Project 2 - Final Computational Application

Minerva University

CS110 - Problem-Solving with Data Structures and Algorithms

December 19, 2024

The Longest Common Subsequences (LCSs)

The objective is to compute **all** the possible Longest Common Subsequences between any two arbitrary strings and return their lengths. I used Dynamic Programming (DP) to solve the problem efficiently. I think DP works best here specifically because of its properties: *overlapping subproblem* and *optimal substructure*. For instance, for strings x = "ABCBDAB" and y = "BDCABA," one of the possible subsequences is "BCAB." To arrive at that conclusion, we compare each character of the first string with each character of the second string. Thus, we can break down the problem into smaller parts and solve them repeatedly. I used DP tables to avoid solving the same problem multiple times (comparing the same characters). We create a grid with **m** and **n** being the lengths of two strings, and then we start from the top left and compare each pair of characters. The algorithm creates a 2D DP table where **dp[i][j]** is the string length between the first characters of two strings. The table is filled iteratively where the LCS length is increased by 1 when the characters match: **dp[i][j] = dp[i-1][j-1] +1**. Otherwise, we look at one character from each string at a time to find their maximum: **dp[i][j] = max(dp[i-1][j], dp[i][j-1])**. Taking the maximum of two options allows the algorithm to proceed with the longest subsequence.

After filling the table, it backtracks from the bottom-right corner to ensure that all possible paths lead to the longest subsequences. A set is used to avoid duplicates because we reverse the results at the end. Thus, the solution exhibits optimal substructure as the main problem depends on the solution to its smaller problems. By storing intermediate values in the table, the algorithm avoids unnecessary computations.

Input String x	Input String x	Expected LCS	LCS Length
ABCBDAB	BDCABA	['BCAB', 'BCBA', 'BDAB']	4
abc	' '	None	0
abc	a	['a']	1
abc	ac	['ac']	2

Table 1. Test Cases to ensure that the LCS code works.

I included the given test cases to ensure that the code works according to and the output is the list of all the possible LCS and the corresponding length (Table 1). All test cases passed (Appendix).

Matrix of the LCS lengths

Now, we need to compute the pairwise LCS lengths for a given set of strings and analyze the matrix to identify relationships between strings. I will generate a symmetric **7x7** matrix because there are 7 strings in the dataset, and we want to explore the relationships between each of them. To obtain a symmetric matrix, I build on the previous code (longest_common_subsequence) to iterate over all pairs of strings in the dataset and calculate the LCS length for each pair. Then, we place the values in the **7x7** matrix, where each cell is the LCS length between **ith** and **jth** strings.

	a	b	c	d	e	f	g
a	**265**	235	199	252	214	251	212
b	235	**250**	211	223	227	223	220
c	199	211	**257**	197	234	195	229
d	252	223	197	**282**	207	241	205
e	214	227	234	207	**252**	205	243
f	251	223	195	241	205	**280**	208
g	212	220	229	205	243	208	**269**

Table 2. Pairwise LCS Lenghts for set_strings

We have 49 total lengths in the set strings, which are all the elements in the matrix. However, the diagonal values are the lengths of the strings themselves. For instance, when we compare the string “a” to itself, we get the number 265, which is bolded in the table. It happens because all the characters obviously match. Thus, we can treat these diagonal strings as a reference for the length of each string. We could have set them to 0 to make the table less redundant with values, but I decided to print them for the general picture. Moreover, in a symmetric matrix, the values with the position **(i, j)** are the same as those with the position **(j, i)**. For instance, when we compare values **a** and **b**, we will get the same LCS value as if we compared b and a. Therefore, to find the number of unique LCS lengths in the **set_strings**, we need to exclude diagonal and mirrored values. We have 42 LCS length values, excluding diagonal values, where only 19 values are unique (Appendix).

To infer which strings are more strongly related to each other, we need to compare their LCS values. A higher value of LCS means that the strings have more genes that are common between the two strings, indicating a stronger relationship. In this case, as each of the strings

underwent the mutation process, where each character could experience an insertion, deletion, or mutation, there are many differences between the strings now. LCS value allows us to identify the similarity of the strings despite the mutations. First, we would exclude diagonal values because they do not represent the relationship between different strings, and then we would need to find the strings with the highest LCS values. Mutations cause the original string to turn into a child/grandchild string. Therefore, the strongly related ones are more likely to share an ancestor because they come from the same branch of a potential genealogy tree.

To infer the resulting genealogy binary tree, we need to identify the roles of the strings: grandparent, parents, and children. The grandparent will take up the tree's root because it is an original sequence that did not experience the mutations. Thus, it would have the highest similarity to the rest of the strings because the rest were created from it after the mutations. The parent nodes are the strings that underwent the “first wave” of the mutations and share a strong relationship with the root because they experience fewer changes. Their children (grandchildren of the root) are relatively less similar to the root because they underwent more mutations. However, they would still have the strongest relationship with their parents.

Relationship Between Strings

Local Strategy vs. Global Strategy

[https://youtu.be/NiYRAARRui0]

In the video, I examined how to identify the precise relationship between strings using local and global strategies. The local algorithms only focus on the relationship between a node and its immediate neighbors. For instance, when we choose children for the root, we do not think about their grandchildren yet. Thus, the algorithm makes decisions based on the current node’s

relationship with other strings. The global approach considers the entire tree at once when deciding how to construct the tree. Thus, it ensures that the resulting tree gives us the best structure.

Strategies Implementation

The local strategy is based on the greedy approach, considering the immediate neighbors of each node. The input is the **len_lcs_matrix,** representing the length between strings **i** and **j** and a list of labeled strings. It outputs a genealogy binary tree as a list of tuples where each tuple describes a parent and its two children. The strings included in the tree are marked as "visited," so we do not backtrack and revisit them because the greedy approach only focuses on the current state. It chooses the root (grandparent) based on the highest pairwise LCS value by iterating through the **len_lcs_matrix** to find the maximum LCS value. It does so because the grandparent has the strongest direct relationship (this, the highest similarity) with the rest of the strings (in our case, with at least one of them). After being processed in the queue, the grandparent is marked as "visited," and it assigns children iteratively. It uses the queue to keep track of what node it processes next. We identify all the unvisited strings for each parent in the queue and find their LCS value with the chosen root. The algorithm removes the node (parent) from the queue with the **queue.pop(0)** and assigns it to two children. Then, the children are processed in the queue. Using the **max1** and **max2**, it tracks the highest and second-highest LCS values to assign the children. Thus, two children are chosen only based on their immediate similarity to the parent. Once two valid children are found, they are added to the tree and marked as "**visited**." This process repeats at each level until all parents have assigned children. It uses the breadth-first traversal because we construct the tree following the relationship grandparent - > parent ->

children. The loop continues until the queue is empty, and it outputs the complete tree. The algorithm outputs a list of tuples with the relationships of the genealogy tree - Parent, Child1, Child2 (Appendix). I created a binary tree diagram representing a hierarchical structure with at most two children for each parent (Figure 1). It gives a more visually appealing version of the output to understand it better.

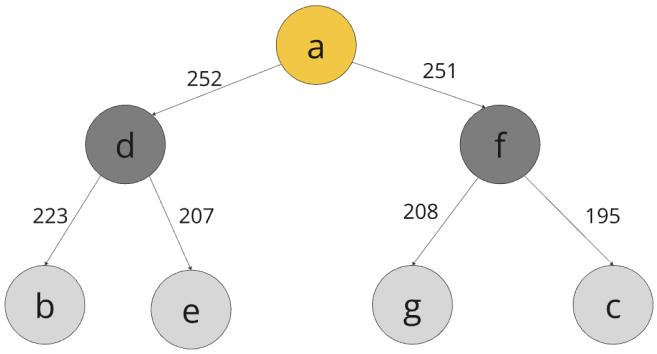

Figure 1. A genealogy tree 1 (built with the local strategy - greedy algorithm).

The root node is “a” as it has the highest LCS value in a pair with string “b” - 252. The numbers on the edges represent the pairwise LCS value of two strings. The algorithm’s greedy nature chooses the best immediate candidate for the root, ‘a’ instead of ‘b.’

For the global algorithm, I used dynamic programming based on the “Similarity Ratio Score” metric obtained by considering all the relationships in the tree. It is calculated using the Longest Common Subsequence of two strings:

Similarity Ratio = Length of LCS / Average Length of the Two Strings

We divide the LCS length by the average of the two strings to account for the potential differences in string sizes. For instance, if we compare two strings, “ABC” and “ABC,” they

have an LCS of “ABC” with a length of 3. Therefore, the second string would match only partially the shorter string. Normalizations allow for more fair comparisons despite the size of the strings. Moreover, it makes the metric more comparable across different datasets. The algorithm constructs a symmetric matrix where each cell contains the similarity ratio between i-th and j-th strings. It computes the total similarity ratio for each string to identify the grandparent and chooses the one with the highest value to become a root. The remaining nodes are assigned based on their similarity score compared to the root. The algorithm keeps track of the two scores based on the iteration rather than sorting. I also created a binary tree diagram representing a hierarchical structure with at most two children for each parent for the global approach (Figure 3). In this case, the root value is “b”, followed by “a” and “e” parent nodes.

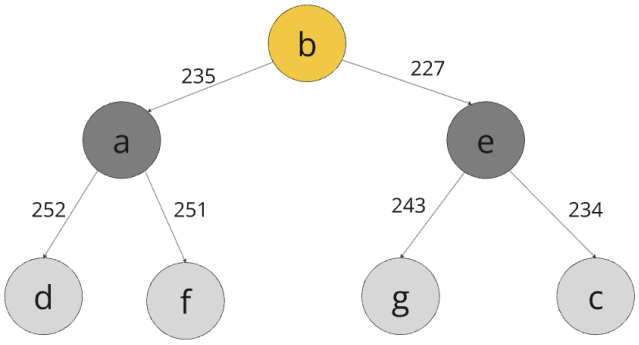

Figure 2. A genealogy tree 1 (built with the global strategy - dynamic programming).

We can calculate the total sum of LCS values by looking at both trees. For the local LCS, the value is

252+251+223+207+208+195 = **1336**

For the global, the total LCS value is

235+227+252+251+243+234 = **1442**

Therefore, the global strategy achieves a higher total LCS value because it considers relationships across all nodes to make the best optimal decision. For instance, it does not immediately go for the highest value of 252 to set it as a node, as the greedy approach does. Thus, it manages to build stronger relationships within the dataset.

Computational Complexity

First, I will analyze the computational complexity of the local approach. The algorithm iterates through all parts of strings in **len_lcs_matrix** to find the tree's root. For **n** strings, the number of pairs is **n(n-1) / 2** because the loops iterate over all possible pairs of strings. The outer loop runs for n strings, and the inner loop runs for **j = i + 1** to **n - 1** to ensure no duplicates. Thus, this operation takes **O(n^2)** time. To compare all the LCS values and update the maximum one, each comparison takes O(1) where we have n(n-1) /2 comparisons; thus, the operations take O(n^2). Adding and removing nodes from the queue and marking nodes as visited takes only O(1). Therefore, the overall time complexity for the local approach is O(n^2). The time complexity for constructing the tree is O(n^2), which involves finding the root and assigning children.

For the dynamic programming approach, we need to run the similarity ratio function, which builds a DP table of size (m+1) x (n+1), taking up O(m * n). After the table is built, the values of the final LCS lengths are retrieved from dp[m][n] and then normalized by the average length of two strings, taking O(1). The time complexity for the similarity ratio operation is O(m * n). We call the function for each pair to build a matrix with the similarity ratio scores. The matrix is symmetric, so only N x N pairs are calculated. If there are N(N-1) / 2 pairs and the function runs in O(m^2), then the whole operation takes O(N^2 * m^2). However, each row has

N values to build the tree, which involves O(N) comparisons for each parent. Therefore, the operation of constructing a tree takes O(N^2).

To induce the scaling growth of both algorithms, I conducted an experiment where the number of genes (n) varied from 10 to 100, with a fixed gene length (m = 100). The plot demonstrates the execution times for the local and global algorithms across the range of n values (Figure 3). When the length of genes m is not fixed and grows with n, the similarity ratio matrix is the main indicator of the time complexity, becoming O(m^2 * n^2). However, since m is a fixed value in this experiment, the term O(m^2) becomes a constant and simplifies the time complexity to O(n^2).

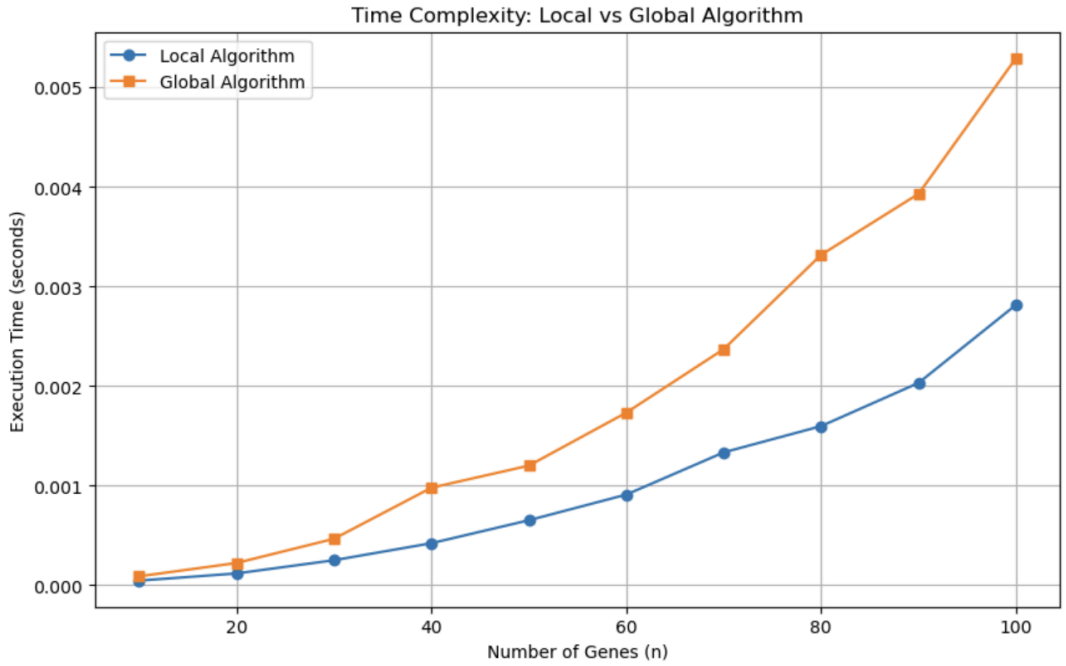

Figure 3. Comparison of the Execution Time: fixed length of genes, increasing number of genes.

The plot for the local algorithm shows an almost quadratic growth O(n^2) as the number of genes increases. The global algorithm also indicates that the quadratic grows, but the execution time is higher than that of the local algorithm. This happens because to identify the relationships

between the strings, we need to compute the overall LCS scores and iterative assigns nodes (parents, children), which makes it more computationally expensive.

Following the feedback from the previous assignment, we can analyze the scaling using the runtime measurements. We can calculate the ratio of how the runtime scales with the size of the input:

Ratio = Runtime at n2 / Runtime at n1

It compares the runtime for two different inputs of n1 and n2. Regarding the time complexity of O(n^k), the ratio is (n2/n1)^2. Thus, if we double the input size, the expected ratio is 2^2 = 4. As we see from Figure 3, the runtime values are measured in seconds and are very small. For simplicity, I convert them to microseconds units to be able to calculate the ratio:

```
Runtime Values (in microseconds):
n = 10, Local Algorithm Time = 98.94 µs, Global Algorithm Time = 272.04 µs
n = 20, Local Algorithm Time = 317.10 µs, Global Algorithm Time = 494.72 µs
n = 30, Local Algorithm Time = 454.90 µs, Global Algorithm Time = 1931.19 µs
n = 40, Local Algorithm Time = 2088.79 µs, Global Algorithm Time = 1147.03 µs
n = 50, Local Algorithm Time = 861.88 µs, Global Algorithm Time = 2235.89 µs
n = 60, Local Algorithm Time = 1504.90 µs, Global Algorithm Time = 2675.06 µs
n = 70, Local Algorithm Time = 1985.07 µs, Global Algorithm Time = 3567.93 µs
n = 80, Local Algorithm Time = 2451.90 µs, Global Algorithm Time = 4791.98 µs
n = 90, Local Algorithm Time = 3102.78 µs, Global Algorithm Time = 5737.78 µs
n = 100, Local Algorithm Time = 3824.95 µs, Global Algorithm Time = 6802.08 µs
```

Then, I compute the ratio and compare it to the expected ratio from the theoretical complexity. Let’s look at the n1 = 50 and n2 = 100.

```
Local Algorithm Ratio: 4.44, Expected: 4.00
Global Algorithm Ratio: 3.04, Expected: 4.00
```

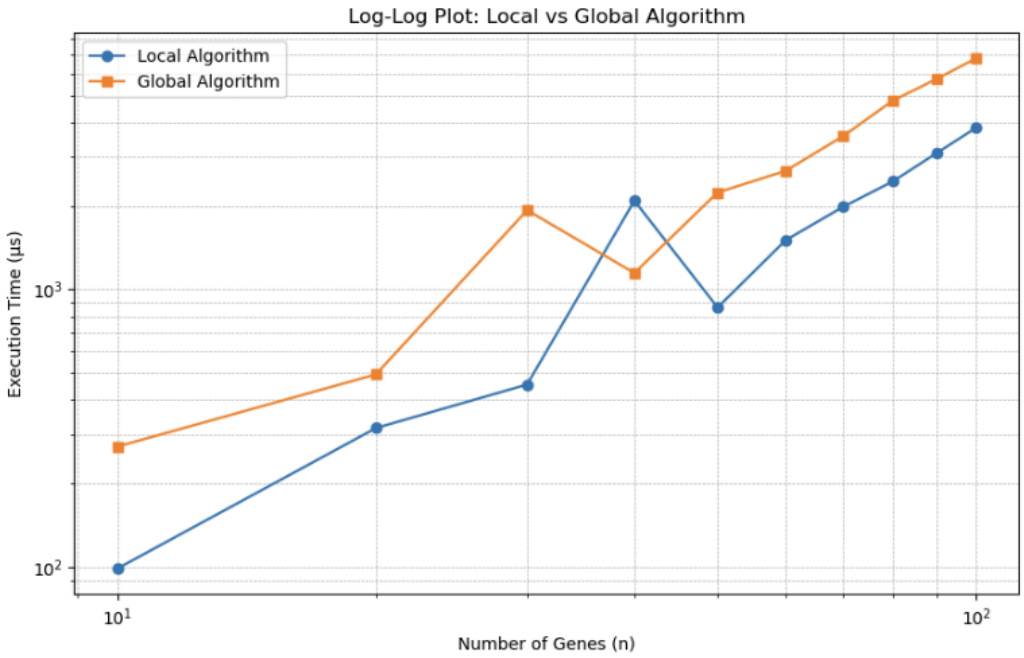

Figure 4. Scaling Analysis of Local and Global Algorithms for n1=50 and n2=100.

The plot demonstrates how the execution time scales as the number of genes increases. Both axes are represented on a logarithmic scale, where x is the number of genes n and y is the execution time in microseconds. As we can see from the plot, the Global Algorithm has a higher execution time than the Local Algorithm because it is more computationally expensive. The expected value for both algorithms is O(n^2), as we derived theoretically. The ratio (4.44) for the local algorithm is relatively close to the expected one, so we can conclude that it aligns with the theoretically derived time complexity. For the global algorithm, the ratio (3.04) is not as close, which might imply some deviations from the theoretical conclusions due to the implementation details of the global matrix. From the plots, both approaches have an upward trend, and the slope of the local approach corresponds to the time complexity of O(n^2). For the global approach, the slope is steeper, so we assume that it might be closer to O(n^2 * m^2).

However, if we experiment with the values of n and set them to 40 and 80, the results become less consistent.

```
Local Algorithm Ratio: 1.17, Expected: 4.00
Global Algorithm Ratio: 4.18, Expected: 4.00
```

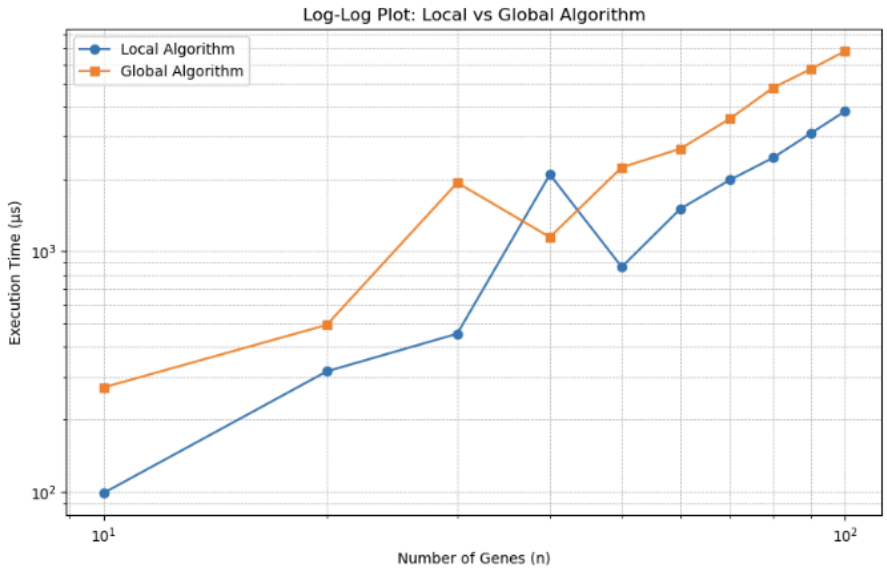

Figure 5. Scaling Analysis of Local and Global Algorithms for n1=40 and n2=80.

The global algorithm's ratio is still close to the expected value. However, the local algorithm's ratio is much lower (1.17). That can potentially happen because the smaller values of n might not represent the scaling trend fully, especially when the deviations in points are involved. Moreover, randomly generated len_lcs_matrix values can create scenarios where certain computations are faster than others. Thus, we can still generalize that the experimentally derived time complexity is consistent with the theoretical.

Estimated Probabilities

I will estimate the probabilities of insertions, deletions, and mutations between the strings based on the pairwise comparison with LCS. Insertions are the characters in the second string that are not a part of the LCS. Deletions are the characters of the first string that are not a part of the LCS. Mutations are scenarios where one character in the first string is replaced by a character in the second string.

Insertions	len(y) - LCS Length
Deletions	len(x) - LCS Length
Mutations	min(len(y) - LCS Length, len(x) - LCS Length)

Table 3. Calculation of Probabilities

According to Table 3, insertions represent the characters we need to remove to align with the LCS and match the first string. Deletions represent the characters in x that would need to be removed to align with the LCS and match the second string. For the mutations, I use the minimum because they are connected by the deviations of both insertions and deletions. The algorithm goes through all the pairs of strings, so for n strings, there are n(n-1) / 2 pairs. The lcs_length function is called for each part to compute the insertions, deletions, and mutations. The outer loop goes through all the strings, while the inner loop compares the current string to the next one. To find the total number of changes, we calculate the sum of all insertions, deletions, and mutations, normalizing the probabilities:

Insertion Probability: insertions / total changes

Deletion Probability: deletions / total changes

Mutation Probability: mutations / total changes

In this case, the total number of changes is a baseline to identify how important each change is compared to the overall string. Insertion probability is 0.379 (37.9%); deletion probability is 0.329 (32.9%); mutation probability is 0.292 (29.2%). We can conclude that insertions happened more often, followed by deletions and mutations. Thus, the string has been extended or shortened while the character substitutions happened less frequently.