CS110: Decoding the relationships between genes

December 19, 2024

1 Manual Tree Estimation

Our initial matrix of longest substrings is:

	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

From the `len_lcs_matrix`, it is possible to infer that strings with higher LCS share more common characters among them, indicating they underwent less mutations after duplication. Thus, the higher the LCS between a pair of sequences, the more related they are. We can use this information to infer parent-child relationships, since the gene sequence of the child would share more common characters with the original sequence from its parent.

One strategy to infer a resulting genealogy tree is to find the two highest LCS for each row, such that we form triads of one parent node and two of its most closely related children. These triads will correspond to the first the first two levels of our tree. Once having formed all triads across all possible parents $[a, g]$, we select two triads (each with one parent, two children), and then connect one remaining grandparent node.

Here, we arrive in a perfect binary tree for 7 nodes. However, we must verify whether it we have effectively manage to organize the genetic hierarchy. Arriving at the original genealogy tree would testing several different configurations of subs-trees.

2 Algorithmic Strategy

2.1 Video

Link: Google Drive

2.2 Local Strategy

2.2.1 Implementation

My goal in this strategy is to use a greedy algorithm to compare pairs of nodes and build a genealogy tree using from the top-down, starting from the root node.

This algorithm is centered on node objects because of its abstraction and encapsulation advantage. An object, defined through a class, allows us to store multiple data structures into one single representation, simplifying the interaction of the algorithms with the data structures. In our local approach, the node object will be useful to store information about pairs of nodes. Namely,

- Index of the first gene sequence
- Index of the second gene sequence
- Longest Common Subsequence length between two sequences (represents the weight)
- Pointer to the parent node
- List of child nodes

In this implementation, we arbitrarily select a root node since we have no prior information about the sequence. For example, let us pick the last sequence of `len_lcs_matrix`, g.

2.2.2 Algorithm Steps

We will use a priority queue implemented through a `MaxHeap()` class to retrieve nodes with the highest LCS between them. A priority queue is well suited for this task because, at each iteration, it allows us to efficiently access the child-node with the highest LCS with its parent node. This establishes how the algorithm selects the next node to connect with its children, which also maximizes the LCS length.

We keep track of the sequence indexes which were already visited through a list of gene sequence indexes. [not sure whether to include this] We can use a list in this case instead of a more strict data structure like a set, for example, because the algorithm contains a condition to only match a current node with an unvisited sequence, avoiding repetitions.

- At each step, while there are elements in the priority queue, we perform the following:
    1. Retrieve a node from the priority queue. This node contains the indexes of two sequences with the current highest LCS length.
    2. Using the index of the first sequence i of the retrieved node, loop through all unvisited second sequence entries j in `len_lcs_matrix`.
    3. Search for two pairs of sequences, (i, j_{top1}) and (i, j_{top2}), with the highest LCS lengths, ensuring that the binary tree structure of two children per parent is maintained.
- For each of the top two sequences:
    1. If the current pair of nodes holds a higher LCS length than the previously computed pair, immediately update the results.
    2. This approach avoids backtracking, making it a greedy algorithm.
- Once j_{top1} and j_{top2} are identified as the child nodes of i:
    1. Update the parent pointers of j_{top1} and j_{top2}.
    2. Update the list of children associated with i.
    3. Add the indexes of j_{top1} and j_{top2} to the visited index set.
- Finally:
    1. Add both child nodes to the priority queue.
    2. In the next iteration, compute their children.
    3. Repeat the process until there are no nodes left in the priority queue.

2.2.3 Analysis of the Greedy Property

A greedy algorithm is guaranteed to find the optimal substructure of a problem if it meets the greedy property. Namely, that locally optimized decisions lead to a globally optimized solution.

In this particular problem, we use the sum of all LCS between nodes as a metric to find the optimal substructure. We consider the LCS between two sequences as a relationship distance, where the higher the LCS, the more closely related these sequences are (i.e. the relationship distance is lower).

Consequently, our optimal substructure is a perfect genealogy tree with one grandparent, two parents and four children which contains the highest possible sum between all pairs of nodes.

Using our greedy algorithm, let us consider that our greedy solutions led to a 7-node genealogy tree. However, this tree can have 7 different possible root nodes, leading to multiple different configurations of subtrees,

where the sum of the LCS lengths between all nodes can be significantly different. A greedy choice in this algorithm decreases subproblems, or subtrees, by computing the children of every parent node at each iteration, but because it does not backtrack, we also cannot alter the root once it has been picked.

Further, since we have no prior contextual information about the gene sequences, besides their length and LCS with the other sequences, we are not guaranteed to select the root which will eventually lead us to the optimal substructure. Thus this local approach does not meet the greedy property, and is not guaranteed to arrive at the tree with maximum LCS across all nodes.

2.3 Global Strategy

2.3.1 LCS as an optimizing metric

Regardless of whether additions, deletions and mutations happen in the middle of the gene sequence, the LCS represents the highest number of characters shared between two sequences. Thus, it would be appropriate to deduce that the higher the subsequence, the less changes to the sequences occurred and the more strongly related these gene sequences are. Still, there are drawbacks to using the LCS as a decision variable which are addressed in section.

2.3.2 Implementation

This implementation differs from the greedy local strategy because instead of comparing pairs nodes in order to build a tree, we are instead comparing different tree configurations, aiming to reach the highest LCS sum across all nodes. In the local strategy, we noticed that we are not guaranteed to reach the global optimal using a greedy algorithm.

Alternatively, as a global approach, we can use Dynamic Programming (DP), which is suitable for this problem because we aim to reach an optimal substructure, and, among different tree configurations, we have overlapping sub-problems or sub-trees.

As opposed to brute-force algorithms, where we would just compare all possible combinations of trees, our global DP approach avoids repeating the sums of the edges for overlapping sub-trees. This solution uses a similar tabulation approach to the `LCS-length` function inspired by Cormen et al.

In the Longest Common Subsequence (LCS) problem, starting from the top left of the matrix, we iteratively accumulate the length of the subsequences as we increase the number of characters.

Now, in our maximum tree distance problem, we are concerned the total relationship distance within the ranges of different subtree combinations. Thus, we can create an $n \times n$ matrix where every cell $[i, j]$ corresponds to an optimal subtree ranging from elements i to j.

Since our optimal substructure is a perfect genealogy tree, we must ensure that the solution to our global approach also remains balanced. o ensure the tree remains balanced, the left and right sub-trees of any root must satisfy the condition that the difference in the number of nodes between the two subt-rees does not exceed 1. If a root results in unbalanced left and right sub-trees, this root is skipped during the calculation.

We must also consider that for a given perfectly balanced sub-tree, an optimal arrangement of our nodes also depends on the selection of the root. To compute the maximum relationship distance for a sub-tree in the range $[i, j]$, we evaluate each possible root r within the range. For each root, we calculate the total weight of the sub-tree by considering the pre-computed weights of the left and right sub-trees. The optimal root is the one that maximizes this total weight while maintaining the balanced tree constraint. We store:

- The maximum total weight in a `DP` table
- The optimal root for each range in a separate `Roots` table

2.3.3 Recurrence Relation

To compute the DP value for a subtree spanning $[i, j]$ with a chosen root r, we combine:

- The sum of relationship distances between nodes in the range $[i, j]$.
- The total weight of the left subtree, $\texttt{DP}[i][r-1]$.
- The total weight of the right subtree, $\texttt{DP}[r+1][j]$.

The recurrence relation is given by:

$$\texttt{DP}[i][j] = \max_{r \in [i,j]} \left(\texttt{DP}[i][r-1] + \texttt{DP}[r+1][j] + \texttt{len\_lcs\_matrix}[i][j] \right),$$

where r is a valid root in the range $[i, j]$, len_lcs_matrix$[i][j]$ represents the sum of the pairwise relationship distances within the range $[i, j]$. Further, if $i = j$, $\texttt{DP}[i][j] = 0$, as there is only one node.

With an update in `DP`, `Roots` is updated in parallel:

$$\texttt{Roots}[i][j] = r, \quad \text{where } r \text{ maximizes } \texttt{DP}[i][j].$$

2.3.4 Algorithm Steps

1. **Initialization:** We start by creating an $n \times n$ `DP` table and an $n \times n$ `Roots` table.
2. **Iterative DP Computation:**
   - For each range length $l = 1$ to n, compute $\texttt{DP}[i][j]$ for all i and $j = i + l - 1$.
   - For each root $r \in [i, j]$, calculate:

     $$\texttt{DP}[i][j] = \texttt{DP}[i][r-1] + \texttt{DP}[r+1][j] + \texttt{len\_lcs\_matrix}[i][j].$$

   - Update $\texttt{Roots}[i][j]$ with the root r that maximizes $\texttt{DP}[i][j]$.
   - Both iterative and recursive DP solutions are compared in section 4 Computational Critique
3. **Tree Reconstruction:**
   - Reconstructing the tree will involve a recursive algorithm similar to `Print-LCS` in Cormen et al.
   - Starting from the full range $[1, n]$, use the `Roots` table to recursively reconstruct the tree.
   - For a given range $[i, j]$, the root $r = \texttt{Roots}[i][j]$ splits the range into left ($[i, r-1]$) and right ($[r+1, j]$) subtrees.

We can visualize simple examples of these matrices as:

DP =

	0	1	2	3
0	0	15	25	40
1	–	0	20	35
2	–	–	0	18
3	–	–	–	0

Roots =

	0	1	2	3
0	0	0	1	2
1	–	1	1	2
2	–	–	2	2
3	–	–	–	3

- In the **DP Table**:
  - For subtree ranging from $[0, 3]$, the maximum weight is 40.
  - For subtree ranging from $[1, 3]$, the maximum weight is 35.
- In the **Roots Table**:
  - For subtree ranging from $[0, 3]$, the root is at index 2.
  - For subtree ranging from $[1, 3]$, the optimal root root is at index 2.

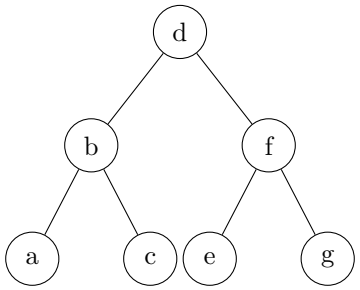
Unlike the greedy approach, the DP strategy is able to explore all possible subproblems, guaranteeing that we find an optimal substructure.

3 Strategy Results

3.1 Global Approach

Since the Dynamic Programming approach explores all possibilities and is guaranteed to reach the optimal result, we can use its results as a ground-truth optimal substructure to compare against the local strategy.

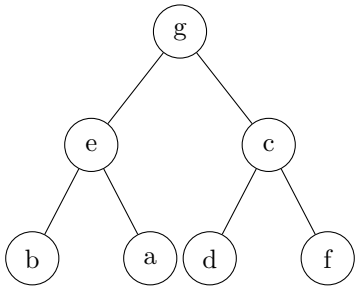
The global approach delivered the following result as the optimal genealogy tree:



Total tree sum: 1323

3.2 Local Approach

The local approach suggests a tree structure of



Total tree sum: 1305

If the local approach meets the greedy property, locally optimized decisions should lead to a globally optimized solution. Because the total LCS length found through the local strategy is smaller than the LCS length of the global strategy tree, we demonstrate that the local approach does not meet the greedy property and thus, does not arrive at a global optimum.

In particular, the local strategy fails because we do not have enough information about what is the most strategic node to pick as the starting root, such that we need to make an arbitrary decision.

The global strategy, however, explores all different combinations of roots, allowing us to compare between different tree structures and pick the largest one.

4 Computational Critique

4.1 Building the LCS matrix

For both global and local approaches, matrix `len_lcs_matrix` is the input. Computing `len_lcs_matrix` will contain the same time complexity for the two approaches.

In order to compute the **len_lcs_matrix** matrix, where we have 7 gene sequences, we compute the LCS distance across all unique pairs of strings, resulting in a 7x7 matrix using the function `generate_len_lcs_matrix()`.

At every iteration, we compute the LCS between each pair of nodes. This algorithm uses the implementation of the `lcs_length` and `print_lcs` functions from Cormen et al.

The function `lcs_length` computes the Longest Common Subsequence (LCS) between two strings x and y, where $m = \text{len}(x)$ and $n = \text{len}(y)$. It uses two matrices:

- C is a $(m+1) \times (n+1)$ matrix storing LCS lengths.
- B is a $(m+1) \times (n+1)$ matrix for backtracking directions, used to reconstruct the Longest Common Subsequence. Which will not be used in the context of building an optimized genealogy tree.

The function iterates over all positions (i, j) where $1 \leq i \leq m$ and $1 \leq j \leq n$. At each position, we execute operations of constant time $O(1)$:

$$C[i][j] = \max(C[i-1][j], C[i][j-1]) \quad \text{or} \quad C[i][j] = C[i-1][j-1] + 1$$

The nested loops over i and j run in $O(m \cdot n)$ time.

Let:

- N be the number of gene sequences (size of the input list `genes`).
- M be the average length of each gene sequence.
- `lcs_matrix` be a precomputed $N \times N$ matrix containing pairwise LCS scores.

Consequently that the complexity of `lcs_length()` is $O(M^2)$.

In `generate_len_lcs_matrix()`, our outer loop iterates N times through all genes, and the inner loop iterates $i-1$ times, with i being dynamically altered by the first loop. The total number of iterations will correspond to the arithmetic sum of the first n integers, where $n = N - 1$, such that:

$$\begin{aligned} T(M, N) &= \sum_{i=0}^{N-1} \sum_{j=0}^{i-1} O(M^2) \\ &= \sum_{i=0}^{N-1} O(M^2) \cdot i \\ &= O(M^2) \cdot \frac{(N-1)(N)}{2} \\ &= O(M^2) \cdot \frac{N^2 - N}{2} \end{aligned}$$

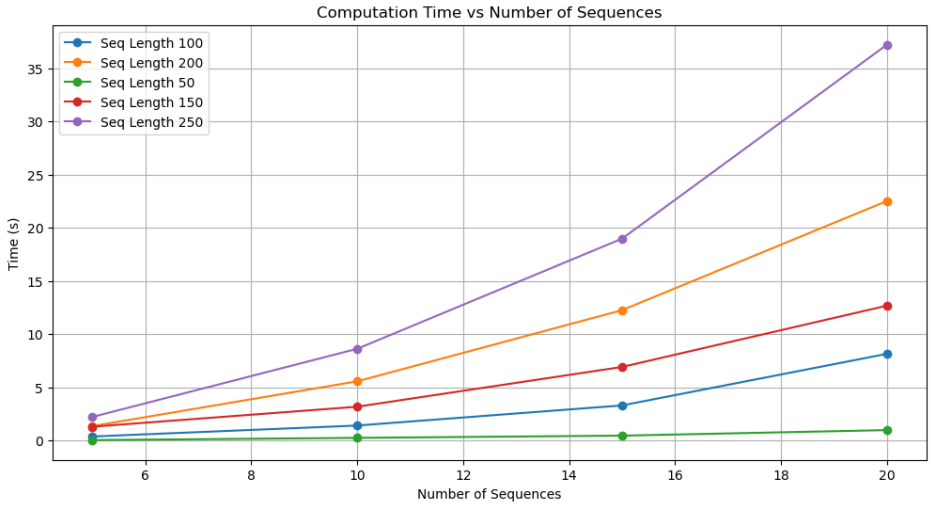
The dominant term of this equation is N^2, the complexity of the code is $O(N^2)$.

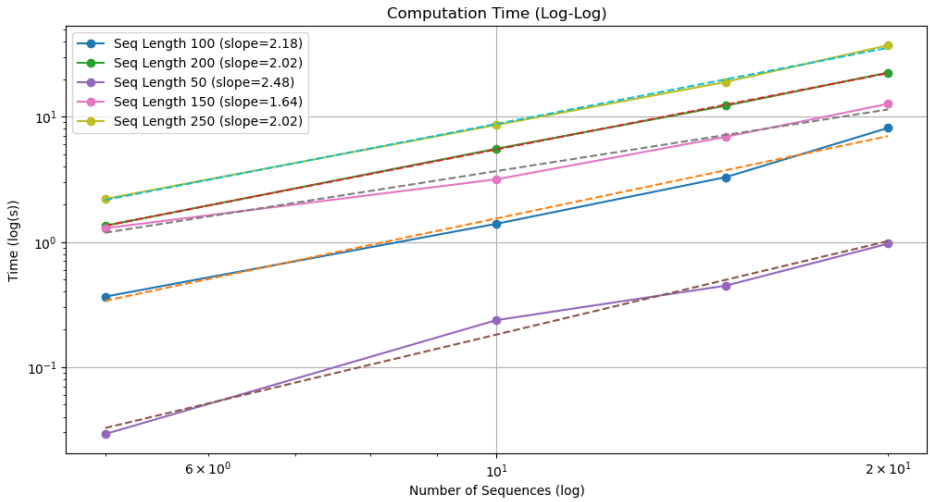
Simplifying, we get:

$$T(M, N) = O(M^2 \cdot N^2)$$

Since we compute the LCS length for all elements in the loop, this algorithm would result in a complexity of: $O(M^2 \cdot N^2)$. In practical terms, this would mean that if the inputs N, M double, the runtime would grow by a factor of 16.

Experimental results with both different sequence lengths and number of genes are documented on the plots below.





In this scenario, we notice that the slopes of the regression line in the log-log plot closely borders 3, which indicates proximity to $O(n^3)$. The fact that the experiments border one order of magnitude lower than the theoretical complexity of $O(M^2 \cdot N^2)$ can have different explanations. One example is that the input size might not be large enough to reveal the worst-case overhead for recursion or nested loops. However, longer ranges of experiments would lead to unreasonably long computed results.

4.2 Implementing the Local Strategy

Let:

- N be the number of gene sequences (size of the input list `genes`).
- M be the average length of each gene sequence.
- `lcs_matrix` be a precomputed $N \times N$ matrix containing pairwise LCS scores.

In this function we use a priority queue to iteratively construct the tree, where each sequence is visited at most once. Our main operations include:

1. **Node Exploration:** this algorithm visits each sequence (node) once. Since there are n sequences, the total number of expansions is $O(n)$.
2. **LCS Comparisons for Unvisited Nodes:** For each sequence, the algorithm loops through all other sequences to find the top two LCS scores among unvisited nodes. Because the LCS values are precomputed and stored in the matrix, each comparison takes $O(1)$ time.

   Therefore, for each node, the comparison step takes $O(N)$ time.
3. **Priority Queue Operations:** Adding a node to the priority queue (`heappush`) or removing a node (`heappop`) has a complexity of $O(\log N)$.

   Since at most n nodes are pushed or popped from the priority queue, the total time for all heap operations is $O(N \log N)$.

Combining the steps:

- Node expansions: $O(N)$
- LCS comparisons: $O(N) \cdot O(N) = O(N^2)$
- Priority queue operations: $O(N \log N)$

The dominant term is $O(N^2)$ for large N, where N is the number of sequences. In practical terms, this would mean that when the input size doubles, we expect the average runtime to grow by a factor of 4 (quadratic scaling).

If we include the time complexity of precomputing the LCS matrix (`generate_len_lcs_matrix`), the overall time complexity becomes:

$$O(N^2 \cdot M^2) + O(N^2) = O(N^2 \cdot M^2).$$

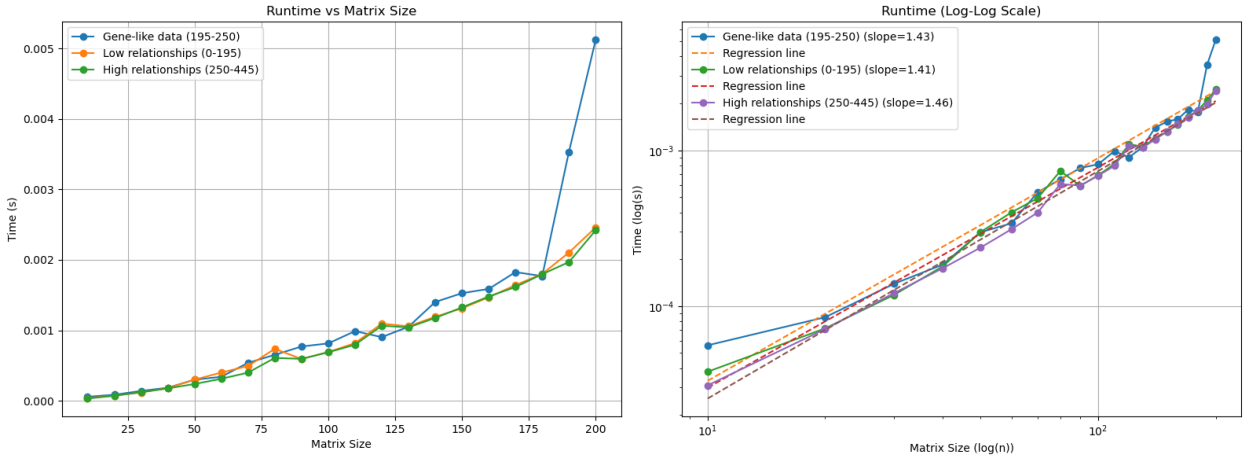
Here, $O(N^2 \cdot M^2)$ dominates because M (the average gene length) is generally much larger than 1.

In practical terms, this analysis highlights how the pre-computation step for the LCS matrix is the most time-consuming part of the algorithm, since the tree construction step adds an additional $O(N^2)$ complexity, but which is negligible compared to $O(N^2 \cdot M^2)$.

The scaling behavior of the algorithm can be further observed through the experimental plots below, which use the metric of time, in seconds, plotted additionally in log scale to demonstrate the asymptotic slope. In these experiments, we are considering the comparison between three different inputs, all of which involve pre-computed, randomized LCS matrices.

Three types of input include:

- Input type 1: similar numbers of LCS as the genes example ($195 <$ sample < 250)
- Input type 2: low relationships (all samples lower than 195 but with a lower bound of 0)
- Input type 3: high relationships (all samples higher than 250 but with an upper bound of the same range as above, 445)



In this scenario, we notice that the slopes of the regression line in the log-log plot closely border 2, which indicates proximity to $O(n^2)$. This roughly matches the expected theoretical estimations.

4.3 Implementing the Global Strategy

4.3.1 Iterative Bottom Up Approach

Let:

- N be the number of gene sequences (size of the input list `genes`).
- M be the average length of each gene sequence.
- `lcs_matrix` be a precomputed $N \times N$ matrix containing pairwise LCS scores.

The `max_LCS_tree_iterative()` function builds a dynamic programming (DP) table to compute the maximum weight for all possible subtrees and chooses the optimal root for each range. Let n be the total number of nodes.

1. **Initialization of DP and root tables:** The tables `dp` and `root` are initialized as $N \times N$ matrices, which takes $O(N^2)$ time.
2. **Nested Loops Over Subtree Ranges:** The function iterates over all possible subtree sizes size $\in [2, N]$ and all starting indices i. For a given size, the corresponding ending index is $j = i + \text{size} - 1$.
   - There are $O(N)$ possible sizes.
   - For each size, the starting index i ranges from 0 to $N - \text{size}$, resulting in $O(N)$ iterations for each size.

   Hence, the total number of (i, j) pairs (ranges) is $O(N^2)$.
3. **Iterating Over All Roots in a Range:** For each range $[i, j]$, the function tries all possible roots $r \in [i, j]$. The number of roots is proportional to the size of the range, i.e., $O(j - i + 1)$.

   This algorithm contains a condition to skip over unbalanced trees. Since the range size can be at most $(j - i + 1)$, where no unbalanced subtrees are found for any choice of root, the worst-case cost for iterating over roots is $O(j - i + 1)$ for each range.
4. **Computing Subtree Weights:** For each root r, the function computes:

- Left and right subtree weights using the DP table: $O(1)$.
- Sum of LCSweights lengths connected to the root: This involves iterating through all nodes in the range $[i, j]$, which takes $O(N)$ time.

Thus, for each root, the cost of computing the total weight is $O(N)$.

Overall, there are $O(N^2)$ ranges $[i, j]$. For each range, there are $O(N)$ roots to try. For each root, computing the sum of all LCS lengths takes $O(N)$ time.

Therefore, the total time complexity is:

$$\begin{aligned} T(n) &= O(N^2) \cdot O(N) \cdot O(N) \\ &= O(N^4) \end{aligned}$$

The `reconstruct_tree()` function reconstructs the tree using the `root` table. It processes each node exactly once and makes recursive calls for its left and right subtrees.

- At each level of recursion, the function performs $O(1)$ work to create the current node and determine the left and right subtrees.
- The total number of recursive calls is equal to the number of nodes, which is $O(N)$.

Thus, the time complexity of `reconstruct_tree()` is:

$$O(N)$$

Combining the results:

- The `maxWeightTree` function has a time complexity of $O(N^4)$.
- The `reconstruct_tree()` function has a time complexity of $O(N)$.

Since $O(N^4)$ dominates $O(N)$, the overall time complexity of the program is:

$$O(N^4), \quad \text{where } N \text{ is the number of nodes.}$$

In practical terms,this would mean that if the number of gene sequences doubles, the runtime increases by a factor of 16 (2^4). This algorithm is significantly expensive and could easily lead to a performance bottleneck for a large number of gene sequences to compare.

If we include the time complexity of precomputing the LCS matrix (`generate_len_lcs_matrix`), the overall time complexity becomes:

$$O(N^2 \cdot M^2) + O(N^4) = O(N^2 \cdot M^2 + N^4).$$

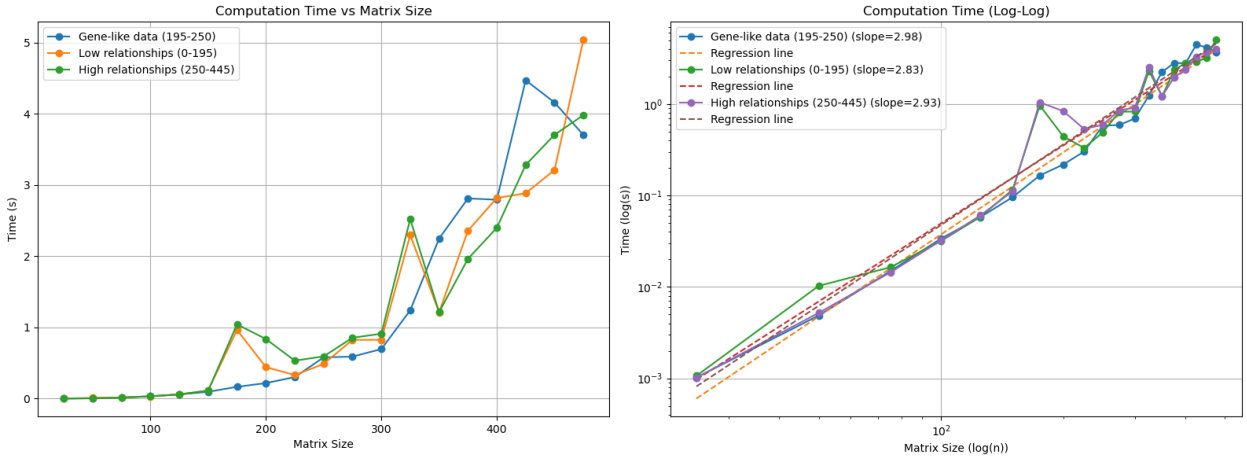
Here, the dominant term in the sum will depend on the size of M.

1. If M is smaller than N, then $O(N^4)$ will dominate
2. If M is larger or approximates N, then $O(N^2 \cdot M^2)$ will dominate.

For small genealogy tree applications where the number of gene sequences is small compared to the gene sequence lengths, it is more likely that in a practical application the complexity of $O(N^2 \cdot M^2)$ will dominate.

The scaling behavior of the algorithm can be further observed through the experimental plots below, which use the metric of time, in seconds, plotted additionally in log scale to demonstrate the asymptotic slope. In

these experiments, we are considering the comparison between three different inputs, all of which involve pre-computed, randomized LCS matrices.



In this scenario, we notice that the slopes of the regression line in the log-log plot closely borders 3, which indicates proximity to $O(n^3)$. The fact that the experiments border one order of magnitude lower than the theoretical complexity of $O(n^4)$ can have different explanations. One example is that the input size might not be large enough to reveal the worst-case overhead for recursion or nested loops. However, longer ranges of experiments would lead to unfeasibly long computed results.

This also applies to other experimental results.

4.3.2 Recursive Top Down Approach

In `max_LCS_tree_recursive()` we compute the maximum weight of the subtree for a given range $[i, j]$ of nodes. Let n be the total number of nodes.

Recursive Call Tree
The function computes the maximum weight for the subtree in the range $[i, j]$.

For each range, it tries every node r in $[i, j]$ as the root. For each r:

- It calculates the left subtree weight for $[i, r-1]$.
- It calculates the right subtree weight for $[r+1, j]$.
- It computes the sum of weights connected to the root node r.

The recursion terminates at the base case when $i > j$ (empty range) or $i == j$ (single node).

Memoization ensures that each unique range $[i, j]$ is computed only once. Since i and j can take values from 0 to $n-1$, the total number of unique subtrees is, $\frac{n(n+1)}{2}$ which eventually scales to $O(n^2)$.

This corresponds to the upper triangular part of an $N \times N$ matrix.

For each range $[i, j]$:

- The function tries every node r in $[i, j]$ as the root. The number of nodes to try is $(j - i + 1)$.
- For each root r:
  - The sum of weights connected to r in the range $[i, j]$ is computed, which takes $O(j - i + 1)$ time.

- Left and right subtree weights are computed recursively (memoized results).
- This algorithm contains a condition to skip over unbalanced trees. Since the range size can be at most $(j - i + 1)$, where no unbalanced subtrees are found for any choice of root, the worst-case cost for iterating over all roots is still $O(j - i + 1)$ for each range.

Computing each subtree is proportional to: $O((j - i + 1)^2)$.

To calculate the total time complexity, we sum up the work for all $O(N^2)$ possible subtrees. The size of each range $(j - i + 1)$ can vary from 1 to N. For each range size k, there are $O(N - k + 1)$ subproblems.

The number of ranges of size k is $O(N - k + 1)$. The work for each such range is $O(k^2)$.

Thus, the total time complexity is:

$$T(n) = \sum_{k=0}^{N-1} O((N - k + 1) \cdot k^2).$$

Simplifying the summation gives:

$$T(n) = O(N^3).$$

- **Time Complexity of `max_LCS_tree_recursive()`:** $O(N^3)$, where n is the number of nodes.
- **Time Complexity of `reconstruct_tree`:** $O(N)$, since the tree reconstruction visits each node exactly once.

In practical terms, this would mean that when the input size doubles, we expect the average runtime to grow by a factor of 8 (cubic scaling).

If we include the time complexity of pre-computing the LCS matrix (`generate_len_lcs_matrix`), the overall time complexity becomes:

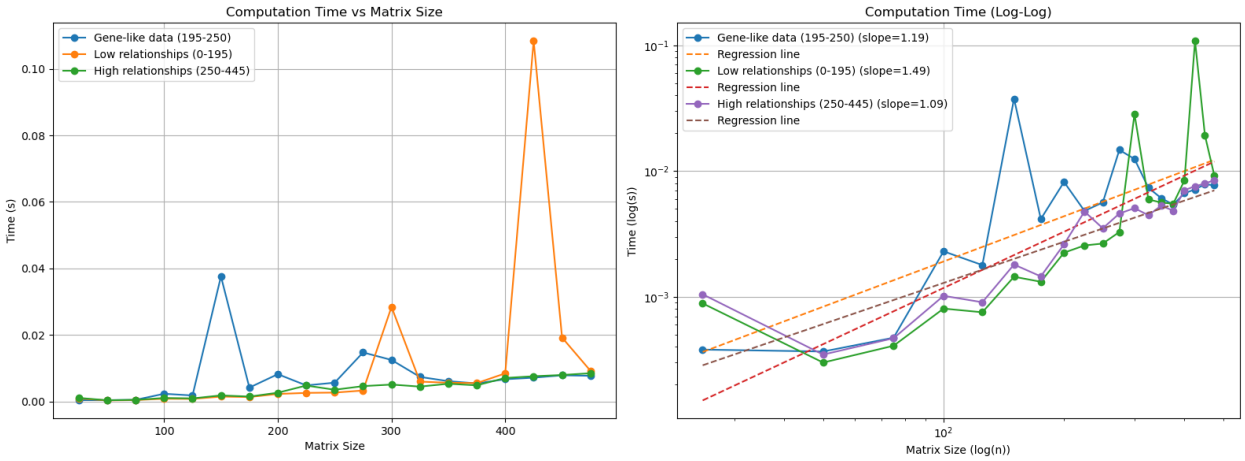
$$O(N^2 \cdot M^2) + O(N^3) = O(N^2 \cdot M^2 + N^3).$$

Here, the dominant term in the sum will depend on the size of M.

1. If M is smaller than N, then $O(N^3)$ will dominate
2. If M is larger or approximates N, then $O(N^2 \cdot M^2)$ will dominate.

For small genealogy tree applications where the number of gene sequences is small compared to the gene sequence lengths, it is more likely that in a practical application the complexity of $O(N^2 \cdot M^2)$ will dominate.

The scaling behavior of the algorithm can be further observed through the experimental plots below, which use the metric of time, in seconds, plotted additionally in log scale to demonstrate the asymptotic slope.



In this scenario, we notice that the slopes of the regression line in the log-log plot closely borders 2, which indicates proximity to $O(n^2)$. The fact that the experiments border one order of magnitude lower than the theoretical complexity of $O(N^3)$ for equivalent reasons explained past chapters.

4.3.3 Bottom-Up vs. Top-Down approaches

Using the evidence above, it is possible to conclude that the Top-down recursive approach performs more efficiently both in terms of theoretical and experimental metrics of time complexity using memoized function calls. The bottom-up approach, however, used a DP tabulation approach where nested loops repeat operations multiple times. This would lead to the unnecessary recalculation of subtrees and lead to a worse time complexity of $O(N^4)$.

It is important to note that the recursive approach has limitations including maximum recursion depth, and additional space complexity considering the function call stack. However, for small dataset applications, the Top-down recursive strategy would be preferred due to its lower time complexity of $O(N^3)$.

4.4 Decision of Local vs. Global Strategy

From the analysis above, we discover that building the LCS matrix `len_lcs_matrix` is the most costly process in developing the genealogy tree. For small applications, both algorithms will converge to $O(N^2 \cdot M^2)$, regardless of the greedy or the DP approach selected.

However, if we isolate the performances of aforementioned algorithms, adopting a Greedy (local) approach to develop a genealogy tree for a given sequence of genes is more computationally efficient than a Dynamic Programming (global) approach. This is due to the fact that the greedy solution does not rely on comparisons between multiple subtrees stored in a table, but rather on the immediate comparisons between nodes.

However, it is important to note that while the greedy strategy is more efficient, as commented in section 3. Strategy Results, it is not guaranteed to find the optimal solution as the global, Dynamic Programming strategy does. Therefore, we arrive at a trade-off of optimality and efficiency, while the DP approach is guaranteed to reach the optimal solution by solving all possible subproblems, the greedy approach is more efficient.

Because of this, the recommended application of each algorithm will depend on the context of the problem. Considering the biotechnology company, on small datasets, both approaches will escalate to the worst time complexity of developing the LCS matrix $O(M^2N^2)$. However, for applications on large datasets, which contain a number sequences N larger than the length of subsequences M, it will be preferable to use the local greedy strategy to achieve a genealogy tree at a lower runtime.

Nevertheless, if the problem requires finding the precise optimal genealogy tree, and it involves a small dataset with shorter gene sequences, then a DP approach could be feasibly applied.

5 Probability Estimation

We can estimate the probabilities of different alterations in the gene sequences using the Levenshtein Distance. It quantifies how different two strings are by counting the minimum number of edits required to change one string into the other. These edits can represent either an insertion, a deletion or a replacement, or in this context, a mutation.

The Levenshtein distance is computed through the following piecewise function, where a corresponds to the first string, b to the second string; i is the terminal character position of a and j is the terminal character position of b:

$$lev_{a,b}(i,j) = \min \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0 \\ \min \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise} \end{cases}$$

In the context of the problem $lev_{a,b}(i-1,j)+1$ represents a deletion, $lev_{a,b}(i,j-1)+1$ an insertion and $lev_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)}$ a substitution.

This function will result in a $n_a \times n_b$ matrix, where n_a is the length of a and n_b is the length of b. Ultimately, our bottom right corner will contain the computed Levenshtein distance.

With the Leventhsein distance function we can calculate the number of minimum insertions, deletions and mutations. However in our gene sequencing problem we will we also have to store operations in which a character remains the same, such that our probabilities are relative to all gene sequence operations, and not only the edits as the Leventhsein distance suggests.

Namely, we will compute the probabilities by dividing the aforementioned quantities by the total number of operations, among matches, deletions, insertions and substitutions. In practical terms, the probabilities would indicate the likelihood of a particular edit, relative to all character-level comparisons made when aligning two strings, being a deletion (p_{del}), an insertion (p_{ins}), a mutation (p_{mut}) or a match (p_{mat}).

In order to estimate the probabilities for any given gene sequence. We can use the concept of Markov matrices to identify a transition matrix which contains the probabilities of transforming any one gene sequence into a second gene sequence.

A Markov matrix is a square matrix which stores the probabilities of transitioning between a current state and a future state in a dynamic system.

Let us consider any two input gene sequences X and Y. A random character from string X is represented by state A and a random character from string Y is state B. To represent the transition state for insertions and deletions, we also consider the presence of a null character (ϵ). Ultimately, the Markov matrix will be a 3x3 matrix where for a specific pair of strings X and Y, the entry on the j-th column and i-th row (a_{ij}) represents the probability of switching from one state to another, and the diagonal entries (a_{ii}) represent the states which remain the same.

In general terms:

	A	B	ϵ
A	1 - p_{mat}	p_{mut}	p_{ins}
B	p_{mut}	1 - p_{mat}	p_{ins}
ϵ	p_{del}	p_{del}	$1 - 2p_{del}$

For example, for the pair of strings a and b from the genes set, we have the following probabilites, rounded up to 3 decimal points and with an additional normalization step to avoid floating point precision errors:

	A	B	ϵ
A	0.887	0.019	0.041
B	0.019	0.887	0.041
ϵ	0.094	0.094	0.918

This would indicate the probability of mutating any character of a (A) into any character of b (B), and vice-versa is 0.019 (1.9%), of deleting any character of a or b is 0.094 (9.4%) and of inserting any character of a or b is 0.036 (3.6%).

Although the current sample size is small, we can produce an estimate by computing transition matrices for every unique pair of genes, resulting in a total of 21 matrices.

Subsequently, we average the entries across all transition matrices in the sample. This way, we decrease the variance in the measurements and approximate the true means of the probabilities for each operation, for any pair of gene sequences, including those outside the mean. Our averaged Markov matrix (M) will be:

$$M = \begin{bmatrix} 0.839 & 0.065 & 0.101 \\ 0.065 & 0.839 & 0.101 \\ 0.096 & 0.096 & 0.797 \end{bmatrix}$$

This would mean that the probability of mutating any character of any sequence X into a character of another sequence Y, and vice-versa, is 0.065 (6.5%), of deleting any character from X or Y is 0.096 (9.6%) and of inserting any character on X or Y is 0.101 (10.1%). In the problem statement, it is specified that the probabilities of mutation, insertion or deletion are small, thus the results are reasonable.

Using Markov matrices also has the added benefit of allowing us to predict the distribution of deletions, insertions and mutations for future sequences. The evolution of the system over time is determined by applying the Markov matrix repeatedly, creating a Markov chain.

In turn, Markov chains would allow the biotechnology company to investigate if there is an equilibrium distribution which the whole system falls into regardless of the initial condition, or gene sequence. That is, if given a particular gene sequence, after several generations, we could answer the question of whether the distributions of mutations, deletions and insertions stabilize. This, in turn, could be useful to inform the inheritance of genetic conditions, among syndromes, allergies and diseases.

6 Appendix

6.1 Part I: Reflection

The class section on the Rod-Cutting problem provided me with the most relevant lessons for this assignment. It was one of our first sessions introducing to Dynamic Programming, and it demonstrated to me how it provides an advantage over brute-force due to the overlapping subproblems, and other previous greedy approaches such as the one used in the knapsack problem, by allowing us to test different configurations. We also explored the trade-off of efficiency and optimality offered by each strategy, which was pivotal to develop my Computational Critique arguments for this assignment.

6.2 Part II: LO and HC Applications

- *#professionalism:* The assignment has included all parts, including videos, explanations, code, and visualizations. I have included my Appendix both as a combined PDF and an .ipynb file in the .zip format. Completed the AI statement. (34 words)
- *#cs110-AlgoDataStruct:* In my video, I explained an overview of both my global and local approaches including information about my data structures. In my report, I developed a comprehensive explanation

about my greedy and dynamic programming strategies, explaining particular features of each (i.e. greedy property, overlapping subproblems). (50 words)

- *#cs110-CodeReadability:* Throughout the assignment, I wrote code that uses appropriate function and variable naming, comments, and docstrings. Added inline comments to clarify details of the implementation. Included test cases using assert statements across all my tests in searching for all possible LCS code. (49 words)
- *#cs110-ComplexityAnalysis:* In the report I provided an in depth overview of the theoretical complexity analysis, deriving the time complexity it both through a theoretical and empirical approach. In the former, I included the practical interpretation and in the latter I considered the comparison of multiple inputs (44 words)
- *#cs110-ComputationalCritique:* I provided a critique of both local vs global strategies, as well as recursive vs iterative implementations of Dynamic Programming. Later in the analysis section, I developed a conclusion on which approach would be more adequate given a particular context from the problem statement, the biotechnology company. (46 words)
- *#cs110-PythonProgramming:* Ensured my code was functional and robust by including the described number of test cases, especially while developing a consistent schedule regardless of the input order. With the experiments, I made sure that the visualizations were plotted for a broad range of inputs and included titles, axis descriptions, and legends. (50 words)

Additional HC *#probability:* Provided an exploration of probability estimations through the concept of Markov matrices. First computed the likelihood for a particular type of edit (deletion, insertion, mutation) considering the sample space of the Levenshtein distance, and later expanded it to to consider the cases in which a character remains the same. Developed sample matrices of probabilities and averaged it to approximate the true independent mutation probabilities for each string. (68 words)

6.3 Part III: Python Code

Included as a merged PDF and through an additional .ipynb file.

7 AI Statement

I used Grammarly AI to support my writing and prevent potential typos and grammatical mistakes.

CS110_Code_Final

December 19, 2024

1 Question 1

Write Python code that, given any two arbitrary strings, outputs all of the Longest Common Subsequences (LCSs) for those two strings and their corresponding lengths. Include several test cases that demonstrate that your code is correct.

[41]:
```
# Helper functions
def lcs_length(x, y):
    """
    Computes the Longest Common Subsequence (LCS) length matrix and backtracking␣
↪matrix for two input strings.

    Parameters:
    x (str): First input string.
    y (str): Second input string.

    Returns:
    tuple: A tuple containing two 2D matrices:
        - C: LCS length matrix, where C[i][j] represents the length of LCS of␣
↪the first i characters of x and the first j characters of y.
        - B: Backtracking matrix, which stores directions for reconstructing LCS.
↪ Possible directions are:
            - "" for diagonal (match).
            - "↑" for up (LCS length preserved by excluding the current character␣
↪of x).
            - "←" for left (LCS length preserved by excluding the current␣
↪character of y).
    """
    m = len(x)
    n = len(y)

    # Initialize backtracking matrix B with empty lists for multiple directions␣
↪and LCS length matrix C with zeros
    B = [
        [[] for _ in range(n + 1)] for _ in range(m + 1)
    ]  # Use lists for multiple directions -> inspiration from multivariable␣
↪calculus and directional derivatives
```

```
    C = [[0 for _ in range(n + 1)] for _ in range(m + 1)]  # LCS lengths

    # Fill the LCS length matrix and backtracking matrix
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if x[i - 1] == y[j - 1]:  # Characters match
                C[i][j] = C[i - 1][j - 1] + 1
                B[i][j] = [""]  # Diagonal direction for match
            else:
                C[i][j] = max(
                    C[i - 1][j], C[i][j - 1]
                )  # Find the direction with more frequent common characters
                if (
                    C[i - 1][j] == C[i][j]
                ):  # Include '↑' if moving up preserves the LCS length
                    B[i][j].append("↑")
                if (
                    C[i][j - 1] == C[i][j]
                ):  # Include '←' if moving left preserves the LCS length
                    B[i][j].append("←")

    return C, B


def print_lcs(B, X, path, i, j, results):
    """
    Recursively constructs all LCS sequences from the backtracking matrix and␣
 ↪stores them in the results set.

    Parameters:
    B (list of lists): The backtracking matrix.
    X (str): The input string for which LCS is calculated.
    path (list): Current path being explored (stores LCS characters).
    i (int): Current index in string X.
    j (int): Current index in string Y.
    results (set): A set to store unique LCS sequences.

    Returns:
    None: The results set is updated with LCS sequences.
    """
    if i == 0 or j == 0:  # Base case: reached the beginning of one of the␣
 ↪strings
        if len(path) > 0:  # Add the reversed path as a valid LCS
            results.add("".join(path[::-1]))  # Add the reversed path as a result
        return

    # Explore all valid directions
```

```
    for direction in B[i][j]:
        if direction == "":  # Diagonal direction (match)
            path.append(X[i - 1])  # Add the current matching character
            print_lcs(B, X, path, i - 1, j - 1, results)  # Recurse with updated␣
↪indices
            path.pop()  # Backtrack after exploring this path
        elif direction == "↑":  # Upward direction (skip current character of X)
            print_lcs(B, X, path, i - 1, j, results)
        elif direction == "←":  # Leftward direction (skip current character of␣
↪Y)
            print_lcs(B, X, path, i, j - 1, results)


# Input strings
X = "ABCBDAB"
Y = "BDCABA"

# Compute LCS length and backtracking matrices
c, b = lcs_length(X, Y)
results = set()
print_lcs(b, X, [], len(X), len(Y), results)

# Print results
print("LCS Length Matrix (c):")
for row in c:
    print(row)

print("\nBacktracking Matrix (b):")
for row in b:
    print(row)

# Print all LCS sequences found
print("All LCS:", sorted(results))
```

```
LCS Length Matrix (c):
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 1, 1]
[0, 1, 1, 1, 1, 2, 2]
[0, 1, 1, 2, 2, 2, 2]
[0, 1, 1, 2, 2, 3, 3]
[0, 1, 2, 2, 2, 3, 3]
[0, 1, 2, 2, 3, 3, 4]
[0, 1, 2, 2, 3, 4, 4]

Backtracking Matrix (b):
[[], [], [], [], [], [], []]
[[], ['↑', '←'], ['↑', '←'], ['↑', '←'], [''], ['←'], ['']]
```

```
[[], [''], ['←'], ['←'], ['↑', '←'], [''], ['←']]
[[], ['↑'], ['↑', '←'], [''], ['←'], ['↑', '←'], ['↑', '←']]
[[], [''], ['↑', '←'], ['↑'], ['↑', '←'], [''], ['←']]
[[], ['↑'], [''], ['↑', '←'], ['↑', '←'], ['↑'], ['↑', '←']]
[[], ['↑'], ['↑'], ['↑', '←'], [''], ['↑', '←'], ['']]
[[], [''], ['↑'], ['↑', '←'], ['↑'], [''], ['↑', '←']]
All LCS: ['BCAB', 'BCBA', 'BDAB']
```

[42]:
```python
def longest_common_subsequences(x, y):
    """Gives the length of the longest common substring between strings x and y.

    Inputs
    ----------
    x, y: strings
        Strings to compute the LCS.

    Returns
    ----------
    all_lcs: tuple ([LCS1, LCS2, ...], len(LCS1))
        Tuple of a list of all the possible LCS and the corresponding length␣
 ↪(size).
    """
    c, b = lcs_length(x, y)
    results = set()
    path = []

    # Compute all LCS sequences
    print_lcs(b, x, path, len(x), len(y), results)

    # Organize the LCSs by the order of discovery and sort alphabetically for␣
 ↪consistency of test cases
    if len(results) > 0:
        all_lcs = tuple([sorted(list(results)), c[-1][-1]])
    else:
        all_lcs = tuple([None, c[-1][-1]])

    return all_lcs


# Input strings
X = "ABCBDAB"
Y = "BDCABA"
print(longest_common_subsequences(X, Y))
```

```
(['BCAB', 'BCBA', 'BDAB'], 4)
```

1.1 Test Cases

[43]:
```
x1, y1 = "ABCBDAB", "BDCABA"
x2, y2 = "abc", ""
x3, y3 = "abc", "a"
x4, y4 = "abc", "ac"

assert longest_common_subsequences(x1, y1) == (['BCAB', 'BCBA', 'BDAB'], 4)
assert longest_common_subsequences(x2, y2) == (None, 0)
assert longest_common_subsequences(x3, y3) == (["a"], 1)
assert longest_common_subsequences(x4, y4) == (["ac"], 2)
```

[44]:
```
# both inputs are null
X = ""
Y = ""

c, b = lcs_length(X, Y)
results = set()
print_lcs(b, X, [], len(X), len(Y), results)

print("LCS Length Matrix (c):")
for row in c:
    print(row)

print("\nBacktracking Matrix (b):")
for row in b:
    print(row)

# Print results
print("All LCS:", longest_common_subsequences(X, Y))
assert longest_common_subsequences(X, Y) == (None, 0)
```

```
LCS Length Matrix (c):
[0]

Backtracking Matrix (b):
[[]]
All LCS: (None, 0)
```

[45]:
```
# words with repeated letters (we can handle repeated letters because they are␣
 ↪placed in different positions of the matrix)
X = "MATTER"
Y = "LETTER"  # its breaking because results is not a set (and when im exploring␣
 ↪different pathways, i can end up with the same LCs repeated multiple times)

c, b = lcs_length(X, Y)
results = set()
print_lcs(b, X, [], len(X), len(Y), results)
```

```
print("LCS Length Matrix (c):")
for row in c:
    print(row)

print("\nBacktracking Matrix (b):")
for row in b:
    print(row)

# Print results
print("All LCS:", longest_common_subsequences(X, Y))
assert longest_common_subsequences(X, Y) == (['TTER'], 4)
```

```
LCS Length Matrix (c):
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 1, 1, 1]
[0, 0, 0, 1, 2, 2, 2]
[0, 0, 1, 1, 2, 3, 3]
[0, 0, 1, 1, 2, 3, 4]

Backtracking Matrix (b):
[[], [], [], [], [], [], []]
[[], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←']]
[[], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←']]
[[], ['↑', '←'], ['↑', '←'], [''], [''], ['←'], ['←']]
[[], ['↑', '←'], ['↑', '←'], [''], [''], ['←'], ['←']]
[[], ['↑', '←'], [''], ['↑', '←'], ['↑'], [''], ['←']]
[[], ['↑', '←'], ['↑'], ['↑', '←'], ['↑'], ['↑'], ['']]
All LCS: (['TTER'], 4)
```

[46]:
```
# words with repeated letters, the longest lC containing 1 char
X = "NULL"
Y = "LETTER"

c, b = lcs_length(X, Y)
results = set()
print_lcs(b, X, [], len(X), len(Y), results)

print("LCS Length Matrix (c):")
for row in c:
    print(row)

print("\nBacktracking Matrix (b):")
for row in b:
    print(row)
```

```
# Print results
print("All LCS:", longest_common_subsequences(X, Y))
assert longest_common_subsequences(X, Y) == (['L'], 1)
```

```
LCS Length Matrix (c):
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0]
[0, 1, 1, 1, 1, 1, 1]
[0, 1, 1, 1, 1, 1, 1]

Backtracking Matrix (b):
[[], [], [], [], [], [], []]
[[], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←']]
[[], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←']]
[[], [''], ['←'], ['←'], ['←'], ['←'], ['←']]
[[], [''], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←']]
All LCS: (['L'], 1)
```

[47]:
```
# Mixed uppercase and lower-case
X = "DYnamIC"
Y = "MecHanIC"

c, b = lcs_length(X, Y)
results = set()
print_lcs(b, X, [], len(X), len(Y), results)

print("LCS Length Matrix (c):")
for row in c:
    print(row)

print("\nBacktracking Matrix (b):")
for row in b:
    print(row)

# Print results
print("All LCS:", longest_common_subsequences(X, Y))
assert longest_common_subsequences(X, Y) == (['aIC', 'nIC'], 3)
```

```
LCS Length Matrix (c):
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 1, 1]
[0, 0, 0, 0, 0, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 1, 1, 2, 2]
```

```
[0, 0, 0, 0, 0, 1, 1, 2, 3]

Backtracking Matrix (b):
[[], [], [], [], [], [], [], [], []]
[[], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'],
['↑', '←'], ['↑', '←']]
[[], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'],
['↑', '←'], ['↑', '←']]
[[], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], [''], ['←'],
['←']]
[[], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], [''], ['↑', '←'], ['↑',
'←'], ['↑', '←']]
[[], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑'], ['↑', '←'], ['↑',
'←'], ['↑', '←']]
[[], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑'], ['↑', '←'], [''],
['←']]
[[], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑'], ['↑', '←'], ['↑'],
['']]
All LCS: (['aIC', 'nIC'], 3)
```

[48]:
```python
# Composite words
X = "eco-friendly"
Y = "editor-in-chief"

c, b = lcs_length(X, Y)
results = set()
print_lcs(b, X, [], len(X), len(Y), results)

print("LCS Length Matrix (c):")
for row in c:
    print(row)

print("\nBacktracking Matrix (b):")
for row in b:
    print(row)

# Print results
print("All LCS:", longest_common_subsequences(X, Y))
assert longest_common_subsequences(X, Y) == (['eo-ie', 'eo-in', 'eorie',␣
 ↪'eorin'], 5)
```

```
LCS Length Matrix (c):
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2]
[0, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
[0, 1, 1, 1, 1, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3]
[0, 1, 1, 1, 1, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4]
```

```
[0, 1, 1, 1, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4]
[0, 1, 1, 2, 2, 2, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4]
[0, 1, 1, 2, 2, 2, 3, 3, 4, 4, 4, 4, 4, 4, 5, 5]
[0, 1, 1, 2, 2, 2, 3, 3, 4, 5, 5, 5, 5, 5, 5, 5]
[0, 1, 2, 2, 2, 2, 3, 3, 4, 5, 5, 5, 5, 5, 5, 5]
[0, 1, 2, 2, 2, 2, 3, 3, 4, 5, 5, 5, 5, 5, 5, 5]
[0, 1, 2, 2, 2, 2, 3, 3, 4, 5, 5, 5, 5, 5, 5, 5]

Backtracking Matrix (b):
[[], [], [], [], [], [], [], [], [], [], [], [], [], [], []]
[[], [''], ['←'], ['←'], ['←'], ['←'], ['←'], ['←'], ['←'], ['←'], ['←'],
['←'], ['←'], ['←'], [''], ['←']]
[[], ['↑'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑',
'←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], [''], ['←'], ['←'], ['←'], ['←']]
[[], ['↑'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], [''], ['←'], ['←'], ['←'],
['←'], ['←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←']]
[[], ['↑'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑'], ['↑', '←'], [''], ['←'],
['←'], [''], ['←'], ['←'], ['←'], ['←'], ['←']]
[[], ['↑'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑'], ['↑', '←'], ['↑'], ['↑',
'←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'],
['']]
[[], ['↑'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑'], [''], ['↑', '←'], ['↑',
'←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'],
['↑']]
[[], ['↑'], ['↑', '←'], [''], ['←'], ['↑', '←'], ['↑'], ['↑', '←'], [''],
['←'], ['←'], ['←'], ['←'], [''], ['←'], ['↑', '←']]
[[], [''], ['↑', '←'], ['↑'], ['↑', '←'], ['↑', '←'], ['↑'], ['↑', '←'], ['↑'],
['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], [''], ['←']]
[[], ['↑'], ['↑', '←'], ['↑'], ['↑', '←'], ['↑', '←'], ['↑'], ['↑', '←'], ['↑'],
[''], ['←'], ['←'], ['←'], ['←'], ['↑', '←'], ['↑', '←']]
[[], ['↑'], [''], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑'], ['↑', '←'], ['↑'],
['↑'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←']]
[[], ['↑'], ['↑'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑'], ['↑', '←'], ['↑'],
['↑'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←']]
[[], ['↑'], ['↑'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑'], ['↑', '←'], ['↑'],
['↑'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←'], ['↑', '←']]
All LCS: (['eo-ie', 'eo-in', 'eorie', 'eorin'], 5)
```

[49]:
```
# two gene sequence examples (a and b)
X =␣
 ↪"ATGGTGCGAAAGCATCTCTTTTCGTGGCGTGATAAGTTTTATGGTATCCCCGGACGTTGGCTACTACAATTCTCCGAAGTATAAGTGAGTAG
Y =␣
 ↪"TGGTGCGAAAGCATCTCTTTTCCGTGGCGTATAGTTTTATGGTATCCCCGGAACGCTGGCTACTACAATCTCCGAAGTATAGAGTGAGTAGA

longest_common_subsequences(X, Y)
```

[49]:
```
(['TGGTGCGAAAGCATCTCTTTTCGTGGCGTATAGTTTTATGGTATCCCCGGACGTGGCTACTACAATCTCCGAAGTAT
AAGTGAGTAGATTTAATAACAGAGGGGTCGTGACGCATTAGCACCAACTGAATCAACGATAACTAACGTGGTTTCAGTGA
CTATGGGCAAAGGATGAACATTTTCGAGCACTCTAATAATGACGTGACAATATGAACCCACCGTCATCTTGAACTCCT',
  'TGGTGCGAAAGCATCTCTTTTCGTGGCGTATAGTTTTATGGTATCCCCGGACGTGGCTACTACAATCTCCGAAGTAT
AAGTGAGTAGATTTAATAACAGAGGGGTCGTGACGCATTAGCACCAACTGAATCAACGATAACTAACGTGGTTTCAGTGA
CTATGGGCAAAGGATGAACATTTTCGAGCACTCTAATAATGACGTGACAATATGAATCCACCGTCATCTTGAACTCCT',
  'TGGTGCGAAAGCATCTCTTTTCGTGGCGTATAGTTTTATGGTATCCCCGGACGTGGCTACTACAATCTCCGAAGTAT
AAGTGAGTAGATTTAATAACAGAGGGGTCGTGACGCATTAGCACCAACTGAATCAACGATAACTAACGTGGTTTCAGTGA
CTATGGGCAAAGGATGAACATTTTCGAGCAGTCTAATAATGACGTGACAATATGAACCCACCGTCATCTTGAACTCCT',
  'TGGTGCGAAAGCATCTCTTTTCGTGGCGTATAGTTTTATGGTATCCCCGGACGTGGCTACTACAATCTCCGAAGTAT
AAGTGAGTAGATTTAATAACAGAGGGGTCGTGACGCATTAGCACCAACTGAATCAACGATAACTAACGTGGTTTCAGTGA
CTATGGGCAAAGGATGAACATTTTCGAGCAGTCTAATAATGACGTGACAATATGAATCCACCGTCATCTTGAACTCCT']
,
 235)
```

2 Question 2

[50]:
```
genes = [
    (
        "a",
    ␣
↪"ATGGTGCGAAAGCATCTCTTTTCGTGGCGTGATAAGTTTTATGGTATCCCCGGACGTTGGCTACTACAATTCTCCGAAGTATAAGTGAGTAG
    ),
    (
        "b",
    ␣
↪"TGGTGCGAAAGCATCTCTTTTCCGTGGCGTATAGTTTTATGGTATCCCCGGAACGCTGGCTACTACAATCTCCGAAGTATAGAGTGAGTAGA
    ),
    (
        "c",
    ␣
↪"TCTGTGCGATATACATCTCTATCGTTGCGGTATGTTTTATGTGCATCACCCCACGCGCTGGCTACAGTACAATCTGCTGGAAGTACTAGGTG
    ),
    (
        "d",
    ␣
↪"ATGAGGCGCAAAATTCTCTTTCTCGTGGCGCTGATTAAGTTTTATGTATCCCCGGACGTTGGCTACTGACAATTGCTCCGAAGTATAAAGTA
    ),
    (
        "e",
    ␣
↪"TGGTGCGATATACATCTCTTTTCGTGCGTATGTTTTATGGTGATCACCCGGAACCGCTGGCTACATACAATCTCTGGAAGTACTAGGTGGTA
    ),
    (
        "f",
```

```
        ␣
↪"GGGGGAAAGCGATCCCTTATCGTGGCTGTGATAAGTTTTTATCGGGTATCCGCCGGACGTTGGCGTACTACAATTCTCCGAAGTTAAGTGAG
    ),
    (
        "g",
        ␣
↪"TGGTGCGATATACATCCTCTTTTCGTGCGTATGTTTTAGGTACACCGGATACGCCTGGCTTACAAGTACCAATCTCTGAGAAGTCACTGAGG
    ),
]
```

Strings which are more closely related to each other will have longer LCS between them

[51]:
```
import numpy as np
import matplotlib.pyplot as plt


def generate_set_strings_matrix(genes):
    m = len(genes)
    len_lcs_matrix = np.zeros((m, m), dtype=int)

    for i in range(m):  # for a gene in genes
        for j in range(m):

            # print(f"Combination between genes: {genes[i][0]}, {genes[j][0]}")
            x = genes[i][1]
            y = genes[j][1]

            # print(len(x), len(y))
            # find matrix of subsequence lengths
            c, _ = lcs_length(x, y)

            # populate row
            len_lcs_matrix[i][j] = c[-1][-1]

        # print('\n')
    return len_lcs_matrix


len_lcs_matrix = generate_set_strings_matrix(genes)

print(len_lcs_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]]
```

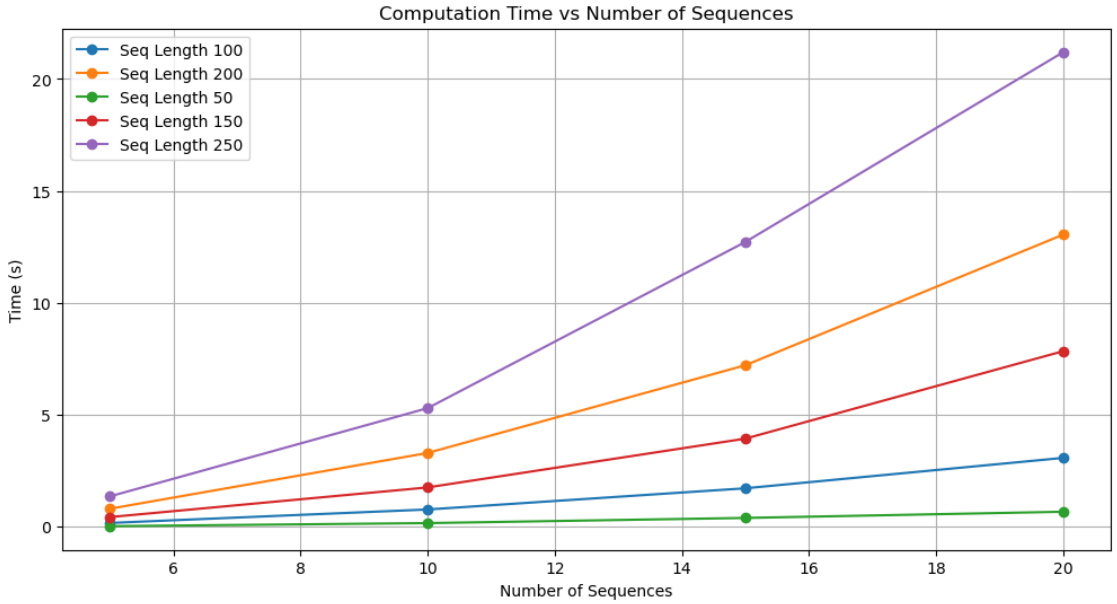
```
 [251 223 195 241 205 280 208]
 [212 220 229 205 243 208 269]]
```

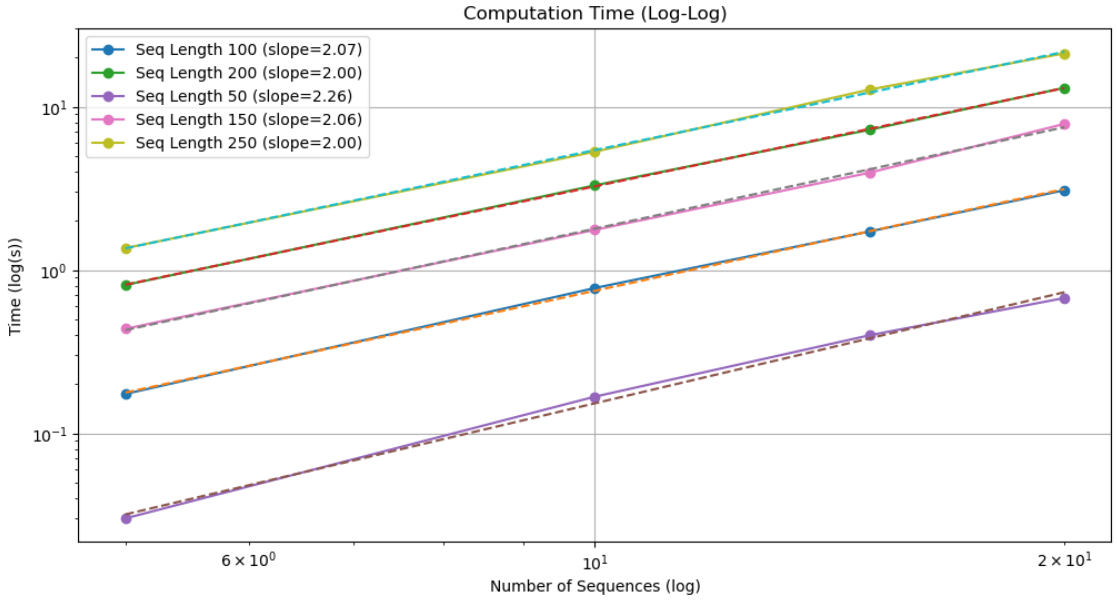
[52]:
```
print(len_lcs_matrix.shape)
```

```
(7, 7)
```

[53]:
```
import random
import time
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import linregress

# Run experiments and collect results for plotting
def run_experiments_with_plotting():
    sequence_lengths = [50, 100, 150, 200, 250]  # Different sequence lengths
    num_sequences = [x for x in range(5, 25, 5)]  # Different numbers of␣
 ↪sequences

    results = {'sequence_lengths': [], 'num_sequences': [], 'times': []}

    for seq_len in sequence_lengths:
        for num_seq in num_sequences:
            # Generate random gene sequences
            genes = [(f"gene_{i}", ''.join(random.choices("ACGT", k=seq_len)))␣
 ↪for i in range(num_seq)]

            # Measure the time taken to compute the LCS matrix
            start_time = time.time()
            _ = generate_set_strings_matrix(genes)
            elapsed_time = time.time() - start_time

            # Log the results
            results['sequence_lengths'].append(seq_len)
            results['num_sequences'].append(num_seq)
            results['times'].append(elapsed_time)

    # Generate plots
    plot_results(results)

# Plot the results
def plot_results(results):
    sequence_lengths = np.array(results['sequence_lengths'])
    num_sequences = np.array(results['num_sequences'])
    times = np.array(results['times'])

    # Linear plot of computation time
    plt.figure(figsize=(12, 6))
```

```
    for seq_len in set(sequence_lengths):
        mask = sequence_lengths == seq_len
        plt.plot(num_sequences[mask], times[mask], marker='o', label=f"Seq␣
↪Length {seq_len}")

    plt.title("Computation Time vs Number of Sequences")
    plt.xlabel("Number of Sequences")
    plt.ylabel("Time (s)")
    plt.legend()
    plt.grid(True)
    plt.show()

    # Log-log plot of computation time with regression
    plt.figure(figsize=(12, 6))
    for seq_len in set(sequence_lengths):
        mask = sequence_lengths == seq_len
        log_num_sequences = np.log(num_sequences[mask])
        log_times = np.log(times[mask])
        slope, intercept, _, _, _ = linregress(log_num_sequences, log_times)
        plt.loglog(num_sequences[mask], times[mask], marker='o', label=f"Seq␣
↪Length {seq_len} (slope={slope:.2f})")
        plt.loglog(num_sequences[mask], np.exp(intercept) * num_sequences[mask]␣
↪** slope, linestyle='--')

    plt.title("Computation Time (Log-Log)")
    plt.xlabel("Number of Sequences (log)")
    plt.ylabel("Time (log(s))")
    plt.legend()
    plt.grid(True)
    plt.show()

if __name__ == '__main__':
    run_experiments_with_plotting()
```





3 Question 3

3.1 (a) Local Strategy

Recyling Implementation of the Max Heap from Task Scheduler Assignment

[54]:
```python
class Node:
    def __init__(self, i, j, lcs_length):
        self.i = i  # Index of the first sequence
        self.j = (
            j  # Index of the second sequence (which will become the parent␣
 ↪sequence)
        )
        self.lcs_length = lcs_length  # LCS score between the two sequences
        self.parent = None  # Reference to parent node
        self.children = []  # List of child nodes

    def __repr__(self):
        return f"Node(i={self.i}, j={self.j}, lcs_length={self.lcs_length})"

    def __lt__(self, other):
        # Priority queue uses this to compare nodes (max heap based on LCS score)
        return self.lcs_length > other.lcs_length
```

[55]:
```python
class MaxHeapq:
    """
    A class that implements properties and methods
    that support a max priority queue data structure.

    Attributes
    ----------
    heap : list
        A Python list where key values in the max heap are stored.
    heap_size : int
        An integer counter of the number of keys present in the max heap.
    """

    def __init__(self):
        """
        Initializes an empty MaxHeapq instance.

        Parameters
        ----------
        None
        """
        self.heap = []
        self.heap_size = 0

    def is_max_heap(self):
        """
        Checks if the current list satisfies the max heap property.

        Returns
```

```
        -------
        bool
            True if the list is a max heap, False otherwise.
        """
        a = self.heap
        n = self.heap_size

        # Check each node to ensure it is greater than its children
        for i in range(n):
            left_child = self.left(i)
            right_child = self.right(i)

            # Check if left child exists and is greater than parent
            if left_child < n and a[i].lcs_length < a[left_child].lcs_length:
                return False

            # Check if right child exists and is greater than parent
            if right_child < n and a[i].lcs_length < a[right_child].lcs_length:
                return False
        return True

    def left(self, i):
        return 2 * i + 1

    def right(self, i):
        return 2 * i + 2

    def parent(self, i):
        return (i - 1) // 2

    def heappush(self, key):
        """
        Inserts a key into the priority queue.

        Parameters
        ----------
        key : Node
            The Node object to be inserted.

        Returns
        -------
        None
        """
        # append a placeholder for new key and then increase its value to␣
↪correct position
        self.heap.append(Node(i=-1, j=-1, lcs_length=float("-inf")))
        self.increase_key(self.heap_size, key)
```

```
        self.heap_size += 1

    def increase_key(self, i, key):
        """
        Modifies the value of a key in a max priority queue with a higher value.

        Parameters
        ----------
        i : int
            The index of the key to be modified.
        key : Node
            The new key value (must have a higher lcs_length).

        Raises
        ------
        ValueError
            If new key has a smaller lcs_length than the current key.

        Returns
        -------
        None
        """
        if key.lcs_length < self.heap[i].lcs_length:
            raise ValueError("new key is smaller than the current key")

        self.heap[i] = key
        while i > 0 and self.heap[self.parent(i)].lcs_length < self.heap[i].
↪lcs_length:
            j = self.parent(i)
            self.heap[i], self.heap[j] = self.heap[j], self.heap[i]
            i = j

    def heapify(self, i):
        """
        Creates a max heap from the index given by ensuring that subtree rooted␣
↪at index i satisfies max heap property.

        Parameters
        ----------
        i : int
            The index of the root node of the subtree to be heapified.

        Returns
        -------
        None
        """
        l = self.left(i)
```

```
        r = self.right(i)
        largest = i

        if l < self.heap_size and self.heap[l].lcs_length > self.heap[largest].
↪lcs_length:
            largest = l

        if r < self.heap_size and self.heap[r].lcs_length > self.heap[largest].
↪lcs_length:
            largest = r

        if largest != i:
            self.heap[i], self.heap[largest] = self.heap[largest], self.heap[i]
            self.heapify(largest)

    def heappop(self):
        """
        Returns and removes the largest key in the max priority queue.

        Raises
        ------
        ValueError
            If there are no keys in priority queue (heap underflow).

        Returns
        -------
        Node
            The maximum value (Node object) extracted from the heap.
        """
        if self.heap_size < 1:
            raise ValueError("Heap underflow: There are no keys in priority␣
↪queue.")

        max_value = self.heap[0]
        self.heap[0] = self.heap[-1]
        self.heap.pop()
        self.heap_size -= 1
        self.heapify(0)
        return max_value
```

[56]:
```
def greedy_max_LCS(selected_root, lcs_matrix):
    """
    Build a tree using a greedy algorithm, maximizing the LCS values
    for connections between sequences.

    Parameters:
        lcs_matrix (list[list[int]]): The LCS similarity matrix.
```

```
    Returns:
        Node: The root of the constructed tree.
    """
    n = len(lcs_matrix)
    visited = set()  # Tracks visited nodes to prevent revisiting
    priority_queue = MaxHeapq()  # Max-heap to prioritize nodes by LCS value

    # Initialize the root node based on the global maximum
    root = Node(i=selected_root, j=-1, lcs_length=0)
    visited.add(selected_root)
    priority_queue.heappush(root)

    # Build the tree by iteratively adding nodes
    while priority_queue.heap_size > 0:
        current_node = priority_queue.heappop()

        # Identify the top two unvisited sequences with the highest LCS scores
        top1 = (-1, float("-inf"))  # Highest score (sequence index, score)
        top2 = (-1, float("-inf"))  # Second-highest score

        for sequence_index in range(n):
            if sequence_index not in visited:
                lcs_score = lcs_matrix[current_node.i][sequence_index]
                if lcs_score > top1[1]:
                    top2 = top1
                    top1 = (sequence_index, lcs_score)
                elif lcs_score > top2[1]:
                    top2 = (sequence_index, lcs_score)

        # Add children to the tree for valid top sequences
        for top in [top1, top2]:
            if top[0] != -1:  # Ensure the index is valid
                child_node = Node(i=top[0], j=current_node.i, lcs_length=top[1])
                child_node.parent = current_node
                current_node.children.append(child_node)

                # Mark as visited and push to the priority queue
                visited.add(top[0])
                priority_queue.heappush(child_node)

    return root

def print_tree(node, genes, depth=0):
    """
    Print the tree structure in a hierarchical format.
```

```
    Parameters:
        node (Node): The current node being printed.
        genes (list[tuple]): List of gene sequences as (name, sequence).
        depth (int): Current depth in the tree for indentation.
    """
    name = genes[node.i][0]  # Gene name
    print(" " * depth * 2 + f"{name} (LCS: {node.lcs_length})")
    for child in node.children:
        print_tree(child, genes, depth + 1)

def calculate_total_cost(node):
    """
    Calculate the total cost (sum of LCS lengths) for the entire tree.

    Parameters:
        node (Node): The root node of the tree.

    Returns:
        int: Total LCS cost of the tree.
    """
    if node is None:
        return 0

    cost = node.lcs_length
    for child in node.children:
        cost += calculate_total_cost(child)

    return cost

# Construct the tree using the greedy LCS algorithm
selected_root = 6
root = greedy_max_LCS(selected_root, len_lcs_matrix)

# Print the tree structure
print("Tree Structure:")
print_tree(root, genes)

# Calculate and display the total tree cost
total_cost = calculate_total_cost(root)
print(f"\nTotal tree cost: {total_cost}")
```

```
Tree Structure:
g (LCS: 0)
  e (LCS: 243)
    b (LCS: 227)
    a (LCS: 214)
  c (LCS: 229)
    d (LCS: 197)
```

```
    f (LCS: 195)

Total tree cost: 1305
```

3.2 (b) Global Strategy

3.3 Iterative Approach

[57]:
```python
def max_LCS_tree_iterative(weights):
    """
    Computes the maximum LCS tree using a dynamic programming approach.

    Args:
        weights (list of list of int): A matrix where weights[i][j] represents␣
↪the LCS weight between
                                       sequences i and j.

    Returns:
        tuple: A tuple containing the DP table (dp) and the root table (root)␣
↪where:
               - dp[i][j] is the maximum total weight for the subtree spanning␣
↪sequences i to j.
               - root[i][j] is the index of the root for the subtree spanning␣
↪sequences i to j.
    """
    n = len(weights)
    dp = [[0 for _ in range(n)] for _ in range(n)]
    root = [[-1 for _ in range(n)] for _ in range(n)]

    # Build the DP and Root tables
    for i in range(n):
        dp[i][i] = 0
        root[i][i] = i

    for size in range(2, n + 1):
        for i in range(n - size + 1):
            j = i + size - 1
            max_weight = float('-inf')

            for r in range(i, j + 1):
                left_size = r - i
                right_size = j - r

                if abs(left_size - right_size) > 1:
                    continue

                left_weight = dp[i][r - 1] if r > i else 0
```

```
                right_weight = dp[r + 1][j] if r < j else 0

                root_weight = sum(weights[r][k] for k in range(i, j + 1) if k !=␣
↪r)
                total_weight = left_weight + right_weight + root_weight

                if total_weight > max_weight:
                    max_weight = total_weight
                    dp[i][j] = max_weight
                    root[i][j] = r


    return dp, root

def reconstruct_tree_with_nodes(root, weights, i, j, parent=None):
    """
    Reconstructs the tree structure based on the root table generated by the␣
↪max_LCS_tree_iterative function.

    Args:
        root (list of list of int): The root table indicating the root for each␣
↪subtree.
        weights (list of list of int): The LCS weight matrix.
        i (int): Starting index of the subtree.
        j (int): Ending index of the subtree.
        parent (int, optional): Parent node index for the current subtree.␣
↪Defaults to None.

    Returns:
        Node: The root node of the reconstructed tree.
    """
    if i > j:
        return None

    r = root[i][j]  # Get the root of the current subtree
    # Compute LCS length based on the parent-child relationship
    lcs_length = 0 if parent is None else weights[r][parent]
    node = Node(i=r, j=parent, lcs_length=lcs_length)  # Use `r` as the current␣
↪node

    # Recurse to build left and right subtrees
    left_child = reconstruct_tree_with_nodes(root, weights, i, r - 1, r)
    right_child = reconstruct_tree_with_nodes(root, weights, r + 1, j, r)

    # Attach children to the current node
    if left_child:
        left_child.parent = node
```

```
        node.children.append(left_child)
    if right_child:
        right_child.parent = node
        node.children.append(right_child)

    return node

def print_tree(node, genes, depth=0):
    """
    Prints the tree structure in a hierarchical format.

    Args:
        node (Node): The root node of the tree to print.
        genes (list of tuple): List of gene names and sequences.
        depth (int, optional): The current depth in the tree. Defaults to 0.
    """
    name = genes[node.i][0]  # Get the gene name for the current node
    lcs_info = f"(LCS: {node.lcs_length})" if node.lcs_length > 0 else ""
    print(" " * depth * 2 + f"{name} {lcs_info}")
    for child in node.children:
        print_tree(child, genes, depth + 1)

def calculate_total_cost(node):
    """
    Calculates the total LCS cost for the entire tree.

    Args:
        node (Node): The root node of the tree.

    Returns:
        int: The total LCS cost of the tree.
    """
    if node is None:
        return 0

    cost = node.lcs_length
    for child in node.children:
        cost += calculate_total_cost(child)

    return cost

# Example usage:
dp, root = max_LCS_tree_iterative(len_lcs_matrix)
tree = reconstruct_tree_with_nodes(root, len_lcs_matrix, 0, len(len_lcs_matrix)␣
 ↪- 1)

print("\nDP Table:")
```

```python
for row in dp:
    print(row)

print("\nRoot Table:")
for row in root:
    print(row)

print("\nReconstructed Tree with Nodes:")
print_tree(tree, genes)

total_cost = calculate_total_cost(tree)
print(f"\nTotal cost of the tree: {total_cost}")
```

```
DP Table:
[0, 235, 446, 866, 1283, 1771, 2184]
[0, 0, 211, 408, 849, 1284, 1732]
[0, 0, 0, 197, 404, 850, 1294]
[0, 0, 0, 0, 207, 412, 863]
[0, 0, 0, 0, 0, 205, 413]
[0, 0, 0, 0, 0, 0, 208]
[0, 0, 0, 0, 0, 0, 0]

Root Table:
[0, 0, 1, 1, 2, 3, 3]
[-1, 1, 1, 2, 2, 3, 4]
[-1, -1, 2, 2, 3, 3, 4]
[-1, -1, -1, 3, 3, 4, 4]
[-1, -1, -1, -1, 4, 4, 5]
[-1, -1, -1, -1, -1, 5, 5]
[-1, -1, -1, -1, -1, -1, 6]

Reconstructed Tree with Nodes:
d
  b (LCS: 223)
    a (LCS: 235)
    c (LCS: 211)
  f (LCS: 241)
    e (LCS: 205)
    g (LCS: 208)

Total cost of the tree: 1323
```

3.4 Recursive Approach

[58]:
```python
def max_LCS_tree_recursive(lcs_matrix, i, j, memo, root):
    """
    Recursive function to calculate the maximum weight of the subtree
    for the range [i, j] and store intermediate results in memo.

    Args:
        lcs_matrix: An nxn matrix of LCS scores between node pairs.
        i: Start index of the current range.
        j: End index of the current range.
        memo: A dictionary for memoization of the maximum weights.
        root: A 2D array to store the root indices of subtrees.

    Returns:
        The maximum weight of the subtree in range [i, j].
    """
    # Base case: empty range
    if i > j:
        return 0

    # Base case: single node
    if i == j:
        root[i][j] = i
        return 0

    # Check memo
    if (i, j) in memo:
        return memo[(i, j)]

    max_weight = float('-inf')
    best_root = -1

    # Try every node in the range [i, j] as the root
    for r in range(i, j + 1):
        left_size = r - i
        right_size = j - r

        # Enforce balance condition
        if abs(left_size - right_size) > 1:
            continue

        # Recursively compute left and right subtree weights
        left_weight = max_LCS_tree_recursive(lcs_matrix, i, r - 1, memo, root)␣
↪if r > i else 0
        right_weight = max_LCS_tree_recursive(lcs_matrix, r + 1, j, memo, root)␣
↪if r < j else 0
```

```
        # Calculate root weight
        root_weight = sum(lcs_matrix[r][k] for k in range(i, j + 1) if k != r)

        # Total weight of the subtree
        total_weight = left_weight + right_weight + root_weight

        # Update maximum weight and root index
        if total_weight > max_weight:
            max_weight = total_weight
            best_root = r

    # Store results in memo and root table
    memo[(i, j)] = max_weight
    root[i][j] = best_root
    return max_weight

def reconstruct_tree_with_nodes(root, lcs_matrix, i, j, parent=None):
    """
    Reconstructs the tree using the root indices and LCS matrix,
    while building Node objects.

    Args:
        root: A 2D array containing root indices for subtrees.
        lcs_matrix: The LCS score matrix.
        i: Start index of the current subtree.
        j: End index of the current subtree.
        parent: Parent index for the current node (None for root).

    Returns:
        A Node object representing the tree.
    """
    if i > j:
        return None

    r = root[i][j]  # Get the root of the current subtree
    # Compute LCS length based on the parent-child relationship
    lcs_length = 0 if parent is None else lcs_matrix[r][parent]
    node = Node(i=r, j=parent, lcs_length=lcs_length)  # Use `r` as the current␣
↪node

    # Recurse to build left and right subtrees
    left_child = reconstruct_tree_with_nodes(root, lcs_matrix, i, r - 1, r)
    right_child = reconstruct_tree_with_nodes(root, lcs_matrix, r + 1, j, r)

    # Attach children to the current node
    if left_child:
```

```
        left_child.parent = node
        node.children.append(left_child)
    if right_child:
        right_child.parent = node
        node.children.append(right_child)

    return node

def print_tree(node, genes, depth=0):
    """Print the tree structure."""
    name = genes[node.i][0]  # Get the gene name for the current node
    lcs_info = f"(LCS: {node.lcs_length})" if node.lcs_length > 0 else ""
    print(" " * depth * 2 + f"{name} {lcs_info}")
    for child in node.children:
        print_tree(child, genes, depth + 1)

def calculate_total_cost(node):
    """Calculate the total cost of the tree."""
    if node is None:
        return 0

    cost = node.lcs_length
    for child in node.children:
        cost += calculate_total_cost(child)

    return cost

# Initialize the memo and root tables
memo = {}
root = [[-1 for _ in range(len(len_lcs_matrix))] for _ in␣
 ↪range(len(len_lcs_matrix))]

# Call the recursive function
max_LCS_tree_recursive(len_lcs_matrix, 0, len(len_lcs_matrix) - 1, memo, root)

# Reconstruct the tree using nodes
tree = reconstruct_tree_with_nodes(root, len_lcs_matrix, 0, len(len_lcs_matrix)␣
 ↪- 1)

# Print the reconstructed tree
print("\nReconstructed Tree with Nodes:")
print_tree(tree, genes)

# Calculate total cost
total_cost = calculate_total_cost(tree)
print(f"\nTotal cost of the tree: {total_cost}")
```

```
Reconstructed Tree with Nodes:
d
  b (LCS: 223)
    a (LCS: 235)
    c (LCS: 211)
  f (LCS: 241)
    e (LCS: 205)
    g (LCS: 208)

Total cost of the tree: 1323
```

4 Experiments

Experiments for each approach will be run considering a matrix of LCS as input.

- Input type 1: similar numbers of LCS as the genes example ($195 <$ sample < 250)
- Input type 2: low relationships (all samples lower than 195 but with a lower bound of 0)
- Input type 3: high relationships (all samples higher than 250 but with an upper bound of the same range as above, 445)

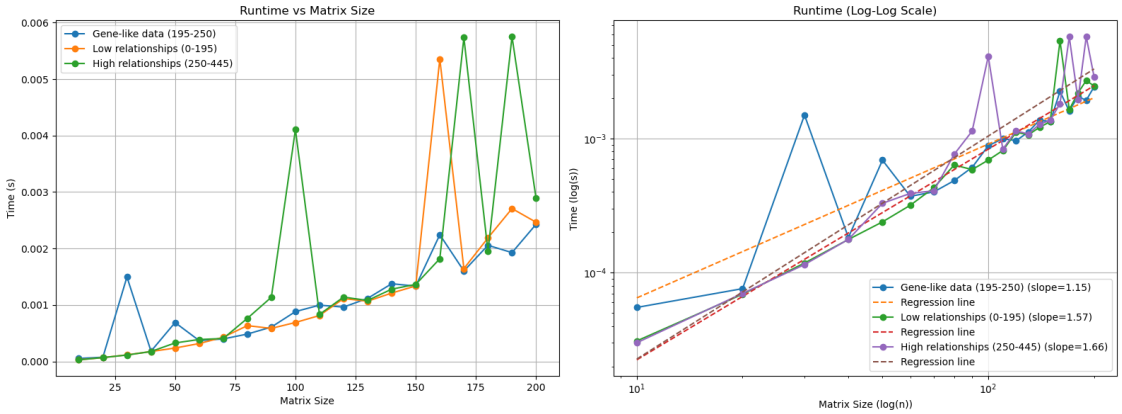
[59]:
```python
def generate_matrix(n, lower_bound, upper_bound):
    """
    Generate a random nxn matrix with values between lower_bound and upper_bound.

    Args:
        n: The size of the matrix.
        lower_bound: Minimum value for matrix entries.
        upper_bound: Maximum value for matrix entries.

    Returns:
        A randomly generated nxn matrix.
    """
    return [[random.randint(lower_bound, upper_bound) for j in range(n)] for i␣
↪in range(n)]
```

4.1 Greedy Approach

[60]:
```python
import time
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import linregress

def run_greedy_max_lcs_experiments_with_inputs():
    """
    Measure runtime of greedy_max_LCS for varying input sizes and types, then␣
↪plot results.
```

```
    Parameters:
        selected_root (int): Index of the initial root node.
    """
    input_types = [
        ("Gene-like data (195-250)", 195, 250),
        ("Low relationships (0-195)", 0, 195),
        ("High relationships (250-445)", 250, 445),
    ]
    sizes = range(10, 201, 10)  # Matrix sizes to test
    results = {name: {'sizes': [], 'times': []} for name, _, _ in input_types}

    for input_name, lower, upper in input_types:
        for size in sizes:
            # Generate random LCS matrix
            matrix = generate_matrix(size, lower, upper)

            selected_root = random.randint(0, size - 1)

            # runtime
            start_time = time.time()
            greedy_max_LCS(selected_root, matrix)
            elapsed_time = time.time() - start_time

            # results
            results[input_name]['sizes'].append(size)
            results[input_name]['times'].append(elapsed_time)

    plot_runtime_vs_input_types(results)

def plot_runtime_vs_input_types(results):
    """
    Plot runtime for different input types.

    Parameters:
        results (dict): Dictionary containing sizes and times for each input␣
 ↪type.
    """
    fig, axes = plt.subplots(1, 2, figsize=(16, 6))

    for input_name, data in results.items():
        sizes = np.array(data['sizes'])
        times = np.array(data['times'])

        # Linear plot
        axes[0].plot(sizes, times, marker='o', label=input_name)

        # Log-log plot with regression
```

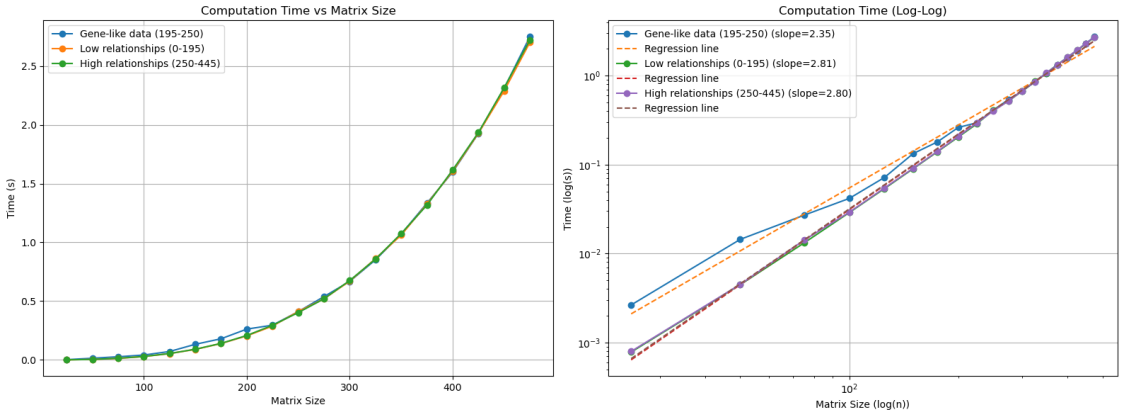
```
        log_sizes = np.log(sizes)
        log_times = np.log(times)
        slope, intercept, _, _, _ = linregress(log_sizes, log_times)
        axes[1].loglog(sizes, times, marker='o', label=f"{input_name}␣
↪(slope={slope:.2f})")
        axes[1].loglog(sizes, np.exp(intercept) * sizes ** slope,␣
↪linestyle='--', label="Regression line")

    # Linear plot
    axes[0].set_title("Runtime vs Matrix Size")
    axes[0].set_xlabel("Matrix Size")
    axes[0].set_ylabel("Time (s)")
    axes[0].legend()
    axes[0].grid(True)

    # Log-log plot
    axes[1].set_title("Runtime (Log-Log Scale)")
    axes[1].set_xlabel("Matrix Size (log(n))")
    axes[1].set_ylabel("Time (log(s))")
    axes[1].legend()
    axes[1].grid(True)

    plt.tight_layout()
    plt.show()

run_greedy_max_lcs_experiments_with_inputs()
```



4.2 Iterative Global Approach

[61]:
```
def run_iterative_experiments_with_plotting():
    """
    Run experiments to measure the computation time for constructing trees
    using the max_LCS_tree_iterative algorithm for varying input types and sizes.
    """
    input_types = [
        ("Gene-like data (195-250)", 195, 250),
        ("Low relationships (0-195)", 0, 195),
        ("High relationships (250-445)", 250, 445),
    ]
    sizes = [x for x in range(25, 500, 25)]  # Adjusted size range for manageability

    results = {name: {'sizes': [], 'times': []} for name, _, _ in input_types}

    for input_name, lower, upper in input_types:
        for n in sizes:
            # Generate random LCS matrix
            matrix = generate_matrix(n, lower, upper)

            # Measure the time for tree construction
            start_time = time.time()
            try:
                max_LCS_tree_iterative(matrix)
                elapsed_time = time.time() - start_time

                # Log the results
                results[input_name]['sizes'].append(n)
                results[input_name]['times'].append(elapsed_time)

            except Exception as e:
                print(f"Error with {input_name}, size {n}: {e}")

    plot_iterative_results(results)

def plot_iterative_results(results):
    """
    Plot the computation time for the max_LCS_tree_iterative algorithm across input types.

    Parameters:
        results (dict): Dictionary containing sizes and times for each input type.
    """
    fig, ax = plt.subplots(1, 2, figsize=(16, 6))
```

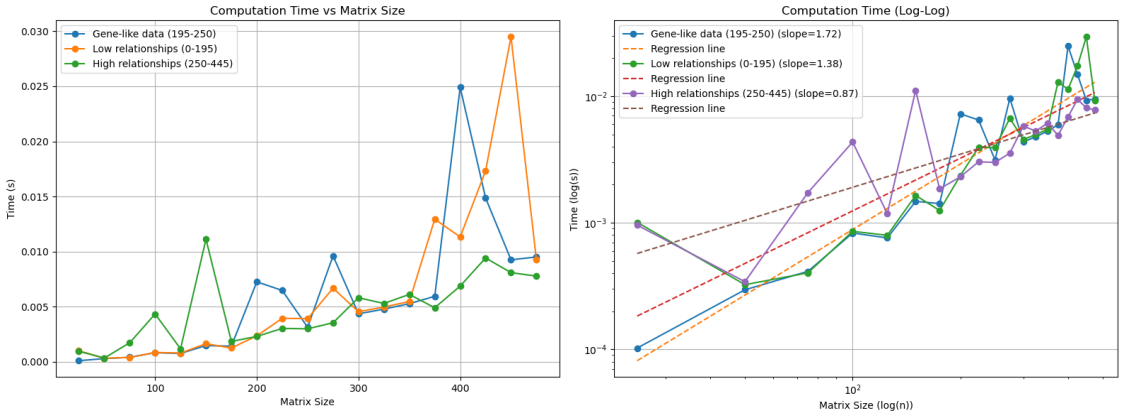
```
    for input_name, data in results.items():
        sizes = np.array(data['sizes'])
        times = np.array(data['times'])

        # Linear plot
        ax[0].plot(sizes, times, marker='o', label=input_name)

        # Log-log plot
        log_sizes = np.log(sizes)
        log_times = np.log(times)
        slope, intercept, _, _, _ = linregress(log_sizes, log_times)
        ax[1].loglog(sizes, times, marker='o', label=f"{input_name}␣
↪(slope={slope:.2f})")
        ax[1].loglog(sizes, np.exp(intercept) * sizes ** slope, linestyle='--',␣
↪label="Regression line")

    # Linear plot
    ax[0].set_title("Computation Time vs Matrix Size")
    ax[0].set_xlabel("Matrix Size")
    ax[0].set_ylabel("Time (s)")
    ax[0].legend()
    ax[0].grid(True)

    # Log-log plot
    ax[1].set_title("Computation Time (Log-Log)")
    ax[1].set_xlabel("Matrix Size (log(n))")
    ax[1].set_ylabel("Time (log(s))")
    ax[1].legend()
    ax[1].grid(True)

    plt.tight_layout()
    plt.show()

run_iterative_experiments_with_plotting()
```



4.3 Recursive Global Approach

[62]:
```python
def run_recursive_experiments_with_plotting():
    """
    Run experiments to measure the computation time for constructing trees
    using the max_LCS_tree_recursive algorithm for varying input types and sizes.
    """
    input_types = [
        ("Gene-like data (195-250)", 195, 250),
        ("Low relationships (0-195)", 0, 195),
        ("High relationships (250-445)", 250, 445),
    ]
    sizes = [x for x in range(25, 500, 25)]  # Adjusted size range for␣
↪manageability

    results = {name: {'sizes': [], 'times': []} for name, _, _ in input_types}

    for input_name, lower, upper in input_types:
        for n in sizes:
            # generate random LCS matrix
            matrix = generate_matrix(n, lower, upper)

            # measure the time
            start_time = time.time()
            memo = {}
            root = [[-1 for _ in range(n)] for _ in range(n)]
            try:
                max_LCS_tree_recursive(matrix, 0, n - 1, memo, root)
                elapsed_time = time.time() - start_time

                # results
                results[input_name]['sizes'].append(n)
                results[input_name]['times'].append(elapsed_time)

            except Exception as e:
                print(f"Error with {input_name}, size {n}: {e}")

    plot_recursive_results(results)

def plot_recursive_results(results):
    """
    Plot the computation time for the max_LCS_tree_recursive algorithm across␣
↪input types.
```

```
    Parameters:
        results (dict): Dictionary containing sizes and times for each input␣
↪type.
    """
    fig, ax = plt.subplots(1, 2, figsize=(16, 6))

    for input_name, data in results.items():
        sizes = np.array(data['sizes'])
        times = np.array(data['times'])

        # linear plot
        ax[0].plot(sizes, times, marker='o', label=input_name)

        # log-log plot of
        log_sizes = np.log(sizes)
        log_times = np.log(times)
        slope, intercept, _, _, _ = linregress(log_sizes, log_times)
        ax[1].loglog(sizes, times, marker='o', label=f"{input_name}␣
↪(slope={slope:.2f})")
        ax[1].loglog(sizes, np.exp(intercept) * sizes ** slope, linestyle='--',␣
↪label="Regression line")

    # Linear plot
    ax[0].set_title("Computation Time vs Matrix Size")
    ax[0].set_xlabel("Matrix Size")
    ax[0].set_ylabel("Time (s)")
    ax[0].legend()
    ax[0].grid(True)

    # Log-log plot
    ax[1].set_title("Computation Time (Log-Log)")
    ax[1].set_xlabel("Matrix Size (log(n))")
    ax[1].set_ylabel("Time (log(s))")
    ax[1].legend()
    ax[1].grid(True)

    plt.tight_layout()
    plt.show()

run_recursive_experiments_with_plotting()
```



5 Question 6: Probability Estimation

Is LCS a good metric for increasing number of gene sequences? NO, because we will just be computing larger and larger sums which could eventually lead to integer explosion and other bottlenecks. To solve this we change our optimization problem to a minimization, instead of maximization problem. That is when we introduce the levensthein distance.

5.1 Levensthein Distance

[63]:
```
genes = [
    (
        "a",
    ␣
↪"ATGGTGCGAAAGCATCTCTTTTCGTGGCGTGATAAGTTTTATGGTATCCCCGGACGTTGGCTACTACAATTCTCCGAAGTATAAGTGAGTAG
    ),
    (
        "b",
    ␣
↪"TGGTGCGAAAGCATCTCTTTTCCGTGGCGTATAGTTTTATGGTATCCCCGGAACGCTGGCTACTACAATCTCCGAAGTATAGAGTGAGTAGA
    ),
    (
        "c",
    ␣
↪"TCTGTGCGATATACATCTCTATCGTTGCGGTATGTTTTATGTGCATCACCCCACGCGCTGGCTACAGTACAATCTGCTGGAAGTACTAGGTG
    ),
    (
        "d",
    ␣
↪"ATGAGGCGCAAAATTCTCTTTCTCGTGGCGCTGATTAAGTTTTATGTATCCCCGGACGTTGGCTACTGACAATTGCTCCGAAGTATAAAGTA
    ),
    (
        "e",
```

```
        ␣
↪"TGGTGCGATATACATCTCTTTTCGTGCGTATGTTTTATGGTGATCACCCGGAACCGCTGGCTACATACAATCTCTGGAAGTACTAGGTGGTA
    ),
    (
        "f",
        ␣
↪"GGGGGAAAGCGATCCCTTATCGTGGCTGTGATAAGTTTTTATCGGGTATCCGCCGGACGTTGGCGTACTACAATTCTCCGAAGTTAAGTGAG
    ),
    (
        "g",
        ␣
↪"TGGTGCGATATACATCCTCTTTTCGTGCGTATGTTTTAGGTACACCGGATACGCCTGGCTTACAAGTACCAATCTCTGAGAAGTCACTGAGG
    ),
]
```

[64]:
```
def levenshtein_distance_with_operations(A, B):
    """
    Compute the Levenshtein distance between two strings and track the number of
    operations (insertions, deletions, substitutions, and matches).

    Parameters:
        A (str): The first string.
        B (str): The second string.

    Returns:
        tuple: A tuple containing:
            - total_operations (int): Total number of operations.
            - matches (int): Number of character matches.
            - insertions (int): Number of insert operations.
            - deletions (int): Number of delete operations.
            - substitutions (int): Number of substitution operations.
    """
    n = len(A)  # Length of the first string
    m = len(B)  # Length of the second string

    # Initialize a 2D matrix to store Levenshtein distances
    D = [[0] * (m + 1) for _ in range(n + 1)]

    # Fill the base cases: transforming an empty string to another string
    for i in range(1, n + 1):
        D[i][0] = i  # Cost of deleting characters
    for j in range(1, m + 1):
        D[0][j] = j  # Cost of inserting characters

    # Populate the distance matrix
    for i in range(1, n + 1):
        for j in range(1, m + 1):
```

```
            if A[i - 1] == B[j - 1]:  # Characters match, no cost
                cost = 0
            else:  # Characters do not match, substitution cost is 1
                cost = 1

            # Minimum cost of deletion, insertion, or substitution
            D[i][j] = min(
                D[i - 1][j] + 1,       # Deletion
                D[i][j - 1] + 1,       # Insertion
                D[i - 1][j - 1] + cost # Substitution
            )

    # Backtrack through the matrix to determine the operations performed
    i, j = n, m
    insertions = deletions = substitutions = matches = 0

    while i > 0 and j > 0:
        if A[i - 1] == B[j - 1]:  # Characters match
            matches += 1
            i -= 1
            j -= 1
        elif D[i][j] == D[i - 1][j] + 1:  # Deletion
            deletions += 1
            i -= 1
        elif D[i][j] == D[i][j - 1] + 1:  # Insertion
            insertions += 1
            j -= 1
        else:  # Substitution
            substitutions += 1
            i -= 1
            j -= 1

    # Handle remaining characters in A or B
    while i > 0:
        deletions += 1
        i -= 1
    while j > 0:
        insertions += 1
        j -= 1

    # Calculate the total number of operations
    total_operations = insertions + deletions + substitutions + matches
    return total_operations, matches, insertions, deletions, substitutions
```

[65]:
```
def normalize_transition_matrix(matrix):
    column_sums = matrix.sum(axis=0)  # Sum across columns
```

```
    normalized_matrix = matrix / column_sums[np.newaxis, :]  # Divide each␣
 ↪column by its sum
    return normalized_matrix

def build_transition_matrix(X, Y):
    n = len(X)
    m = len(Y)

    total_operations, matches, insertions, deletions, substitutions = (
        levenshtein_distance_with_operations(X, Y)
    )

    # Total operations include matches
    total_operations = matches + insertions + deletions + substitutions

    # Probabilities
    p_match = matches / total_operations
    p_substitute = substitutions / total_operations
    p_insert = insertions / total_operations
    p_delete = deletions / total_operations

    # Transition matrix
    transition_matrix = np.array([
        [p_match, p_substitute, p_insert],  # Row A
        [p_substitute, p_match, p_insert],  # Row B
        [p_delete, p_delete, 1 - 2 * p_delete]  # Row ∈
    ])

    # Normalize rows to ensure they sum to 1
    #normalized_matrix = normalize_transition_matrix(transition_matrix)
    return transition_matrix

# Example usage
example_matrix =␣
 ↪normalize_transition_matrix(build_transition_matrix(genes[0][1], genes[1][1]))
example_rounded_matrix = np.round(example_matrix, 3)
print(f'Transition Matrix from a to b:\n {example_rounded_matrix}')
```

```
Transition Matrix from a to b:
 [[0.887 0.019 0.041]
 [0.019 0.887 0.041]
 [0.094 0.094 0.918]]
```

[66]:
```
# Assuming 'genes' is a list of tuples, where each tuple contains a gene name␣
 ↪and its sequence
transition_matrices = []
```

```
# Compute pairwise transition matrices for all pairs of genes
for i in range(len(genes)):
    for j in range(i + 1, len(genes)):  # Compute only upper triangular matrix
        # Extract sequences from genes[i] and genes[j]
        markov =␣
↪normalize_transition_matrix(build_transition_matrix(genes[i][1], genes[j][1]))
        transition_matrices.append(markov)

# Compute the average transition matrix from all pairwise matrices
average_matrix = np.mean(transition_matrices, axis=0)

# Round the average matrix to 3 decimals
average_matrix_rounded = np.round(average_matrix, 3)

# Print the final averaged matrix
print("Average Transition Matrix (Rounded to 3 decimals):")
print(average_matrix_rounded)
```

```
Average Transition Matrix (Rounded to 3 decimals):
[[0.839 0.065 0.101]
 [0.065 0.839 0.101]
 [0.096 0.096 0.797]]
```

[ ]: