Decoding The Relationship Between Genes.

Project Overview:

In this project, I work with a set of seven mutated gene sequences and try to reconstruct their genealogy using two different algorithmic strategies: a local approach and a global approach. The main idea is to first figure out how similar the gene sequences are to each other, and then use those similarities to infer grandparent-parent-child relationships between them.

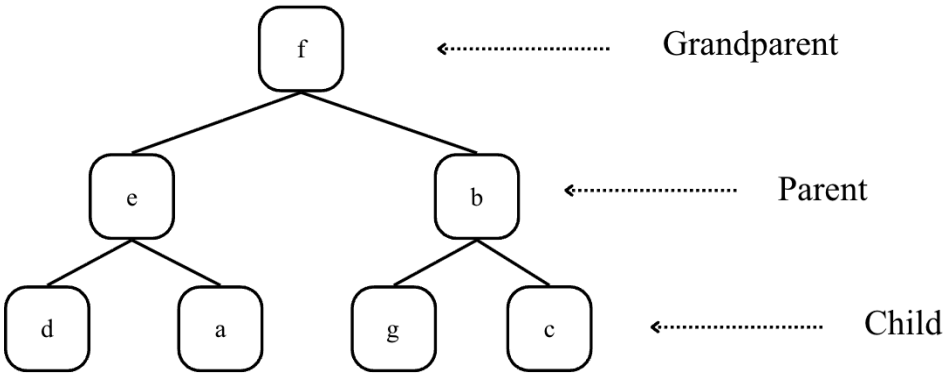

Figure 1: Inferred genealogy tree (using the global strategy)

To measure similarity, I compute the longest common subsequence (LCS) for every pair of genes and store these values in a 2D matrix. This gives me a concrete way to reason about how closely related any two sequences are.

	a	b	c	d	e	f	g
a	334	226	223	255	270	226	228
b	226	305	266	228	235	248	279
c	223	266	320	223	222	228	251
d	255	228	223	336	278	233	231
e	270	235	222	278	304	250	229
f	226	248	228	233	250	276	235
g	228	279	251	231	229	235	338

Figure 2: LCS similarity matrix (higher values indicate greater similarity).

I then convert these similarity values into distances, which I use as a proxy for how likely two genes are to be directly connected in the genealogy. Using these distances, I build two binary genealogy trees. The first uses a greedy, local strategy that makes decisions based only on immediate neighbors. The second uses a global, dynamic programming strategy that evaluates the quality of the entire tree at once. Comparing these two trees helps highlight the differences between local and global reasoning when inferring relationships.

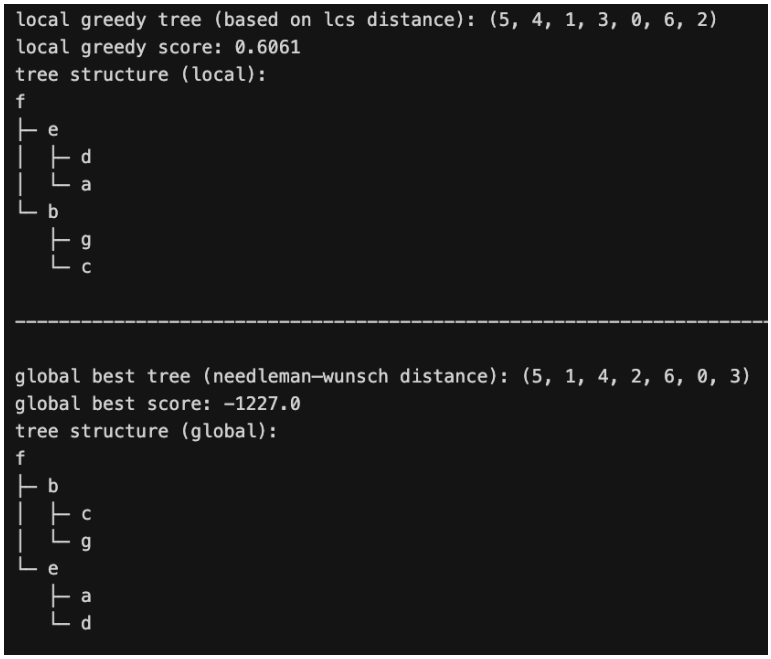

Figure 3: Inferred genealogy trees through implementing a local and global strategy.

Finally, I go beyond tree construction by analyzing algorithmic scaling and optimization, and by using the inferred genealogies to estimate mutation probabilities, including insertions, deletions, and substitutions. This ties the algorithmic side of the project back to biological intuition and evolutionary interpretation.

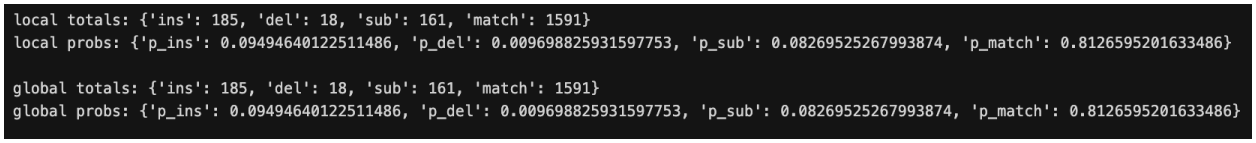

Figure 4: Number and estimated probabilities of insertions, deletions, and substitutions.

Methods Summary:

VIDEO 1- Local and Global Strategy Explanation

Local Strategy:

I construct a binary genealogy tree using a greedy approach based on pairwise sequence similarity. I begin by converting the LCS lengths into a normalized distance matrix, where each entry is defined as $1 - \frac{LCS\,(i,j)}{min(|i|, |j|)}$. The heuristic used is that smaller values indicate more similar gene sequences.

The tree is constructed through a sequence of local decisions. First, I select the grandparent node as the gene with the smallest average distance to all others, treating centrality in similarity space as a greedy heuristic for ancestral placement. Next, I choose the two parent nodes as the genes closest to the grandparent. Finally, the remaining genes are assigned as children by selecting the closest available nodes for each parent independently.

To quantify the quality of the inferred local tree, I compute a local greedy score defined as the sum of pairwise distances along all parent-child edges in the tree. Lower scores indicate tighter clustering of related genes and therefore a more consistent genealogy under the chosen distance metric.

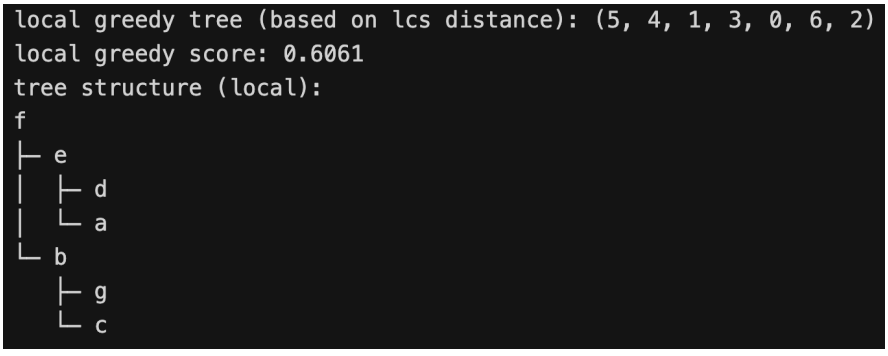

Figure 5: Inferred genealogy trees through implementing a local strategy.

This strategy is local because each decision is made using only immediate pairwise distances and is never revisited. The algorithm does not evaluate alternative tree configurations or optimize a global objective. It builds the tree incrementally by choosing the best local option at each step.

Global Strategy:

I construct a binary genealogy tree by explicitly optimizing a global objective that evaluates the quality of the entire tree at once. Instead of relying only on local similarity, this approach uses Needleman-Wunsch global alignment scores to capture the entire sequence similarity between every pair of genes. These scores are converted into a pairwise distance matrix, where smaller values indicate more similar sequences.

Rather than making incremental greedy choices, I evaluate all valid binary tree configurations consistent with the fixed grandparent-parent-child structure and compute a total tree score for each candidate. The score is defined as the sum of distances along all parent-child edges in the tree. The globally optimal tree is the one that minimizes this total edge score, thereby balancing similarity across the entire genealogy rather than favoring any single local decision.

This strategy is global because the placement of each node is determined by how it affects the overall tree score. Decisions are not made independently or at every step. Instead, every possible configuration is compared under the same global metric, and the tree with the best score is selected. This ensures that the final structure reflects the strongest overall consistency among all the gene relationships.

```
global best tree (needleman–wunsch distance): (5, 1, 4, 2, 6, 0, 3)
global best score: -1227.0
tree structure (global):
f
├─ b
│  ├─ c
│  └─ g
└─ e
   ├─ a
   └─ d
```

Figure 6: Inferred genealogy trees through implementing a global strategy.

Local vs Global Strategy Comparison:

As expected, the genealogy trees produced by the local and global strategies differ slightly. The local strategy builds the tree using a greedy, bottom-up process that relies only on immediate pairwise distances. Once a node is placed, the decision is never revisited, which makes the approach efficient but sensitive to early choices and unable to correct suboptimal placements.

In contrast, the global strategy evaluates the genealogy as a whole. Rather than committing to local decisions, it compares all valid tree configurations under a fixed structure and selects the one that minimizes a global objective based on Needleman-Wunsch alignment distances. This allows the algorithm to capture deeper shared structure between genes that may not be apparent from local similarity alone.

Although both strategies produce tree scores, these values cannot be compared directly, as they are calculated from different distance metrics and use different numerical scales. Instead, the comparison shows a "conceptual" trade-off: the local strategy prioritizes proximity and efficiency, while the global strategy prioritizes overall consistency across all the possible choices to create the genealogy tree.

Complexity Analysis:

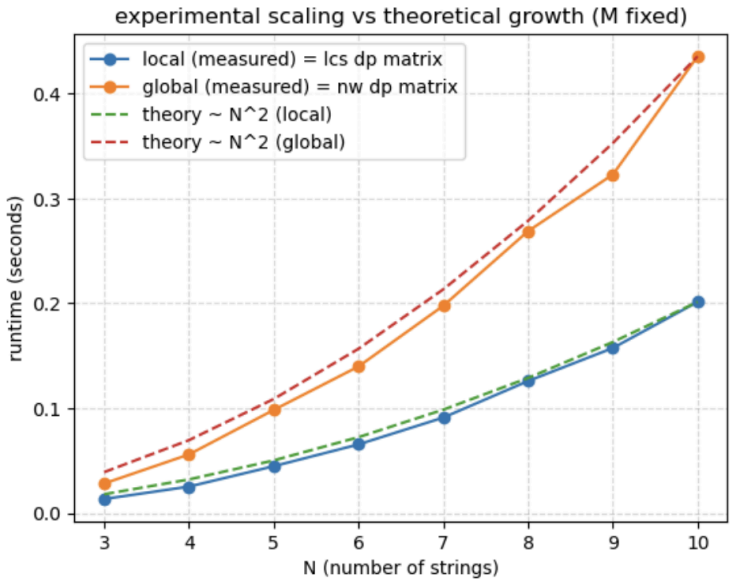

Figure 7: Experimental scaling vs. theoretical growth (M fixed).

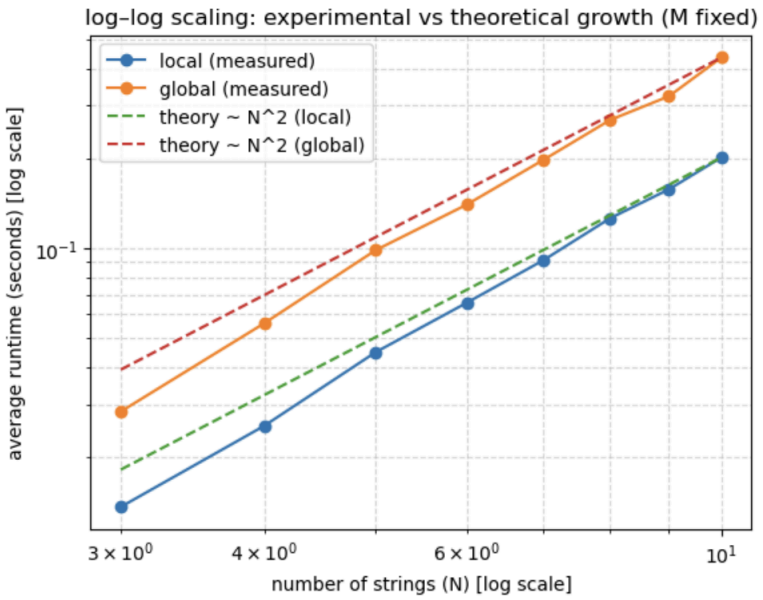

Figure 8: Log-log scaling of experimental vs. theoretical runtime (M fixed).

To study scaling, I fixed the gene length at $M = 200$ and varied the number of genes $N \in \{3, \cdots, 10\}$. For each N, I generated a random dataset with *make_random_set_strings(N, M, seed = N)*, extracted the sequences with *unpack_set_strings*, and then measured average runtime using *avg_runtime_seconds(..., repeats=3)*.

For the local strategy, I timed only *lcs_dist_matrix_from_lcs_length(seqs)* inside *run_greedy()*. This isolates the dominant cost: computing pairwise LCS dynamic programming across all pairs of genes. Each LCS computation is $O(M^2)$, and there are $O(N^2)$ pairs, so the dominant runtime is $\Theta(N^2 M^2)$.

For the global strategy, I timed only *nw_dist_matrix_from_score(seqs)* inside *run_global()*. Needleman-Wunsch global alignment is also a DP over prefixes with cost $O(M^2)$ per pair, again across $O(N^2)$ pairs, so the dominant runtime is also $\Theta(N^2 M^2)$. I intentionally did not time the later tree-search step, because my Q3 tree inference was built around a fixed 7-node structure; isolating the distance-matrix DP step keeps the scaling experiment valid when N changes.

Since M is fixed, both theoretical curves reduce to $\Theta(N^2)$ growth, which is why I plotted *theory_greedy* = N^2 and *theory_global* = N^2 (scaled only for visual comparison). In both the regular and log-log plots, the measured runtimes follow the expected N^2 trend: as the number of genes increases, the runtime grows roughly quadratically. This makes sense because the dominant cost in both strategies is computing dynamic programming alignments for all pairs of sequences. The global (Needleman-Wunsch) curve consistently sits above the local (LCS) curve because each alignment is more expensive to compute, even though both methods scale the same way asymptotically.

Mutation Probability Estimation:

Video 2 - Explanation on how to estimate the probabilities of insertions, deletions, and mutations given the genealogy.

Definitions:

Mutations are defined using aligned parent-child sequences. An insertion occurs when the child has a base and the parent has a gap. A deletion occurs when the parent has a base and the child has a gap. A substitution occurs when both have bases but they differ. A match occurs when both have the same base. Each aligned column corresponds to exactly one of these events.

Algorithmic Approach:

To estimate mutation probabilities, I treat each parent-child edge in the inferred genealogy as one evolutionary step. For every edge, I compute a global Needleman-Wunsch alignment and use the traceback to identify insertions, deletions, substitutions, and matches along that edge. These counts are counted across all edges in the tree to compensate for the small dataset size. Because the dataset is limited, I apply Laplace smoothing to avoid zero probabilities. Alignment ambiguity is handled using deterministic tie-breaking in the traceback, ensuring consistent and reproducible estimates.

Results and Interpretation:

The estimated probabilities show that matches dominate, while insertions and substitutions occur at moderate rates and deletions are pretty rare in comparison. This makes sense from a biological perspective: most bases are similar because genes are related, while substitutions and insertions introduce “manageable” variation. Deletions are especially “bad” because removing a base can cause a frameshift, disrupting all downstream codons and often leading to nonfunctional proteins. My hypothesis is that, from an evolutionary perspective, deletions are more strongly selected against because it creates a deleterious mutation, which can explain their lower estimated probability.

Learning for Growth-Supporting Materials:

Video 3 - LO and HC application.

Moment from class: last breakout from last session
LO application: #AlgoStratDataStruct
HC: #networks

AI Statement:

I used AI primarily as a support tool while working on this assignment. It helped me think through the overall structure of the project and debug parts of my code when I got stuck on implementation challenges. The core ideas, algorithmic choices, and biological reasoning were developed by me, but AI helped me move past specific points of confusion more efficiently by suggesting ways to organize functions, fix errors, or clarify logic. I treated AI as a way to speed up iteration and deepen my understanding, rather than as a replacement for my own problem-solving or reasoning.

References:

2.5: The Needleman-Wunsch Algorithm. (2020, October 5). Biology LibreTexts. https://bio.libretexts.org/Bookshelves/Computational_Biology/Book%3A_Computational_Biology_-_Genomes_Networks_and_Evolution_(Kellis_et_al.)/02%3A_Sequence_Alignment_and_Dynamic_Programming/2.05%3A_The_Needleman-Wunsch_Algorithm

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms, third edition*. MIT Press.

CS110 Session 25 - [14.2] Synthesis of algorithms and computational applications. (2017). *Forum*. Minerva.edu. https://forum.minerva.edu/app/courses/3717/sections/12784/classes/98193

Interactive demo for Needleman–Wunsch algorithm. (n.d.). Bioboot.github.io. https://bioboot.github.io/bimm143_W20/class-material/nw/

Moreanu, A. (2025a). *VIDEO 1*. Loom. https://www.loom.com/share/52436fdb527441778123f5f3994ac6f1

Moreanu, A. (2025b). *Video 2*. Loom. https://www.loom.com/share/9771c7fa796d45159f990d32a9425a25

Moreanu, A. (2025c). *Video 3*. Loom. https://www.loom.com/share/299e5c0ddac54dda9d8120d4aae0e707

Needleman-Wunsch Explained Clearly. (2022, February 7). YouTube. https://www.youtube.com/watch?v=FIxYGV7WPA8

Wikipedia Contributors. (2021, March 24). *Needleman–Wunsch algorithm*. Wikipedia; Wikimedia Foundation. https://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch_algorithm

Code Appendix:

```
Outline:

  1.  Problem setup + data (set_strings)
  2.  Q1 LCS function (all LCSs) + tests
  3.  Q2 LCS-length matrix + interpretation
```

```
4. Q3 Local vs Global strategy (design)
5. Q4 Implement + draw trees + compare
6. Q5 Complexity (theory + experimental plots)
7. Q6 Mutation probability estimation (idea + code + sanity check)
```

```
Q1 LCS function (all LCSs) + tests
```

```python
def lcs_tables(x, y):
    """
    Construct the DP tables for the LCS problem, following the algorithm LCS-LENGTH
from Cormen et al.

    Function compares two strings x and y and computes:
    1) A table of LCS lengths for all prefixes of the strings.
    2) A table of directional choices that records how each LCS length was obtained.

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

    Returns
    -------
    c : list of lists of int
        The LCS length table, where c[i][j] is the length of a longest common
        subsequence of the prefixes x[:i] and y[:j].
    b : list of lists of str
        The direction table used for reconstruction. Each entry is one of:
          - "NW" : diagonal move (characters matched),
          - "N"  : move up (skip a character from x),
          - "W"  : move left (skip a character from y).
    """
    # lenghts of strings
    m = len(x)
    n = len(y)

    # create c and b tables
    c = [[0 for _ in range(n + 1)] for _ in range(m + 1)]
    b = [["" for _ in range(n + 1)] for _ in range(m + 1)]
```

```
    # fill in tables
    for i in range(1, m + 1):
        for j in range(1, n + 1):

            if x[i - 1] == y[j - 1]:
                c[i][j] = c[i - 1][j - 1] + 1
                b[i][j] = "NW" # diagonal

            else:
                if c[i - 1][j] >= c[i][j - 1]:
                    c[i][j] = c[i - 1][j]
                    b[i][j] = "N" # up
                else:
                    c[i][j] = c[i][j - 1]
                    b[i][j] = "W" #left

    return c, b

def reconstruct_lcs(x, y, c, i, j, memoization):
    """
    Recursively reconstruct all LCSs of x[:i] and y[:j] using the CLRS LCS length
table.

    This function follows the same logic used in Cormen et al. to reconstruct an LCS,
but
    extends it to return all the possible LCSs.

    Memoization ensures that each subproblem (i, j) is solved only once, avoiding
redundant
    recursion when multiple optimal paths exist

    Parameters
    ----------
    x : str
        input of first string
    y : str
        input of second string
    c : list of lists of int
        LCS length table produced by teh lcs_tables)
    i : int
        the current index into x
```

```
    j : int
        the current index into y
    memoization : dict
        Memoization dictionary mapping (i, j) to a set of LCS strings

    Returns
    -------
    set of str: all longest common subsequences of x[:i] and y[:j].
    """
    # if we have already solved this subproblem, we return the stored result
    if (i, j) in memoization:
        return memoization[(i, j)]

    # base case: one of the prefixes is empty
    if i == 0 or j == 0:
        memoization[(i, j)] = {""}
        return memoization[(i, j)]

    # case 1: last characters match
    if x[i - 1] == y[j - 1]:

        previous_lcs = reconstruct_lcs(x,y, c, i - 1, j - 1, memoization)

        result = set()
        for seq in previous_lcs:
            result.add(seq + x[i - 1])

        memoization[(i, j)] = result
        return result

    # case 2: last characters do not match
    results = set()

    # move up if it preserves the optimal LCS length
    if c[i - 1][j] >= c[i][j - 1]:
        upper_results = reconstruct_lcs(x, y, c, i - 1, j, memoization)
        results.update(upper_results)

    # move left if it preserves the optimal LCS length
    if c[i][j - 1] >= c[i - 1][j]:
        left_results = reconstruct_lcs(x, y, c, i, j - 1, memoization)
        results.update(left_results)
```

```
    memoization[(i, j)] = results
    return results


def find_lcs(x, y):
    """
    This function finds all LCSs between two strings.
    It follows the dynamic programming approach described in Cormen et al. by first
    computing the LCS length table and then backtracking through this table to recover
    the subsequences.
    Cormen et al. describes how to recover a single LCS, but this function extends the
    method to return all possible LCSs of maximum length.

    Parameters
    ----------
    x : str
        First input string
    y : str
        Second input string

    Returns
    -------
    (list[str] | None, int)
        A tuple containing:
        - a sorted list of all longest common subsequences,
        - the length of the LCS.

        If there is no common subsequence, returns (None, 0).
    """
    # handle edge cases where one string is empty
    if not x or not y:
        return None, 0

    # building the LCS tables using the CLRS algorithm
    c, _ = lcs_tables(x, y) #we only need the length table c, the direction table b is
not required here

    # the LCS length is stored in the bottom-right cell of the table
    lcs_length = c[len(x)][len(y)]

    if lcs_length == 0:
```

```
        return None, 0

    # reconstruct all LCSs using the memoization
    memoization = {}
    lcs_set = reconstruct_lcs(x, y, c, len(x), len(y), memoization)

    # remove empty string if present
    lcs_set.discard("")

    return sorted(lcs_set), lcs_length
```

```
# Test 1: CLRS example
x1 = "ABCBDAB"
y1 = "BDCABA"
assert find_lcs(x1, y1) == (['BCAB', 'BCBA', 'BDAB'], 4)

# Test 2: one empty string
# tests the base case
x2 = "abc"
y2 = ""
assert find_lcs(x2, y2) == (None, 0)

# Test 3: one single character match
# edge case with small input --> see if DP table is reconctructed correctly
x3 = "abc"
y3 = "a"
assert find_lcs(x3, y3) == (['a'], 1)

# Test 4: case sensitivity check
# edge case to check if character nature matters
x4 = "ABC"
y4 = "abc"
assert find_lcs(x4, y4) == (None, 0)

# Test 5: identical strings
# best case scenario for LCS --> the full string is returned; no duplicates introduced
x5 = "genetics"
y5 = "genetics"
assert find_lcs(x5, y5) == (['genetics'], len("genetics"))

# Test 6: no common characters
# edge case where no charecters match
```

```
x6 = "xyz"
y6 = "abc"
assert find_lcs(x6, y6) == (None, 0)

# Test 7: Mixed-case characters (partial match)
# checking if the algorithm handels symbol matching correctly
x7 = "AbC"
y7 = "abc"
assert find_lcs(x7, y7) == (['b'], 1)

print("All LCS tests passed!")
```

```
Q2 LCS-length matrix + interpretation
```

```
set_strings = [
   ('a',

"TGGTGGGAACATTGATCTGCTTTTACGGTGACAGTCTAGTTTTTGGTACCCCTGCGGAACGATTGGGCCATCTACAGTGCCCGCG
CCACAGTTTAAAGTAGTGACGTGGATCTGATATTAACAGAGGACGTTGTTGGACGGAAACTTATCACCCAGCTAACAATCATATGA
TGAACCCGACGTTACGGTGATGAGTGACCCATTACTGTCAACACGCATGAAGATCGCGCCCGTAGGCCACGCTCTTAGTAATGACC
GTTGCTCCACATATGTTTCCGACATCTGTGTCGTCTTTGGAAGAACCTATATCGTAGCAGGAGGGATGTGTACTCGG"),

   ('b',

"TGGTACGAAGGCATCTCTTTTCCCGTGGCGTGATGGTGTTATGGGTATCCACCCGAACGGTGGCGTCTACAACTCTCCCAAGATA
TACGAGCGAGTTAGAGCTTAATTAAACAGAGGGCGTCGTTGACGTCATTAGGGCACCAGCATGAATCAACCGCATGAACCGTTATC
GTGGGTTTCAGGACTCTATGTAAAAGGATGAAGATTTTCGCAACAGCTACTAAATAATGGAGTGTGTACAATAATGCAACCCTACA
CCGTGCAATCTTGTCACTGCTAGTAGAGCCTGGGGATTGGTTCCTCGG"),

   ('c',

"TGGTCGAAGGCATCTATTTTGCCGTGCGCGTGCTGGTGTAATTGGCAATCCACCCGAACGGTCTCGTCTACCACTCTCCCAAGAG
TATACGAGCGCAGTCAGACCTCTACATAAACATAGGGCGTCCTTTACGTCATTAGGCCACCAGCGATGAATCAACCTCATCGAACC
CTTATGGTGGGTTTACGCAGCTCTCTACTAAAAGCACTCGTGAGGTTTTCGTCAACAGCTACTAAATAATTCAAGGTGTGCAAATT
GCATACCCAACACCGTGCAATCTTGTCCTGATAGTAGAGCATGGGGGAGTCTAGGGACCTCGG"),

   ('d',

"GGGTGCGACCAAGATCTATTTCTCCGTGACGGATAAGGTCTCTATTTGTCCTTCCTGGGAATCGATTGGAAACCTACATTCAGCG
```

```
AGCAGTATAAGTAGGGACGTAGATCATTATATTATATCTAGAGGGCGTGGTTTGGGACGCAAGTTTCCAGCGGAACTAGGAATCGT
CCGATGAACTCCTACGATGAGAGAGTCCGTGAACTATAGGGGCAGAAGGATTGATATGCGCCGAGGACCACGATCATTAGTAATGA
GCGTTGCGTCACATATGTATCCCGACCATTCGGTCGTCTTGGAACTATCTTGATCGTACCCACGGAGGATGTTTCTCGG"),

   ('e',

"GGGTGCGACAAGATCTCTTTTCCGTGACGTATAGTTTTATGGTACTCCCCGGAACGATTGGCAACTACAATCCCGCGAACAGTAT
AAGTAGTGACGTAGATCTTATATTAACAGAGGGCGTTGTTGGACGCAAGTATCACCAACTAGAATCATCCGATGAACTCGACGTGA
GGTGTCAGTGAACTATAGGGCAAAGGATGAATATGCGCCGAGGCCACGCTCTAGTAATGAGCGTTGCTCACATATGTATCCGACAT
CGGTCGTCTTGGAACTACCTAGATCGTAGCAGGAGGATGTTACTCGG"),

   ('f',

"TGGTGCGAAAGCATCTCTTTTCCGTGGCGTATAGTTTTATGGTATCCCCGGAACGCTGGCTACTACAATCTCCGAAGTATAGAGT
GAGTAGATTTAATTAACAGAGGGCGTCGTTGACGCATTAGCACCAACTGAATCAACCGATAACTTAACGTGGGTTTCAGTGACTAT
AGGGCAAAGGATGAACATTTTCGAGCAGCTCTAATAATGAGCGTGACAATATGAATCCACACCGTCATCTTGAACTCCTAGATTGA
GCAAGAGGTTGTTCCTCGG"),

   ('g',

"TGGGACGTAGGCACCTCTTTTCCCGTGGCGTGATGGGTGTGACGGGTATCCACCCGTACGGGCACCTCTTAACAACGCTCTCACT
AAGATCATACGAGCGAGTATAGAGCTTAATTACCAGAGGGACCTCGCTGCCGACATTAGAGGCAACCAGCATGAACTCAAGCCCCA
ATGTAACGTCATCGTGGGGATTGCAGGACTCTATATAAAAGGATGAAAGGAATTTTCCGCAACCAGTTACTAAAGTAAATGGAGTG
TGTACAAATAATGTGAAGCCCTCACACCGTGCCTCTTTGTCACTGCGTAGTAAGGAGCACTAGGTGTATTGGTTCGCTCGG")
]
```

```
def lcs_length(x, y):
   """
   Computing the length of the LCS between two strings.

   This function follows the same dynamic programming recurrence as the
   LCS-LENGTH algorithm described in Cormen et al., but only computes
   the final LCS length (not the subsequence itself).

   In order to reduce memory usage, we store only two rows of the DP table at a time:
the previous row and the current row.
   This is possible because each DP c[i][j] depends only on c[i-1][j], c[i][j-1], and
c[i-1][j-1].

   Parameters
   ----------
```

```
    x : str
    y : str

    Returns
    -------
    int
        Length of the longest common subsequence between x and y.
    """

    # making sure that y is the shorter string to minimize memory usage
    if len(y) > len(x):
        x, y = y, x

    m, n = len(x), len(y)

    # prev[j] represents c[i-1][j]
    # curr[j] represents c[i][j]
    prev = [0] * (n + 1)
    curr = [0] * (n + 1)

    # Build the DP table row by row (CLRS recurrence)
    for i in range(1, m + 1):
        xi = x[i - 1]
        curr[0] = 0  # base case: LCS with empty prefix is 0

        for j in range(1, n + 1):
            if xi == y[j - 1]:
                # If characters match, then extend the LCS
                curr[j] = prev[j - 1] + 1
            else:
                # If characters do not match, then take the best of skipping one
character
                curr[j] = max(prev[j], curr[j - 1])

        # move current row to prev for next iteration
        prev, curr = curr, prev

    # Final LCS length is in the last cell of the last completed row
    return prev[n]
```

```
import numpy as np
```

```
"""
Constructing a pairwise LCS length matrix for all strings in set_strings.

Each entry len_lcs_matrix[i, j] stores the length of the LCS betweenthe i-th and j-th
strings.
This matrix summarizes pairwise similarity between all gene sequences.

"""

n = len(set_strings)

# initialize a 7x7 matrix of integers
len_lcs_matrix = np.zeros((n, n), dtype=int)

for i in range(n):
   seq_i = set_strings[i][1]

   len_lcs_matrix[i, i] = len(seq_i) # the LCS of a string with itself is the full
length of the string


   for j in range(i + 1, n):
       seq_j = set_strings[j][1] #compute only the upper triangle and mirror (matrix
is symmetric)

       # Compute LCS length using DP (CLRS)
       length = lcs_length(seq_i, seq_j)

       len_lcs_matrix[i, j] = length
       len_lcs_matrix[j, i] = length  # symmetry

len_lcs_matrix
```

```
"""
Non-trivial pairs correspond to comparisons between different strings (i != j).
We are extracting only the upper-triangular entries (i < j) to avoid duplicates.
"""

upper_triangle_values = len_lcs_matrix[np.triu_indices(n, k=1)]
```

```
unique_lcs_lengths = np.unique(upper_triangle_values)

print("Unique LCS lengths (non-trivial pairs):", unique_lcs_lengths)
print("Number of unique LCS lengths:", len(unique_lcs_lengths))
```

```
"""
Printing the matrix in a readable format to manually examine it.
"""
labels = [label for label, _ in set_strings]

# Print column headers
print("     " + "  ".join(f"{lab:>4}" for lab in labels))

# Print each row with its corresponding label
for i in range(n):
    row_values = "  ".join(f"{len_lcs_matrix[i, j]:>4}" for j in range(n))
    print(f"{labels[i]:>4}  {row_values}")
```

```
Q3 Local vs Global strategy
```

```
def unpack_set_strings(set_strings):
    """
    Return labels and sequences as two parallel lists.

    I use this as a small helper so the rest of my code is easier to read.
    It also helps me avoid repeating tuple indexing everywhere.

    Parameters
    ----------
    set_strings : list of tuples
        A list where each element is of the form (label, sequence).

    Returns
    -------
    labels : list of str
        The list of labels (e.g. ['a', 'b', 'c', ...]).
    sequences : list of str
        The corresponding list of DNA sequences.
```

```
    """
    labels = []
    sequences = []

    for label, seq in set_strings:
        labels.append(label)
        sequences.append(seq)

    return labels, sequences


def tree_from_indices(labels, root, left_child, right_child):
    """
    Build a tiny nested-dict tree from explicit parent/child indices.

    This is a convenience function so I can print trees in a consistent way.
    """
    return {
        "name": labels[root],
        "left": {"name": labels[left_child], "left": None, "right": None},
        "right": {"name": labels[right_child], "left": None, "right": None},
    }


def print_tree(tree, indent=0):
    """
    Print a nested-dict binary tree in a readable way.
    """
    if tree is None:
        return

    print("  " * indent + tree["name"])
    print_tree(tree["left"], indent + 1)
    print_tree(tree["right"], indent + 1)
```

```
Local Strategy:

Metric: LCS distance from Q2 matrix
```

```
def lcs_distance_matrix_from_lengths(len_lcs_matrix, sequences):
    """
    Convert the LCS length matrix into a normalized distance matrix in [0, 1].

    Each distance is computed as:
        1 - (LCS length / min(sequence lengths))

    This is suitable for a local strategy becuase it is a simple pairwise metric based
on the information we already computed in Q2.
    Heuristic = smaller distance means more similar

    Parameters
    ----------
    len_lcs_matrix : np.ndarray
        A square matrix where entry (i, j) is the LCS length between sequences i and j.
    sequences : list of str
        The DNA sequences corresponding to the matrix indices.

    Returns
    -------
    np.ndarray
        A square distance matrix with values in the range [0, 1].
    """
    n = len(sequences)
    dist = np.zeros((n, n), dtype=float)

    for i in range(n):
        for j in range(n):
            denom = min(len(sequences[i]), len(sequences[j]))
            dist[i, j] = 1.0 - (len_lcs_matrix[i, j] / denom)

    return dist
```

```
Greedy tree building (closest neighbors)
```

```
def argmin_average_distance(dist):
    """
    Find the index with the smallest average distance to all other nodes.
```

```
   This function is used as a greedy heuristic to select a root-like node
(grandparent) that is centrally located with respect to all others.

   Parameters
   ----------
   dist : np.ndarray
       A square distance matrix.

   Returns
   -------
   int
       The index of the node with the smallest average distance.
   """
   n = dist.shape[0]
   best_i = 0
   best_avg = float("inf")

   for i in range(n):
       avg = (np.sum(dist[i]) - dist[i, i]) / (n - 1)
       if avg < best_avg:
           best_avg = avg
           best_i = i

   return best_i


def two_closest_nodes(dist, anchor, candidates):
   """
   Identify the two candidate nodes closest to a given anchor node (smallest distance)
=> these will be parents (most "similar" to the grandparent).

   This function supports greedy local decisions by comparing only immediate neighbors
based on pairwise distance.

   Parameters
   ----------
   dist : np.ndarray
       A square distance matrix.
   anchor : int
       Index of the reference node.
   candidates : list of int
       Indices of candidate nodes to compare against the anchor.
```

```
    Returns
    -------
    tuple of int
        The indices of the two closest candidate nodes.

    """
    # store (distance, node) pairs and pick the smallest two
    pairs = []
    for node in candidates:
        pairs.append((dist[anchor, node], node))

    pairs.sort(key=lambda t: t[0])
    return pairs[0][1], pairs[1][1]


def local_greedy_tree_from_distance(dist, labels):
    """
    Construct a genealogy tree (grandparent -> 2 parents -> 4 children) using a local greedy strategy.

    Logic:
    1) Select a central node with minimal average distance as the grandparent.
    2) Choose the two closest nodes to act as parents.
    3) Assign remaining nodes to parents based on closest local distance.

    This strategy is local because each decision considers only immediate neighbor comparisons, not the global quality of the entire tree.

    Parameters
    ----------
    dist : np.ndarray
        A square distance matrix between all sequences.
    labels : list of str
        Labels corresponding to each node index.

    Returns
    -------
    tuple of int
        A 7-tuple representing the tree structure:
        (grandparent, parent1, parent2, child11, child12, child21, child22)
    """
```

```
    n = dist.shape[0]
    nodes = list(range(n))

    # choose grandparent (most central by average distance)
    g = argmin_average_distance(dist)

    # choose two closest nodes to g as the parents
    remaining = [x for x in nodes if x != g]
    p1, p2 = two_closest_nodes(dist, g, remaining)

    # remove parents from the pool
    remaining = [x for x in remaining if x not in (p1, p2)]

    # assign children to each parent using a greedy local rule
    # i keep this simple: each parent takes its two closest remaining nodes
    c11, c12 = two_closest_nodes(dist, p1, remaining)
    remaining = [x for x in remaining if x not in (c11, c12)]

    c21, c22 = two_closest_nodes(dist, p2, remaining)

    return (g, p1, p2, c11, c12, c21, c22)
```

```
Tree Score (to help compare local vs global strategy)
```

```
def tree_edge_score(dist, tree_tuple):
    """
    Compute the total edge score for a fixed 7-node genealogy tree.

    The score is the sum of distances along all parent-child edges.
    Lower scores indicate a tighter clustering of related nodes.

    Parameters
    ----------
    dist : np.ndarray
        A square distance matrix.
    tree_tuple : tuple of int
        A tuple encoding the tree structure as node indices.

    Returns
```

```
    -------
    float
        The total edge distance score for the tree.
    """
    g, p1, p2, c11, c12, c21, c22 = tree_tuple

    score = 0.0
    score += dist[g, p1]
    score += dist[g, p2]
    score += dist[p1, c11]
    score += dist[p1, c12]
    score += dist[p2, c21]
    score += dist[p2, c22]

    return score
```

```
Global Strategy:

Metric: Needleman-Wunsch (DP)
```

```
def needleman_wunsch_score(x, y, match=1, mismatch=-1, gap=-1):
    """
    Compute the Needleman-Wunsch global alignment score between two sequences.

    This function uses dynamic programming, following the same prefix-based recurrence
style as CLRS. Each DP cell represents the best alignment score for two prefixes of
the sequences.

    Unlike LCS, this method enforces end-to-end alignment and therefore captures global
similarity between sequences.

    Parameters
    ----------
    x : str
        First DNA sequence.
    y : str
        Second DNA sequence.
    match : int
        Score for matching characters.
```

```
    mismatch : int
        Penalty for mismatched characters.
    gap : int
        Penalty for inserting a gap.

    Returns
    -------
    int
        The optimal global alignment score.
    """
    m, n = len(x), len(y)
    dp = [[0 for _ in range(n + 1)] for _ in range(m + 1)]

    # base cases: aligning against empty prefix forces gaps
    for i in range(1, m + 1):
        dp[i][0] = dp[i - 1][0] + gap
    for j in range(1, n + 1):
        dp[0][j] = dp[0][j - 1] + gap

    for i in range(1, m + 1):
        xi = x[i - 1]
        for j in range(1, n + 1):
            yj = y[j - 1]

            diag = dp[i - 1][j - 1] + (match if xi == yj else mismatch)
            up = dp[i - 1][j] + gap
            left = dp[i][j - 1] + gap

            dp[i][j] = max(diag, up, left)

    return dp[m][n]
```

```
Build a NW distance matrix + choose the best tree globally (search)
```

```
def nw_distance_matrix(sequences, match=1, mismatch=-1, gap=-1):
    """
    Build a pairwise distance matrix using Needleman-Wunsch alignment scores.
```

```
   Higher alignment scores indicate greater similarity, so distances are computed as
the negative of the alignment score.

   Parameters
   ----------
   sequences : list of str
       DNA sequences to compare.
   match : int
       Score for matching characters.
   mismatch : int
       Penalty for mismatched characters.
   gap : int
       Penalty for gaps.

   Returns
   -------
   np.ndarray
       A symmetric distance matrix based on global alignment.
   """
   n = len(sequences)
   dist = np.zeros((n, n), dtype=float)

   for i in range(n):
       for j in range(i, n):
           if i == j:
               dist[i, j] = 0.0
           else:
               s = needleman_wunsch_score(sequences[i], sequences[j],
                                          match=match, mismatch=mismatch, gap=gap)
               dist[i, j] = -float(s)
               dist[j, i] = dist[i, j]

   return dist


def best_tree_by_global_score(dist):
   """
  Identify the best genealogy tree by minimizing a global distance objective.

   This function evaluates all valid tree configurations under a fixed
grandparent-parent-child structure and selects the one with the lowest total edge
distance.
```

```
Because the entire tree is evaluated at once => global strategy.

Parameters
----------
dist : np.ndarray
    A square distance matrix.

Returns
-------
best_score : float
    The minimum total edge distance found.
best_tree : tuple of int
    The tree structure that achieves this minimum score.
"""
n = dist.shape[0]
nodes = list(range(n))

best_score = float("inf")
best_tree = None

# choose grandparent
for g in nodes:
    others = [x for x in nodes if x != g]

    # choose two parents (unordered pair)
    for i in range(len(others)):
        for j in range(i + 1, len(others)):
            p1 = others[i]
            p2 = others[j]

            kids = [x for x in others if x not in (p1, p2)]

            # choose 2 kids for p1, remaining 2 go to p2
            for a in range(len(kids)):
                for b in range(a + 1, len(kids)):
                    c11 = kids[a]
                    c12 = kids[b]

                    remaining = [x for x in kids if x not in (c11, c12)]
                    c21 = remaining[0]
                    c22 = remaining[1]
```

```
                        candidate = (g, p1, p2, c11, c12, c21, c22)
                        score = tree_edge_score(dist, candidate)

                        if score < best_score:
                            best_score = score
                            best_tree = candidate

    return best_score, best_tree
```

```
Q4 Implement + draw trees + compare
```

```
labels, sequences = unpack_set_strings(set_strings)

# local strategy uses your lcs matrix from q2
lcs_dist = lcs_distance_matrix_from_lengths(len_lcs_matrix, sequences)
local_tree_tuple = local_greedy_tree_from_distance(lcs_dist, labels)
local_score = tree_edge_score(lcs_dist, local_tree_tuple)

print("local greedy tree (based on lcs distance):", local_tree_tuple)
print("local greedy score:", round(local_score, 4))

g, p1, p2, c11, c12, c21, c22 = local_tree_tuple
print("tree structure (local):")
print(labels[g])
print("├─", labels[p1])
print("│  ├─", labels[c11])
print("│  └─", labels[c12])
print("└─", labels[p2])
print("   ├─", labels[c21])
print("   └─", labels[c22])

print("\n" + "-" * 80 + "\n")

# global strategy uses needleman-wunsch distance
nw_dist = nw_distance_matrix(sequences, match=1, mismatch=-1, gap=-1)
best_score, best_tree_tuple = best_tree_by_global_score(nw_dist)

print("global best tree (needleman-wunsch distance):", best_tree_tuple)
```

```
print("global best score:", round(best_score, 4))

g, p1, p2, c11, c12, c21, c22 = best_tree_tuple
print("tree structure (global):")
print(labels[g])
print("├─", labels[p1])
print("│  ├─", labels[c11])
print("│  └─", labels[c12])
print("└─", labels[p2])
print("   ├─", labels[c21])
print("   └─", labels[c22])
```

```
Q5 Complexity (theory + experimental plots)
```

```
import random

def random_dna_string(m):
    """
    generate a random dna string of length m.
    """
    alphabet = ["A", "C", "G", "T"]
    return "".join(random.choice(alphabet) for _ in range(m))


def make_random_set_strings(n, m, seed=0):
    """
    create a list like set_strings but with random sequences.

    Parameters
    ----------
    n : int
        number of genes (strings)
    m : int
        length of each gene
    seed : int
        seed so results are reproducible

    Returns
    -------
```

```
    list[tuple[str, str]]
        [('a', 'ACGT...'), ('b', 'TGCA...'), ...]
    """
    random.seed(seed)
    labels = [chr(ord("a") + i) for i in range(n)]
    return [(labels[i], random_dna_string(m)) for i in range(n)]
```

```
def lcs_dist_matrix_from_lcs_length(sequences):
    """
    build a normalized lcs distance matrix using the lcs_length function we already
have.

    distance(i, j) = 1 - lcs_length(i, j) / min(len(i), len(j))

    Returns
    -------
    np.ndarray
        n x n symmetric distance matrix in [0, 1]
    """
    n = len(sequences)
    dist = np.zeros((n, n), dtype=float)

    for i in range(n):
        dist[i, i] = 0.0
        for j in range(i + 1, n):
            L = lcs_length(sequences[i], sequences[j])
            denom = min(len(sequences[i]), len(sequences[j]))
            d = 1.0 - (L / denom)
            dist[i, j] = d
            dist[j, i] = d

    return dist
```

```
def nw_dist_matrix_from_score(sequences, match=1, mismatch=-1, gap=-1):
    """
    build a distance matrix using needleman-wunsch scores.

    i convert score -> distance by using dist = -score.
    this keeps ordering consistent: smaller distance means more similar.
```

```
    Returns
    -------
    np.ndarray
        n x n symmetric distance matrix
    """
    n = len(sequences)
    dist = np.zeros((n, n), dtype=float)

    for i in range(n):
        dist[i, i] = 0.0
        for j in range(i + 1, n):
            s = needleman_wunsch_score(sequences[i], sequences[j],
                                       match=match, mismatch=mismatch, gap=gap)
            d = -float(s)
            dist[i, j] = d
            dist[j, i] = d

    return dist
```

```
import time
import matplotlib.pyplot as plt

def avg_runtime_seconds(run_fn, repeats=3):
    """
    Time a function a few times and return the average runtime.

    Parameters
    ----------
    run_fn : callable
        a function that takes no arguments and runs the computation we want to time
    repeats : int
        how many times to repeat the timing (we average to reduce noise)

    Returns
    -------
    float
        average runtime in seconds
    """
    times = []
    for _ in range(repeats):
```

```
        t0 = time.perf_counter()
        run_fn()
        t1 = time.perf_counter()
        times.append(t1 - t0)
    return sum(times) / len(times)


# choose M fixed and vary N
M = 200
N_values = [3, 4, 5, 6, 7, 8, 9, 10]

greedy_times = []
global_times = []

for N in N_values:
    test_set = make_random_set_strings(N, M, seed=N)
    _, seqs = unpack_set_strings(test_set)

    def run_greedy():
        """
        local strategy timing.

        for q5 i only time the dominant cost of the local method:
        building the pairwise lcs distance matrix (dp over all pairs).
        the actual greedy tree building step is lower-order and also
        assumes a fixed 7-node structure in our q3 implementation,
        so timing just the dp part is the clean scaling experiment.
        """
        _ = lcs_dist_matrix_from_lcs_length(seqs)

    def run_global():
        """
        global strategy timing.

        for q5 i only time the dominant cost of the global method:
        building the pairwise needleman-wunsch distance matrix (dp over all pairs).
        the tree search/inference step is a separate layer on top of the matrix and
        in our project code it was designed specifically for 7 nodes, so this keeps
        the scaling experiment valid when n changes.
        """
        _ = nw_dist_matrix_from_score(seqs)
```

```
    greedy_times.append(avg_runtime_seconds(run_greedy, repeats=3))
    global_times.append(avg_runtime_seconds(run_global, repeats=3))


# ---- theoretical curves (scaled to match the plot shape) ----
# since M is fixed, the dp part scales like:
#   pairwise comparisons = O(N^2)
# and each comparison costs O(M^2), which is constant here
Ns = np.array(N_values, dtype=float)

theory_greedy = Ns**2
theory_global = Ns**2  # same dominant dp scaling when M is fixed

# scale curves so they sit near the measured curves (for shape comparison only)
theory_greedy = theory_greedy / theory_greedy.max() * max(greedy_times)
theory_global = theory_global / theory_global.max() * max(global_times)


# ---- plot ----
plt.figure()
plt.plot(N_values, greedy_times, marker="o")
plt.plot(N_values, global_times, marker="o")
plt.plot(N_values, theory_greedy, linestyle="--")
plt.plot(N_values, theory_global, linestyle="--")

plt.xlabel("N (number of strings)")
plt.ylabel("runtime (seconds)")
plt.title("experimental scaling vs theoretical growth (M fixed)")
plt.legend([
    "local (measured) = lcs dp matrix",
    "global (measured) = nw dp matrix",
    "theory ~ N^2 (local)",
    "theory ~ N^2 (global)"
])
plt.grid(True, linestyle="--", alpha=0.5)
plt.show()
```

```
# ---- log-log plot (both axes on a log scale) ----

plt.figure()
```

```
# experimental measurements
plt.loglog(N_values, greedy_times, marker="o")
plt.loglog(N_values, global_times, marker="o")

# theoretical curves
plt.loglog(N_values, theory_greedy, linestyle="--")
plt.loglog(N_values, theory_global, linestyle="--")

plt.xlabel("number of strings (N) [log scale]")
plt.ylabel("average runtime (seconds) [log scale]")
plt.title("log-log scaling: experimental vs theoretical growth (M fixed)")

plt.legend([
    "local (measured)",
    "global (measured)",
    "theory ~ N^2 (local)",
    "theory ~ N^2 (global)"
])

plt.grid(True, which="both", linestyle="--", alpha=0.5)
plt.show()
```

```
Q6 Mutation probability estimation
```

```
def nw_align(parent, child, match=1, mismatch=-1, gap=-1):
    """
    Global alignment (needleman-wunsch) with traceback.

    I need the alignment, not just the score, because i want to count: insertions,
deletions, and substitutions along a parent -> child edge.

    parameters
    ----------
    parent : str
        the "parent" gene sequence
    child : str
        the "child" gene sequence
    match, mismatch, gap : int
        scoring settings for the dp recurrence
```

```
    returns
    -------
    (str, str)
        aligned_parent, aligned_child using '-' for gaps
    """
    m = len(parent)
    n = len(child)

    # dp table of best scores for prefixes
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    # traceback table: "D" diagonal, "U" up, "L" left
    move = [[None] * (n + 1) for _ in range(m + 1)]

    # initialize borders (only gaps possible)
    for i in range(1, m + 1):
        dp[i][0] = dp[i - 1][0] + gap
        move[i][0] = "U"
    for j in range(1, n + 1):
        dp[0][j] = dp[0][j - 1] + gap
        move[0][j] = "L"

    # fill dp table (same idea as clrs lcs-length, just different recurrence)
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            diag = dp[i - 1][j - 1] + (match if parent[i - 1] == child[j - 1] else
mismatch)
            up = dp[i - 1][j] + gap
            left = dp[i][j - 1] + gap

            best = max(diag, up, left)
            dp[i][j] = best

            # deterministic tie-break so results are reproducible
            if best == diag:
                move[i][j] = "D"
            elif best == up:
                move[i][j] = "U"
            else:
                move[i][j] = "L"

    # traceback
```

```
    aligned_p = []
    aligned_c = []
    i, j = m, n

    while i > 0 or j > 0:
        step = move[i][j]
        if step == "D":
            aligned_p.append(parent[i - 1])
            aligned_c.append(child[j - 1])
            i -= 1
            j -= 1
        elif step == "U":
            aligned_p.append(parent[i - 1])
            aligned_c.append("-")
            i -= 1
        else:  # "L"
            aligned_p.append("-")
            aligned_c.append(child[j - 1])
            j -= 1

    aligned_p.reverse()
    aligned_c.reverse()
    return "".join(aligned_p), "".join(aligned_c)
```

```
def count_events(aligned_parent, aligned_child):
    """
    Count insertions/deletions/substitutions/matches from an alignment.

    Definitions i use for q6:
    - insertion: parent has '-' and child has a base
    - deletion: parent has a base and child has '-'
    - substitution: both have bases but they differ
    - match: both have bases and they are the same

    parameters
    ----------
    aligned_parent : str
    aligned_child : str

    returns
    -------
```

```
    dict
        counts for 'ins', 'del', 'sub', 'match'
    """
    counts = {"ins": 0, "del": 0, "sub": 0, "match": 0}

    for a, b in zip(aligned_parent, aligned_child):
        if a == "-" and b != "-":
            counts["ins"] += 1
        elif a != "-" and b == "-":
            counts["del"] += 1
        elif a != "-" and b != "-":
            if a == b:
                counts["match"] += 1
            else:
                counts["sub"] += 1

    return counts
```

```
def estimate_probs_from_tree(tree_tuple, sequences, match=1, mismatch=-1, gap=-1,
alpha=1):
    """
    Estimating probabilities of insertion/deletion/mutation using the genealogy.

    I treat each edge as one evolutionary step (parent -> child).
    For each edge I:
      1) compute a global alignment using needleman-wunsch (dp + traceback)
      2) count events from the alignment
    then I sum counts over all edges and convert to probabilities.

    parameters
    ----------
    tree_tuple : tuple
        (g, p1, p2, c11, c12, c21, c22) as indices into sequences
    sequences : list[str]
        sequences in the same index order as the tree
    match, mismatch, gap : int
        nw scoring settings
    alpha : float
        laplace smoothing (helps because dataset is tiny)

    returns
```

```
    -------
    (dict, dict)
        (total_counts, probabilities)
    """
    g, p1, p2, c11, c12, c21, c22 = tree_tuple
    edges = [(g, p1), (g, p2), (p1, c11), (p1, c12), (p2, c21), (p2, c22)]

    total = {"ins": 0, "del": 0, "sub": 0, "match": 0}

    for parent_i, child_i in edges:
        parent_seq = sequences[parent_i]
        child_seq = sequences[child_i]

        aligned_p, aligned_c = nw_align(parent_seq, child_seq, match=match,
mismatch=mismatch, gap=gap)
        counts = count_events(aligned_p, aligned_c)

        # add into totals
        for k in total:
            total[k] += counts[k]

    # convert counts to probabilities with laplace smoothing
    denom = (total["ins"] + total["del"] + total["sub"] + total["match"]) + 4 * alpha

    probs = {
        "p_ins": (total["ins"] + alpha) / denom,
        "p_del": (total["del"] + alpha) / denom,
        "p_sub": (total["sub"] + alpha) / denom,
        "p_match": (total["match"] + alpha) / denom,
    }

    return total, probs
```

```
# we already have these from q4
# labels, sequences = unpack_set_strings(set_strings)

local_counts, local_probs = estimate_probs_from_tree(local_tree_tuple, sequences)
global_counts, global_probs = estimate_probs_from_tree(best_tree_tuple, sequences)

print("local totals:", local_counts)
print("local probs:", local_probs)
```

```
print("\nglobal totals:", global_counts)
print("global probs:", global_probs)
```