine its children by comparing LCS values between **a** and the other strings. The two strings with the highest LCS values to **a** become its immediate children. (Full code on Appendix C)

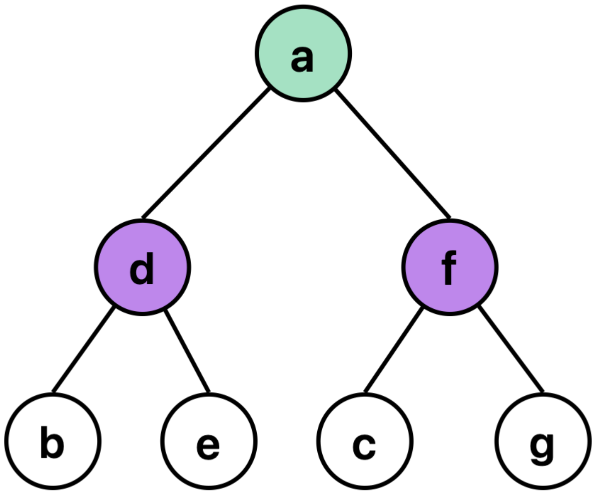
```
#Build the tree using a greedy approach
    queue = [root]  #Start with the root in the queue
    while queue:
        parent = queue.pop(0)  #Get the next node to process as the parent

        #Find the two nodes with the strongest connections (highest LCS values) to
the parent
        candidates = sorted(
            [(lcs_matrix[parent, i], i) for i in unassigned],  #Get LCS scores with
unassigned nodes
            reverse=True  #Sort by LCS value in descending order
        )[:2]  #Take the top two candidates
```

This section of the algorithm implements the greedy approach for assigning children to each parent node. By selecting the two nodes with the highest LCS values, the algorithm ensures that the parent is connected to the most strongly related nodes first, without considering the impact on the rest of the tree.

This is the result of the greedy algorithm:

```
{'a': ['d', 'f'], 'd': ['b', 'e'], 'f': ['g', 'c'], 'b': [], 'e': [], 'g': [], 'c': []}
```

The term on the outside is the parent, and the ones inside are the children. Therefore, we get almost the same tree as the one in question 2.II, but with ‘c’ and ‘e’ inverted.

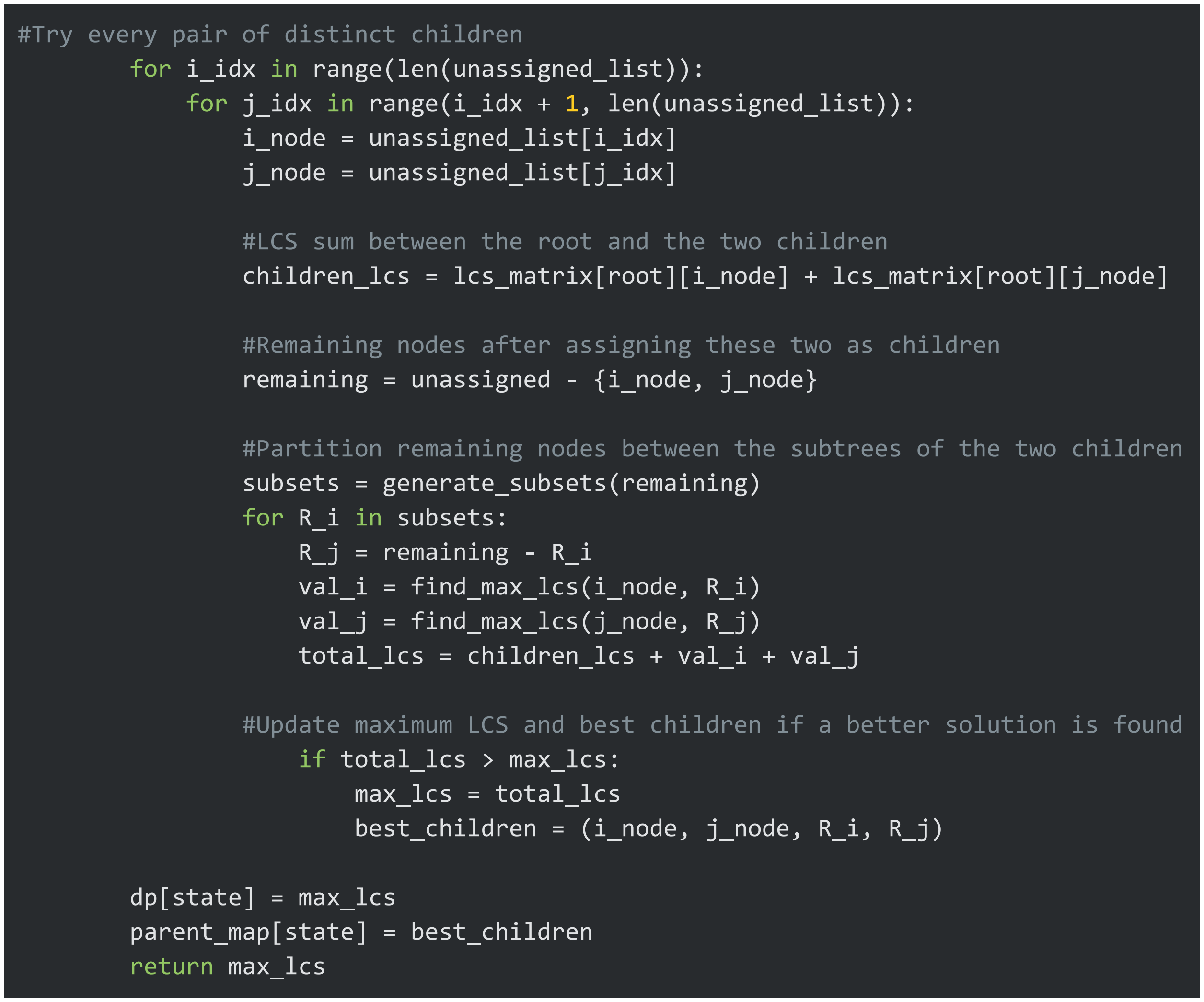


Global Strategy - Dynamic Programming

A global strategy considers all nodes and their relationships simultaneously to find the tree structure that maximizes an overall quality metric—here, the total LCS value across all parent-child edges. Unlike a greedy strategy that only focuses on the strongest immediate connections, the global (dynamic programming) approach tries every possible arrangement of nodes under each root candidate, exploring every combination of children and recursively subdividing the remaining nodes.

The total LCS metric sums up how similar each parent-child pair is, capturing the "global" coherence of the tree. Dynamic programming works by breaking the complex problem of building the entire tree into smaller subproblems: for each potential root node and set of remaining (unassigned) nodes, we compute the optimal subtree structure and the maximum possible total LCS. These subproblems are combined so that once we've solved all the smaller decisions (which children to pick, and how to partition the unassigned nodes for their subtrees), we have the best possible global solution. The DP ensures that we do not miss a more optimal arrangement deeper in the tree by only focusing on immediate, local improvements.

(Full code on Appendix D)



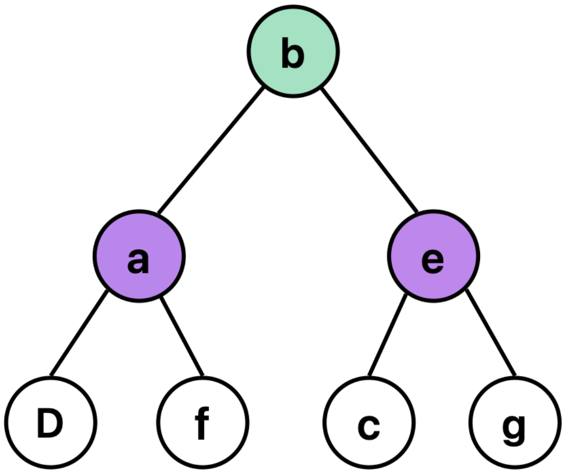
```
#Try every pair of distinct children
        for i_idx in range(len(unassigned_list)):
            for j_idx in range(i_idx + 1, len(unassigned_list)):
                i_node = unassigned_list[i_idx]
                j_node = unassigned_list[j_idx]

                #LCS sum between the root and the two children
                children_lcs = lcs_matrix[root][i_node] + lcs_matrix[root][j_node]

                #Remaining nodes after assigning these two as children
                remaining = unassigned - {i_node, j_node}

                #Partition remaining nodes between the subtrees of the two children
                subsets = generate_subsets(remaining)
                for R_i in subsets:
                    R_j = remaining - R_i
                    val_i = find_max_lcs(i_node, R_i)
                    val_j = find_max_lcs(j_node, R_j)
                    total_lcs = children_lcs + val_i + val_j

                #Update maximum LCS and best children if a better solution is found
                    if total_lcs > max_lcs:
                        max_lcs = total_lcs
                        best_children = (i_node, j_node, R_i, R_j)

        dp[state] = max_lcs
        parent_map[state] = best_children
        return max_lcs
```

In this process, the algorithm:

- Considers each node as a potential root.
- For that root, tries all possible pairs of nodes as children.
- For each pair of children, divides the remaining nodes between these children's subtrees.
- Recursively computes the maximum total LCS achievable from those subtrees.
- Selects the configuration that yields the highest overall total LCS.

By storing (memoizing) results for each combination of "root" and "unassigned nodes," the algorithm avoids redundant calculations and systematically explores the entire solution space. Ultimately, this DP approach yields the tree that best reflects the global relationships between the genes. This is the result from the dynamic programming approach:

```
{'b': ['a', 'e'], 'a': ['d', 'f'], 'd': [], 'f': [], 'e': ['c', 'g'], 'c': [], 'g': []}
```



<u>After Coding Comparison</u>

After implementing both strategies, we confirmed that the resulting trees are indeed different. The greedy method produced a tree that closely follows local maxima, linking parents to children that appear most similar at the time, but not necessarily fitting perfectly into the larger genealogical structure. In contrast, the DP approach generated a tree that maximizes the total LCS across all branches, potentially selecting pairs of nodes that do not look like the best short-term match but ultimately form a more coherent global arrangement.

For example, the greedy tree might have siblings that have strong LCS matches to their parent but do not lead to the optimal overall total LCS across the entire tree. The global tree, on the other hand, ensures that the sum of all parent-child relationships is maximized, leading to a different configuration. This discrepancy is precisely because the global method looks beyond immediate neighbors and incorporates the relationships of all remaining nodes, considering all possible subtree configurations before settling on a final, globally optimal solution.

To decide which strategy produces the "best" results, we can directly compare their total LCS values. Logically, the global is supposed to be the best one, but comparing to prove, we get:

Local Approach	Global Approach
AD: 252	BA: 235
AF: 251	BE: 227
DB: 223	AD: 252
DE: 207	AF: 251
FC: 195	EC: 234
FG: 208	EG: 243
Total LCS Score: 1336	Total LCS Score: 1442

With this, we can confirm that while the greedy strategy might be simpler and faster to implement, it may not capture the true genealogical structure as effectively as the global approach.

Computational Complexity to Produce Genealogy Binary Trees

Theoretical Analysis

Building the LCS Length Matrix (len_lcs_matrix)

We have N gene strings, each of length M. Computing the LCS for two strings of length M typically requires $O(M^2)$ time using the standard dynamic programming approach (filling a M x M table). Since we need to build a full N x N matrix of LCS lengths, we perform this $O(M^2)$ computation for every pair of strings (including comparing each string with itself). There are N^2 such pairs. So, the complexity for building the len_lcs_matrix is $O(N^2 * M^2)$.

Greedy Approach

After we have the len_lcs_matrix, the greedy approach selects a root and then iteratively assigns children with the highest LCS values. Each assignment involves sorting the unassigned nodes by their LCS value to the parent.

In the worst case, at each node, we remove two nodes from the unassigned set and move on. We perform a sorting step at each assignment. At the start, we sort N-1 nodes, then N-3, then N-5, and so forth. Sorting takes O(N log N) in the worst step, and repeated across the entire tree building leads approximately to $O(N^2 \log N)$ complexity.

However, compared to building the matrix, this might be negligible for large M or may be overshadowed by the LCS computations. Since the matrix is already computed, and N is smaller relative to M in some contexts, the dominating factor might still be the matrix computation. If we consider the matrix building done, the greedy tree-building step is roughly $O(N^2 \log N)$ in the worst case.

Total for Greedy Approach (including matrix construction): $O(N^2 * M^2 + N^2 \log N)$. For large M, $O(N^2 * M^2)$ dominates. For large N with fixed M, both $N^2 * M^2$ and $N^2 \log N$ matter, but $N^2 * M^2$ will usually be larger since M^2 can be large for long genes.

<u>Global (DP) Approach</u>

The global approach uses a more complex DP that tries all possible pairings and partitions of the unassigned set. Its complexity is much higher due to the combinatorial nature of partitioning sets among child subtrees. At each state, defined by (root, unassigned_set), we consider every children's pair ($O(N^2)$ in the worst case). For each pair, we consider all remaining nodes partitions. The number of subsets of a set size k is 2^k, so it is exponential in nature.

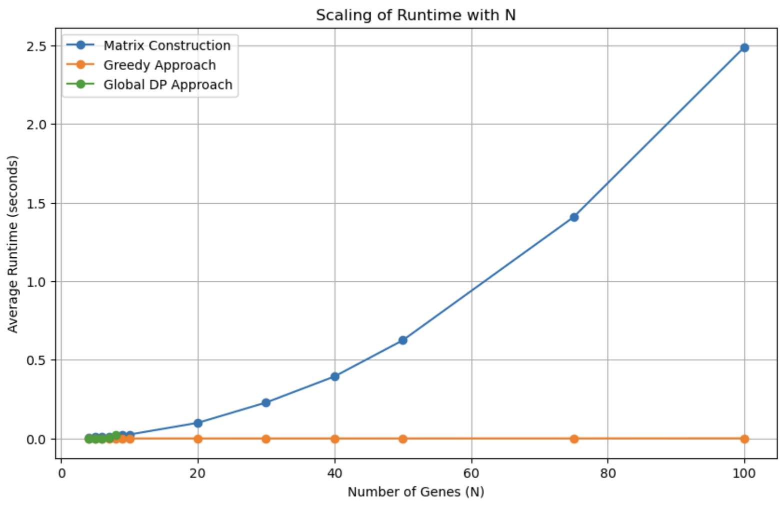
Roughly, the global approach has a complexity on the order of $O(N^2 * 2^N)$ for the subtree assignments, not counting the LCS computations. When integrated with the LCS computations (already done once in the matrix), the matrix building is still $O(N^2 * M^2)$. The global tree construction step dominates for larger N, becoming extremely expensive. This method becomes practically infeasible as N grows.

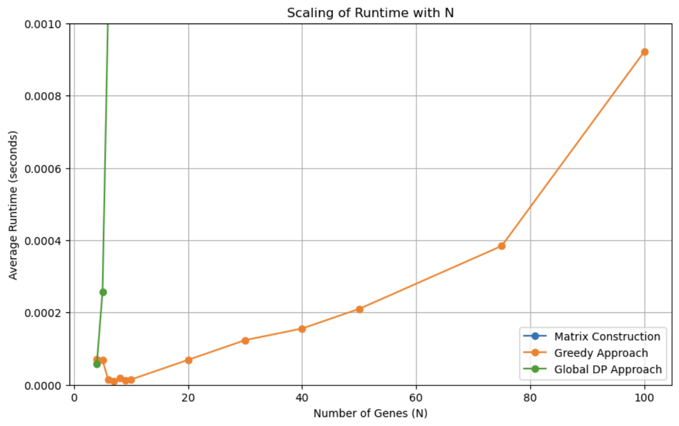
Total for Global DP Approach (including matrix construction): $O(N^2 * M^2) + O(N^2 * 2^N)$ (The exponential term eventually dominates the polynomial terms when N grows). As N grows large, the global DP approach becomes intractable. For practical sizes, it might still run, but it scales very poorly.

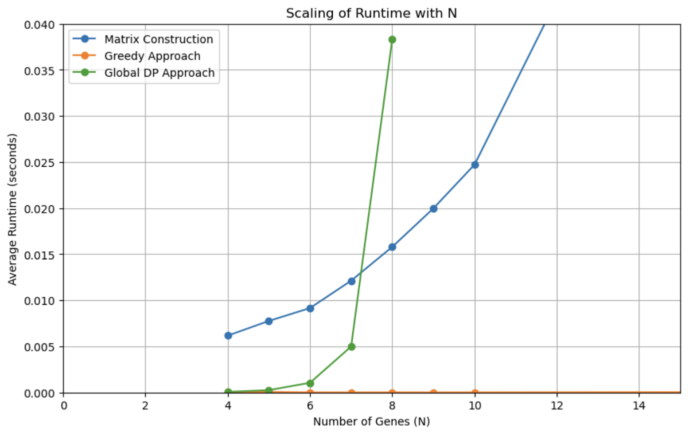
In the end, if we don't consider the matrix building, greedy has O(N^2 log N) and global O(N^2 * 2^N).

Analytical Analysis

To prove this, we can first plot the graph of the number of genes by the average runtime (same graph, just zoomed in) (See Appendices E, F, and G for the codes).







The plotted runtimes go with the theoretical complexities we discussed, though the exact curves may not be perfectly smooth due to randomness and overhead.

Matrix Construction

As N grows, the runtime increases roughly as $O(N^2)$ (since M is fixed). In the plot, as N doubles, the matrix build time should roughly quadruple, which is what we'd expect. To provide a mathematical confirmation that the matrix construction scales approximately as $O(N^2)$, we can use the approach of comparing the runtime at two different input sizes and checking how the runtime changes relative to the size increase.

Suppose we pick two points for the number of genes, N_1 and N_2, and look at their corresponding matrix construction runtimes $T(N_1)$ and $T(N_2)$.

For instance, from the graph:

- At $N_1 = 50$, let's say the measured average runtime is $T(50) \approx 0.6$ seconds.
- At $N_2 = 100$, the measured average runtime is $T(100) \approx 2.5$ seconds.

Going from N=50 to N=100, we are doubling N. Mathematically: $\frac{N_2}{N_1} = \frac{100}{50} = 2$. If the runtime truly scales as $O(N^2)$, then when we double N, we expect the runtime to multiply by a factor of (N_2^2 / N_1^2). Since $N_2 = 2 * N_1$, this factor should be 4. Checking with our data, we see that it matches: $\frac{2.5}{0.6} \approx 4$. The measured runtime at N=100 is about 4 times the runtime at N=50, which is exactly what $O(N^2)$ scaling would predict for a doubling of N.

Greedy Approach

The greedy method shows relatively minor growth compared to the matrix construction. It stays near zero or very low because sorting N nodes and selecting children is inexpensive relative to the $O(N^2 * M^2)$ matrix computation. Even though we know it's $O(N^2 \log N)$, the scale and overhead make it look almost constant compared to matrix building.

Performing the ratio test, we take the points:

- At $N_1 = 50$, $T(50) \approx 0.0002$ seconds
- At $N_2 = 100$, $T(100) \approx 0.0009$ seconds

We have doubled N from 50 to 100 ($\frac{N_2}{N_1} = 2$). Doubling N should roughly multiply the runtime by a factor of about $4 * \frac{\log N_2}{\log N_1}$). This is true because $\frac{T(N_2)}{T(N_1)} \approx \frac{N_2^2 \log(N_2)}{N_1^2 \log(N_1)}$.

For $N_1=50$ and $N_2=100$:

- $\log_{10}(50) \approx 1.699$
- $\log_{10}(100) \approx 2$

So, when we plug back to $\frac{T(N_2)}{T(N_1)} \approx \frac{N_2^2 \log(N_2)}{N_1^2 \log(N_1)}$, we get $4 * \frac{2}{1.699} \approx 4 * 1.1776 \approx 4.7.$

And when we take the ratio from the points we have, we get that $\frac{T(100)}{T(50)} = \frac{0.0009}{0.0002} = 4.5$. The value is very close to the expected 4.7, which strongly supports the $O(N^2 \log N)$ growth rate for the greedy approach.

<u>Global DP Approach</u>

For small N, it runs quickly. But as N approaches 7 or 8, we see a dramatic increase, reflecting the combinatorial explosion ($O(N^2 * 2^N)$). Beyond that, it's no longer feasible, which aligns with our complexity analysis.

For the analysis, we can take the points:

- At $N_1 = 7$, $T(7) \approx 0.005$ seconds
- At $N_2 = 8$, $T(8) \approx 0.038$ seconds

From N=7 to N=8, the ratio is approximately:

- $\frac{T(8)}{T(7)} \approx \frac{8^2 * 2^8}{7^2 * 2^7} = \frac{65 * 256}{49 * 128} = \frac{16384}{6272} \approx 2.6$

Just from the exponential factor 2^N, going from N=7 to N=8 roughly doubles the 2^N portion, plus the slight increase in the polynomial factor (N^2) gives us around 2.6x expected growth.

Taking the actual increase:

- $\frac{T(8)}{T(7)} = \frac{0.038}{0.005} = 7.6$

The actual increase is even larger than the predicted 2.6x. This discrepancy is not unusual for very small sample sizes of N, as overheads and other factors can magnify differences. However, the key point is that a single increment in N caused more than a sevenfold increase in runtime. Such a dramatic jump from 7 to 8 strongly confirms an exponential-like growth pattern, consistent with O(N^2 * 2^N).

Interpretation

The empirical results and theoretical analyses both confirm that the three main components of the problem—matrix construction, greedy genealogy construction, and global DP-based genealogy construction—scale in dramatically different ways as the number of genes (N) increases.

The Greedy Approach offers a middle ground. It may not guarantee a globally optimal solution, but its runtime grows much more slowly and remains tractable for larger N. If the problem involves a large number of genes, the greedy method becomes a realistic choice for producing a genealogy in a reasonable time.

The Global DP Approach, while more accurate in theory, is only feasible for small N due to exponential blow-up. It can provide a globally optimal solution, but only if we are dealing with a very small dataset or are willing to accept extremely long runtimes.

Some possible improvements could be to first find ways to speed up the LCS calculations. Parallelization or string compression, for example, could reduce the cost. Any reduction here benefits both greedy and global strategies. The global DP approach could have heuristics or cropping methods to skip examining certain configurations. By limiting the state space or applying bounding rules (e.g., early cutoffs when the partial solution is already too costly), we might reduce the effective complexity, making the global approach slightly more practical.

Instead of a purely local greedy or a fully global DP strategy, a hybrid method could combine heuristics with some limited global optimization. For example, we could start with a greedy solution and then apply a limited DP refinement to improve the structure without exploring the entire exponential search space.

Probabilities of Insertions, Deletions, and Mutations

To calculate the probabilities, we assume we have the reconstructed genealogy tree, where each edge connects a parent string to its child. Each parent-child relationship represents one "generation" of mutations. By analyzing how the child differs from the parent, we can count how many insertions, deletions, and mutations occurred from one generation to the next.

From the final genealogy tree we inferred (using the global approach, as it is the one with the highest LCS value), we know which node is the parent and which are its children. For each edge in this tree, we have a pair (parent_seq, child_seq). To distinguish between insertions, deletions, and mutations, we need to align the two sequences in a way that reveals these. Once aligned, we compare the sequences character by character:

- Insertion: If the parent has a gap ('-') and the child has a character c at the same position, it means the child gained that character. This is an insertion.
- Deletion: If the child has a gap ('-') and the parent has a character c at that position, it means the child lost that character. This is a deletion.
- Mutation (Substitution): If both parent and child have different characters at the same position (e.g., 'A' in the parent and 'G' in the child), that's a mutation (substitution).

To estimate the probabilities, we can take the total positions for all parent-child pairs, and then use this to divide the total insertions, total deletions, and total substitutions each by the total positions. These ratios are not perfect probabilities in a strict statistical sense, but they give an intuitive estimate. With very few sequences, these numbers can be skewed. Still, it provides a rough idea.

Python Implementation

To do the implementation, I searched for different ways to align the strings, and the one mentioned on multiple websites, including the article on Biopython's website, *Pairwise Sequence Alignment*, mentioned the Needleman-Wunsch algorithm. So, I searched for different ways to implement it, and with Bioinformatica (2020) videos, I was able to adapt the code (See Appendix H).

From the computed probabilities, we see that insertion events (~**7.76%)** occur more frequently than deletions **(~3.49%)** and substitutions **(~2.89%)** in our small dataset. This suggests that, under the given alignment and inferred genealogy, children tend to gain extra characters slightly more often than they lose or alter existing ones. However, these numbers are not conclusive—they are based on a very limited set of sequences and a single inferred genealogy. With more extensive data and careful alignment parameters, these estimates might change. Still, it provides a preliminary insight that, in this particular scenario, insertions may have played a more significant role in the evolutionary divergence of these gene sequences than deletions or direct substitutions.

AI Note: I acknowledge that all of the work included in this workbook is my own, and I have not shared (or received) any working solutions with (or from) anyone, not even an AI engine or tool.

Resources

Bioinformatica. (2020, September 1). *Needleman-Wunsch algorithm for global alignment - Python* [Video]. YouTube. https://www.youtube.com/watch?v=um8h3P216Fk

Pairwise sequence alignment. (n.d.). Biopython. https://biopython.org/docs/dev/Tutorial/chapter_pairwise.html

Appendix A

```
def longest_common_subsequences(x, y):
    """
    Finds all of the Longest Common Subsequences (LCSs) of strings x and y and
their corresponding length.

    Parameters
    ----------
    x : str
        The first input string.
    y : str
        The second input string.

    Returns
    -------
    tuple
        A tuple (all_lcs_list, length_lcs), where:
        - all_lcs_list is a list of all LCS strings (no duplicates).
        - length_lcs is the integer length of each LCS.
    """
    m, n = len(x), len(y)

    #Edge case: if one string is empty, there is no common subsequence
    if m == 0 or n == 0:
        return (None, 0)

    #Initialize the DP table to store LCS lengths
    #c[i][j] will hold the length of the LCS of x[:i] and y[:j]
    c = [[0] * (n + 1) for _ in range(m + 1)]
    #Initialize the table to store the actual LCS sets
    #lcs_sets[i][j] will hold a set of all LCS for x[:i] and y[:j]
    lcs_sets = [[set() for _ in range(n + 1)] for _ in range(m + 1)]

    #Fill in the DP tables row by row
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if x[i - 1] == y[j - 1]:
                #Characters match, extend all LCS from the previous diagonal cell
                if c[i - 1][j - 1] == 0:
                    #If the previous diagonal cell has no LCS, start a new set
                    lcs_sets[i][j] = {x[i - 1]}
                else:
           #Append the matching character to all subsequences in lcs_sets[i-1][j-1]
                    lcs_sets[i][j] = {seq + x[i - 1] for seq in lcs_sets[i - 1][j -
1]}
                c[i][j] = c[i - 1][j - 1] + 1
            else:
                #Characters do not match, choose the direction of the longer LCS
                if c[i - 1][j] > c[i][j - 1]:
                    #Take the LCS from the cell above
                    c[i][j] = c[i - 1][j]
                    lcs_sets[i][j] = lcs_sets[i - 1][j]
                elif c[i - 1][j] < c[i][j - 1]:
                    #Take the LCS from the cell to the left
                    c[i][j] = c[i][j - 1]
                    lcs_sets[i][j] = lcs_sets[i][j - 1]
                else:
                    #If both directions have the same LCS length, merge the sets
                    c[i][j] = c[i - 1][j]
                    lcs_sets[i][j] = lcs_sets[i - 1][j].union(lcs_sets[i][j - 1])

    #Extract the final LCS length from the bottom-right cell of the DP table
    length_lcs = c[m][n]
    if length_lcs == 0:
        #If no common subsequence exists, return None and length 0
        return (None, 0)
    else:
        #Convert the set of LCS to a sorted list for consistent output
        all_lcs_list = sorted(lcs_sets[m][n])
        return (all_lcs_list, length_lcs)


#Test cases
x1, y1 = 'ABCBDAB', 'BDCABA'
x2, y2 = 'abc', ''
x3, y3 = 'abc', 'a'
x4, y4 = 'abc', 'ac'
x6, y6 = "HELLO", "HELLOWORLD"
x7, y7 = "ABC", "ABC"

#Assertions to verify correctness of the implementation
expected_x1_y1 = sorted(['BCAB', 'BDAB', 'BCBA'])
assert longest_common_subsequences(x1, y1) == (expected_x1_y1, 4)  #Test with
multiple LCS
assert longest_common_subsequences(x2, y2) == (None, 0)  #Test with an empty string
assert longest_common_subsequences(x3, y3) == (['a'], 1)  #Test with a single
matching character
assert longest_common_subsequences(x4, y4) == (['ac'], 2)  #Test with partial
matching
assert longest_common_subsequences(x6, y6) == (["HELLO"], 5)  #Test with
overlapping subsequences
assert longest_common_subsequences(x7, y7) == (["ABC"], 3)  #Test with identical
strings
```

```
print("All tests passed.")
```

Output:

```
All tests passed.
```

Appendix B

```
import numpy as np

def lcs_length(x, y):
    """
    Computes the length of the Longest Common Subsequence (LCS) between two
strings.

    Parameters
    ----------
    x : str
        The first input string.
    y : str
        The second input string.

    Returns
    -------
    int
        The length of the LCS.
    """
    m, n = len(x), len(y)

    #Initialize a DP table where c[i][j] stores the LCS length for x[:i] and y[:j]
    c = [[0] * (n + 1) for _ in range(m + 1)]

    #Fill the DP table row by row
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if x[i - 1] == y[j - 1]:
                #If characters match, add 1 to the LCS length from the diagonal
cell
                c[i][j] = c[i - 1][j - 1] + 1
            else:
                #If no match, take the maximum value from the top or left cell
                c[i][j] = max(c[i - 1][j], c[i][j - 1])
    #Return the LCS length for the full strings
    return c[m][n]

#The strings:
a =
'ATGGTGCGAAAGCATCTCTTTTCGTGGCGTGATAAGTTTTATGGTATCCCCGGACGTTGGCTACTACAATTCTCCGAAGTAT
AAGTGAGTAGGATATGTCAATAACAAGA' \

'GGGGATGCGTGACGCATTAGCACCAACTGAATCAAACGATAACTAACGTGGTTTCAGTGAGCGTATGTGGCAAAGGATTGGA
TACATTTTTCGAGCACGTCTACATAATGA' \
    'CCGTGACAATACTGGAGACTCCGTACCGTCATCTTGACACTCCT'

b =
'TGGTGCGAAAGCATCTCTTTTCCGTGGCGTATAGTTTTATGGTATCCCCGGAACGCTGGCTACTACAATCTCCGAAGTATAG
AGTGAGTAGATTTAATTAACAGAGGGCG' \

'TCGTTGACGCATTAGCACCAACTGAATCAACCGATAACTTAACGTGGGTTTCAGTGACTATAGGGCAAAGGATGAACATTTT
CGAGCAGCTCTAATAATGAGCGTGACAAT' \
    'ATGAATCCACACCGTCATCTTGAACTCCT'

c =
'TCTGTGCGATATACATCTCTATCGTTGCGGTATGTTTTATGTGCATCACCCCACGCGCTGGCTACAGTACAATCTGCTGGAA
GTACTAGGTGGTAGTTAATAACTAGGGT' \

'GCGTCGTTGCGCATTACACAACTGGACAACCACTTAACTGGGGTAATCAGTGTTTAGGGCAGACAAGATGAAAACAAGTTTT
CGAGCAGGCTCCTATAATGAGGACGGAAC' \
    'GTTAATAAATCCAACACCGCACTGCTTCGTAACCCT'

d =
'ATGAGGCGCAAAATTCTCTTTCTCGTGGCGCTGATTAAGTTTTATGTATCCCCGGACGTTGGCTACTGACAATTGCTCCGAA
GTATAAAGTAGTAGGATATGTCAATAAC' \

'AAAGACGGGGATAGCGTGACAGCATTAGAACGCAACTGGAATCAAACGTAACCTAAAGGGTTGTCAGGAGCGTATGTGGTCA
AAAAGGATTGGATGACATTTTTCGACACG' \
    'TCTACATAATGACCTGTGACAAACTAGGAGACCTCCTACTCGGTCAATCTTGACGACTCCT'

e =
'TGGTGCGATATACATCTCTTTTCGTGCGTATGTTTTATGGTGATCACCCGGAACCGCTGGCTACATACAATCTCTGGAAGTA
CTAGGTGGTAGTTTAATAACTAGAGGTG' \

'CGTCGTTGACGCATTACACAACTGGATCAACCGAACTTAACTGGGTATCAGTGATATAGGGCGACAAGATGAACAATTTTCG
AGCAGCTCCTGAATAATGAGACGGAACGT' \
    'ATAATCCAACACCGTCACTGCTTCGAACCCT'

f =
'GGGGGAAAGCGATCCCTTATCGTGGCTGTGATAAGTTTTTATCGGGTATCCGCCGGACGTTGGCGTACTACAATTCTCCGAA
GTTAAGTGAGTTAGGGATATAGTCAATA' \
'ACAAGAGGGGATTGTCGTGACGCATAGCACACAACTGAATCAAATCGATAACTAAACGGGTTTCAGTAGAGCGTTGTGGCAA
AGATTGGATACATTTTTCGCAGGACGTCT' \
    'TACCTAATGACGTGGACAATAACTGGCAGACGTCCGTACCGTCATCTTGACCACTCCCT'

g =
'TGGTGCGATATACATCCTCTTTTCGTGCGTATGTTTTAGGTACACCGGATACGCCTGGCTTACAAGTACCAATCTCTGAGAA
GTCACTGAGGTGGTAGTTTAATAACTAG' \

'AAGGGTGCGTCGGACGCATTCACACATACTGGATCAACCGAGACTTAACTGGGGTATCAGTGATTGATAGGGCGACAAGATA
TACAATTTTCGAGCAGCTCCCTGAATAAG' \
    'TGAAGAACGGAGACGTATAATCCAACACGATTCACTGCTTCGAACCCT'

set_strings = [a, b, c, d, e, f, g]

#Create a 7x7 LCS length matrix
len_lcs_matrix = np.zeros((len(set_strings), len(set_strings)), dtype=int)

#Compute the LCS lengths for every pair of strings
for i in range(len(set_strings)):
    for j in range(len(set_strings)):
        len_lcs_matrix[i, j] = lcs_length(set_strings[i], set_strings[j])

print("LCS Length Matrix:\n", len_lcs_matrix)
```

Output:

```
LCS Length Matrix:
 [[265 235 199 252 214 251 212]
 [235 250 211 223 227 223 220]
 [199 211 257 197 234 195 229]
 [252 223 197 282 207 241 205]
 [214 227 234 207 252 205 243]
 [251 223 195 241 205 280 208]
 [212 220 229 205 243 208 269]]
```

Appendix C

```
def construct_tree_greedy(strings, lcs_matrix):
    """
    Constructs a binary tree using a greedy local strategy.
    Parameters
    ----------
    strings : list of str
        List of strings representing the nodes (e.g., [a, b, c, d, e, f, g]).
    lcs_matrix : numpy.ndarray
        A 2D numpy array where lcs_matrix[i][j] is the LCS length between
strings[i] and strings[j].

    Returns
    -------
    dict
        A dictionary representing the tree where keys are parent nodes (indices)
        and values are lists of child node indices.
    """
    n = len(strings)  #Total number of strings (nodes)
    tree = {}  #Dictionary to represent the tree structure
    unassigned = set(range(n))  #Set of nodes that are not yet part of the tree

    #Select the root node
    #Root is the node with the highest total LCS sum with all other nodes
    root = np.argmax(np.sum(lcs_matrix, axis=1))  #Find the index of the root
    tree[root] = []  #Initialize the root with no children
    unassigned.remove(root)  #Remove the root from the unassigned set

    #Build the tree using a greedy approach
    queue = [root]  #Start with the root in the queue
    while queue:
        parent = queue.pop(0)  #Get the next node to process as the parent

        #Find the two nodes with the strongest connections (highest LCS values) to
the parent
        candidates = sorted(
            [(lcs_matrix[parent, i], i) for i in unassigned],  #Get LCS scores with
unassigned nodes
            reverse=True  #Sort by LCS value in descending order
        )[:2]  #Take the top two candidates

        #Assign the top two candidates as children of the current parent
        children = [child for _, child in candidates]  #Extract the indices of the
children
        tree[parent] = children  #Add the children to the tree
        for child in children:
            unassigned.remove(child)  #Remove the children from the unassigned set
            queue.append(child)  #Add the children to the queue for further
processing

    #Handle any remaining unassigned nodes (if any)
    #These nodes are added as leaves with no children
    for node in unassigned:
        tree[node] = []  #Add remaining nodes with no children

    return tree


labels = ["a", "b", "c", "d", "e", "f", "g"]

#Construct the tree using the greedy approach
tree_greedy = construct_tree_greedy(set_strings, len_lcs_matrix)

#Map the indices in the tree back to their corresponding string labels for
readability
tree_greedy_labeled = {labels[parent]: [labels[child] for child in children] for
parent, children in tree_greedy.items()}

print(tree_greedy_labeled)
```

Output:

```
{'a': ['d', 'f'], 'd': ['b', 'e'], 'f': ['g', 'c'], 'b': [], 'e': [], 'g': [], 'c':
[]}
```

Appendix D

```
def construct_tree_global(strings, lcs_matrix):
    """
    Constructs a binary tree using a global optimization strategy to maximize the
total LCS.
    The approach uses dynamic programming to calculate the optimal way to divide
the tree,
    ensuring the maximum LCS sum across all nodes.

    Parameters
    ----------
    strings : list of str
        List of strings representing the nodes (e.g., [a, b, c, d, e, f, g]).
    lcs_matrix : numpy.ndarray
        A 2D numpy array where lcs_matrix[i][j] is the LCS length between
strings[i] and strings[j].

    Returns
    -------
    dict
        A dictionary representing the binary tree structure, where keys are parent
nodes
        (indices) and values are lists of child node indices.
    """

    n = len(strings)  #Number of nodes
    dp = {}  #Dictionary to store dynamic programming results
    parent_map = {}  #Map to store parent-child relationships for reconstruction

    def generate_subsets(elements):
        """
        Generate all subsets of a given set of elements.

        Parameters
        ----------
        elements : set
            The set of elements to generate subsets for.

        Returns
        -------
        list of set
            A list containing all subsets of the input set.
        """
        elements_list = list(elements)

        def backtrack(index, current_subset, all_subsets):
            #Base case: If we have considered all elements, add the current subset
            if index == len(elements_list):
                all_subsets.append(set(current_subset))
                return
            #Exclude the current element
            backtrack(index + 1, current_subset, all_subsets)
            #Include the current element
            current_subset.append(elements_list[index])
            backtrack(index + 1, current_subset, all_subsets)
            #Backtrack to remove the last added element
            current_subset.pop()

        all_subsets = []
        backtrack(0, [], all_subsets)
        return all_subsets

    def find_max_lcs(root, unassigned):
        """
        Recursively calculates the maximum total LCS for a subtree rooted at
```

```
'root',
        given a set of unassigned nodes.

        Parameters
        ----------
        root : int
            The index of the current root node.
        unassigned : set
            The set of indices for unassigned nodes.

        Returns
        -------
        int
            The maximum total LCS for the subtree rooted at 'root'.
        """
        state = (root, frozenset(unassigned))  #State is a tuple of root and
unassigned nodes
        if state in dp:
            return dp[state]

        #Base case: If fewer than 2 nodes remain, this node becomes a leaf
        if len(unassigned) < 2:
            dp[state] = 0
            parent_map[state] = None
            return 0

        max_lcs = 0  #To track the maximum LCS
        best_children = None  #To track the best pair of children

        #Convert unassigned set to a list for iteration
        unassigned_list = list(unassigned)

        #Try every pair of distinct children
        for i_idx in range(len(unassigned_list)):
            for j_idx in range(i_idx + 1, len(unassigned_list)):
                i_node = unassigned_list[i_idx]
                j_node = unassigned_list[j_idx]

                #LCS sum between the root and the two children
                children_lcs = lcs_matrix[root][i_node] + lcs_matrix[root][j_node]

                #Remaining nodes after assigning these two as children
                remaining = unassigned - {i_node, j_node}

                #Partition remaining nodes between the subtrees of the two children
                subsets = generate_subsets(remaining)
                for R_i in subsets:
                    R_j = remaining - R_i
                    val_i = find_max_lcs(i_node, R_i)
                    val_j = find_max_lcs(j_node, R_j)
                    total_lcs = children_lcs + val_i + val_j

                    #Update maximum LCS and best children if a better solution is
found
                    if total_lcs > max_lcs:
                        max_lcs = total_lcs
                        best_children = (i_node, j_node, R_i, R_j)

        dp[state] = max_lcs
        parent_map[state] = best_children
        return max_lcs

    #Try each node as the root to find the optimal tree
    global_max_lcs = 0  #To store the maximum LCS across all roots
    best_root = None
    best_state = None
    all_nodes = set(range(n))
    for root in range(n):
        remaining = all_nodes - {root}
        val = find_max_lcs(root, remaining)
        if val > global_max_lcs:
            global_max_lcs = val
            best_root = root
            best_state = (root, frozenset(remaining))

    #Reconstruct the tree using the parent_map
    tree = {}

    def build_tree(root, unassigned):
        """
        Reconstructs the tree by traversing the parent_map.

        Parameters
        ----------
        root : int
            The current root node.
        unassigned : set
            The set of unassigned nodes.
        """
        state = (root, frozenset(unassigned))
        if parent_map[state] is None:
            tree[root] = []  #Leaf node
            return
        i_node, j_node, R_i, R_j = parent_map[state]
        tree[root] = [i_node, j_node]  #Assign children
        build_tree(i_node, R_i)  #Recursively build left subtree
        build_tree(j_node, R_j)  #Recursively build right subtree

    if best_root is not None:
        remaining = all_nodes - {best_root}
        build_tree(best_root, remaining)
    else:
        #Edge case: No valid tree found, construct a trivial tree
        tree[0] = []

    return tree


labels = ["a", "b", "c", "d", "e", "f", "g"]

#Construct the tree using the global optimization approach
tree_global = construct_tree_global(set_strings, len_lcs_matrix)

#Map the indices in the tree back to their corresponding string labels for
readability
tree_global_labeled = {labels[parent]: [labels[child] for child in children] for
parent, children in tree_global.items()}

print(tree_global_labeled)
```

Output:

```
{'b': ['a', 'e'], 'a': ['d', 'f'], 'd': [], 'f': [], 'e': ['c', 'g'], 'c': [], 'g':
[]}
```

Appendix E

```
import time
import matplotlib.pyplot as plt
import random

def random_gene(length=100):
    """
    Generates a random DNA sequence of a specified length.

    Parameters
    ----------
    length : int, optional
```

```
        The length of the random gene string to generate, by default 100.

    Returns
    -------
    str
        A random string of length 'length' consisting of characters 'A', 'C', 'G',
and 'T'.
    """
    #We use random.choices to generate a random sequence of length 'length'
    return ''.join(random.choices('ACGT', k=length))

def build_matrix(strings):
    """
    Builds an LCS length matrix for a given set of strings.

    Parameters
    ----------
    strings : list of str
        List of DNA strings for which to compute the LCS length matrix.

    Returns
    -------
    numpy.ndarray
        A 2D numpy array where element (i, j) is the LCS length between strings[i]
and strings[j].
    """
    n = len(strings)  #Number of strings
    matrix = np.zeros((n, n), dtype=int)  #Initialize an n x n matrix with zeros

    #Compute LCS length for each pair of strings
    for i in range(n):
        for j in range(n):
            matrix[i, j] = lcs_length(strings[i], strings[j])  #Compute LCS length
    return matrix

def measure_runtime(N, M, trials=5):
    """
    Measures the average runtime for building the LCS matrix, constructing a greedy
tree,
    and constructing a globally optimized tree.

    Parameters
    ----------
    N : int
        Number of gene strings to generate.
    M : int
        Length of each gene string.
    trials : int, optional
        Number of trials to average the runtime, by default 5.

    Returns
    -------
    tuple
        A tuple (avg_matrix, avg_greedy, avg_global), where:
        - avg_matrix : float
            Average runtime for building the LCS matrix.
        - avg_greedy : float
            Average runtime for the greedy tree construction.
        - avg_global : float
            Average runtime for the global tree construction, or NaN if not
computed.
    """
    times_matrix = []  #To store runtimes for matrix construction
    times_greedy = []  #To store runtimes for greedy tree construction
    times_global = []  #To store runtimes for global tree construction

    for _ in range(trials):
        #Generate N random strings of length M
        strings = [random_gene(M) for _ in range(N)]

        #Measure runtime for building the LCS matrix
        start = time.time()
        lcs_mat = build_matrix(strings)
        end = time.time()
        times_matrix.append(end - start)  #Record runtime

        #Measure runtime for greedy tree construction
        start = time.time()
        construct_tree_greedy(strings, lcs_mat)
        end = time.time()
        times_greedy.append(end - start)  #Record runtime

        #Measure runtime for global tree construction (only for small N)
        if N <= 8:  #Global tree construction is infeasible for large N due to
exponential complexity
            start = time.time()
            construct_tree_global(strings, lcs_mat)
            end = time.time()
            times_global.append(end - start)  #Record runtime
        else:
            times_global.append(np.nan)  #Append NaN for large N

    #Compute average runtimes
    avg_matrix = np.mean(times_matrix)
    avg_greedy = np.mean(times_greedy)
    avg_global = np.nanmean(times_global) if not np.isnan(times_global).all() else
np.nan

    return avg_matrix, avg_greedy, avg_global

#Experiment
M = 50  #Length of each gene
N_values = [4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 75, 100]  #List of different N
values to test

matrix_times = []  #To store average runtimes for matrix construction
greedy_times = []  #To store average runtimes for greedy tree construction
global_times = []  #To store average runtimes for global tree construction

#Get runtimes for each value of N
for N in N_values:
    m_t, g_t, gl_t = measure_runtime(N, M, trials=5)  #Average over 5 trials
    matrix_times.append(m_t)
    greedy_times.append(g_t)
    global_times.append(gl_t)

#Plotting
plt.figure(figsize=(10, 6))
plt.plot(N_values, matrix_times, marker='o', label='Matrix Construction')
plt.plot(N_values, greedy_times, marker='o', label='Greedy Approach')
# Plot global approach runtimes (only valid points)
valid_indices = [i for i, val in enumerate(global_times) if not np.isnan(val)]
plt.plot([N_values[i] for i in valid_indices], [global_times[i] for i in
valid_indices], marker='o', label='Global DP Approach')

plt.xlabel('Number of Genes (N)')
plt.ylabel('Average Runtime (seconds)')
plt.title('Scaling of Runtime with N')
plt.legend()
plt.grid(True)
plt.show()
```

Appendix F

```
#Plotting zoomed in to check the global approach
plt.figure(figsize=(10,6))
plt.plot(N_values, matrix_times, marker='o', label='Matrix Construction')
plt.plot(N_values, greedy_times, marker='o', label='Greedy Approach')
valid_indices = [i for i, val in enumerate(global_times) if not np.isnan(val)]
```

```
plt.plot([N_values[i] for i in valid_indices], [global_times[i] for i in
valid_indices], marker='o', label='Global DP Approach')

plt.xlim(0, 15)     #This will show the x-axis from N=3 to N=8
plt.ylim(0, 0.04)    #This will show the y-axis from 0 to 10 seconds of runtime
plt.xlabel('Number of Genes (N)')
plt.ylabel('Average Runtime (seconds)')
plt.title('Scaling of Runtime with N')
plt.legend()
plt.grid(True)
plt.show()
```

Appendix G

```
#Plotting zoomed in to check the greedy approach
plt.figure(figsize=(10,6))
plt.plot(N_values, matrix_times, marker='o', label='Matrix Construction')
plt.plot(N_values, greedy_times, marker='o', label='Greedy Approach')
valid_indices = [i for i, val in enumerate(global_times) if not np.isnan(val)]
plt.plot([N_values[i] for i in valid_indices], [global_times[i] for i in
valid_indices], marker='o', label='Global DP Approach')

plt.ylim(0, 0.001)    #This will show the y-axis from 0 to 10 seconds of runtime
plt.xlabel('Number of Genes (N)')
plt.ylabel('Average Runtime (seconds)')
plt.title('Scaling of Runtime with N')
plt.legend()
plt.grid(True)
plt.show()
```

(Outputs for E, F, and G are the plots in text)

Appendix H

```
labels = ["a", "b", "c", "d", "e", "f", "g"]
label_to_index = {lab: i for i, lab in enumerate(labels)}  #Maps string labels to
their indices

#global tree result: {'b': ['a', 'e'], 'a': ['d', 'f'], 'd': [], 'f': [], 'e':
['c', 'g'], 'c': [], 'g': []}

def needleman_wunsch(seq1, seq2, match_score=0, mismatch_score=1, gap_score=1):
    """
    Performs global sequence alignment using the Needleman-Wunsch algorithm.
    Parameters
    ----------
    seq1 : str
        The first input sequence to align.
    seq2 : str
        The second input sequence to align.
    match_score : int, optional
        The score for a match between characters, by default 0.
    mismatch_score : int, optional
        The penalty for a mismatch between characters, by default 1.
    gap_score : int, optional
        The penalty for introducing a gap in the alignment, by default 1.

    Returns
    -------
    tuple
        A tuple containing two aligned sequences (aligned1, aligned2).
    """
    m, n = len(seq1), len(seq2)
    #Initialize the DP score matrix
    score = [[0] * (n + 1) for _ in range(m + 1)]

    #Fill the first row and column with gap penalties
    for i in range(1, m + 1):
        score[i][0] = score[i - 1][0] + gap_score
    for j in range(1, n + 1):
        score[0][j] = score[0][j - 1] + gap_score

    #Fill the rest of the DP table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            char1 = seq1[i - 1]
            char2 = seq2[j - 1]
            cost_match = score[i - 1][j - 1] + (0 if char1 == char2 else
mismatch_score)  #Cost for match/mismatch
            cost_del = score[i - 1][j] + gap_score  #Cost for deletion
            cost_ins = score[i][j - 1] + gap_score  #Cost for insertion
            score[i][j] = min(cost_match, cost_del, cost_ins)  #Take the minimum
cost

    #Traceback to reconstruct the alignment
    aligned1 = []
    aligned2 = []
    i, j = m, n
    while i > 0 and j > 0:
        char1 = seq1[i - 1]
        char2 = seq2[j - 1]
        cost_match = score[i - 1][j - 1] + (0 if char1 == char2 else
mismatch_score)
        cost_del = score[i - 1][j] + gap_score
        cost_ins = score[i][j - 1] + gap_score

        if score[i][j] == cost_match:  #Match/mismatch
            aligned1.append(char1)
            aligned2.append(char2)
            i -= 1
            j -= 1
        elif score[i][j] == cost_del:  #Deletion
            aligned1.append(char1)
            aligned2.append('-')
            i -= 1
        else:  # Insertion
            aligned1.append('-')
            aligned2.append(char2)
            j -= 1

    #Handle remaining characters in seq1 or seq2
    while i > 0:
        aligned1.append(seq1[i - 1])
        aligned2.append('-')
        i -= 1
    while j > 0:
        aligned1.append('-')
        aligned2.append(seq2[j - 1])
        j -= 1

    #Reverse to get the correct alignment order
    aligned1.reverse()
    aligned2.reverse()
    return "".join(aligned1), "".join(aligned2)

def estimate_mutation_probabilities(tree, strings):
    """
    Estimates insertion, deletion, and substitution probabilities from a
genealogical tree.

    Parameters
    ----------
    tree : dict
        A dictionary {parent_label: [child_labels]} representing the genealogical
tree.
    strings : list of str
        List of DNA sequences corresponding to the nodes in the tree.
    Returns
    -------
    tuple
        A tuple (p_ins, p_del, p_sub) where:
        - p_ins : float
            Probability of an insertion.
        - p_del : float
            Probability of a deletion.
        - p_sub : float
            Probability of a substitution.
    """
    total_insertions = 0  #Count of insertions across all alignments
    total_deletions = 0  #Count of deletions across all alignments
    total_substitutions = 0  #Count of substitutions across all alignments
    total_positions = 0  #Total positions considered in all alignments

    #Iterate through the tree to align parent-child pairs
    for parent_label, child_labels in tree.items():
        parent_idx = label_to_index[parent_label]
        parent_seq = strings[parent_idx]  #Retrieve the parent sequence
        for c_label in child_labels:
            child_idx = label_to_index[c_label]
            child_seq = strings[child_idx]  #Retrieve the child sequence

            #Perform alignment using Needleman-Wunsch
            aligned_parent, aligned_child = needleman_wunsch(parent_seq, child_seq)

            #Compare aligned sequences to count mutations
            for p_char, c_char in zip(aligned_parent, aligned_child):
                total_positions += 1
                if p_char == c_char:
                    continue  # No mutation
                elif p_char == '-' and c_char != '-':
                    total_insertions += 1  #Count insertion
                elif c_char == '-' and p_char != '-':
                    total_deletions += 1  #Count deletion
                else:
                    total_substitutions += 1  #Count substitution

    #If no positions were aligned, return zero probabilities
    if total_positions == 0:
        return (0, 0, 0)

    #Calculate probabilities
    p_ins = total_insertions / total_positions
    p_del = total_deletions / total_positions
    p_sub = total_substitutions / total_positions
    return p_ins, p_del, p_sub


tree_global_labeled = {'b': ['a', 'e'], 'a': ['d', 'f'], 'd': [], 'f': [], 'e':
['c', 'g'], 'c': [], 'g': []}
p_insert, p_delete, p_mut = estimate_mutation_probabilities(tree_global_labeled,
set_strings)

print("Estimated Probabilities:")
print("Insertion:", p_insert)
print("Deletion:", p_delete)
print("Mutation:", p_mut)
```