Understanding minimalTree.py
This script constructs a **minimum spanning tree** from a list of sequences.
- What is the core principle behind the tree structure?
  ☐ Mutation rules
  ☐ Similarity distances
  ☐ Hypothesis testing
- (Recap) What kind of distances are used?
  The script uses… distance to measure how similar two sequences are.
- How is the root of the tree selected?
  The root is chosen based on…

Rule-based Validity in reconstructTree_a.py
In this script, only **valid mutations** are allowed.
- What defines a "valid mutation"?
  ☐ Arbitrary character change
  ☐ Only forward letter changes (e.g., a → b)
  ☐ Levenshtein below a threshold
- What happens when a connection violates mutation rules?
  It is…
- How is the resulting structure different from minimalTree.py?
  Now the tree…

Comparison & Reflection

Which tree structure might be more useful in:

- **Biological contexts?**
- **Textual/cultural transmission?**
- Explain the difference between reversible and irreversible mutations (Leitfehler).
- Which is true?
  ☐ minimalTree.py is taking Leitfehler into account
  ☐ reconstructTree_a.py is taking Leitfehler into account**.**

- What purpose do **hypothetical nodes** serve in the reconstruction process?
- Why are they sometimes introduced between nodes?
  ☐ To make the tree more balanced.
  ☐ To represent unknown or missing evolutionary stages.
  ☐ To remove "dead" nodes from the tree.
  ☐ To increase mutation distance between sequences.

How good is the reconstruction in reconstructTree_a.py?

This script expects a file mutation_tree.json in folder output of your project. It should be there as a result of running tree1.py to tree7a.py. Run reconstructTree.py.

- Compare with mutation_tree.json.
    - Count the number of hypothetical nodes created in the final tree.
    - Identify one "alive" node and its path back to the root.
    - Note any hypothetical nodes along that path.
    - Modify the JSON file and set any "dead" node to "alive": true.
        - Did new hypothetical nodes appear?
        - Are the paths of previously alive nodes affected?
- Compare with mutation_tree_loaded.svg.
    - What do you find?
      ☐ The forward letter changes (e.g., a → b) are respected.
      ☐ Surviving documents are put in a plausible position
      ☐ Reconstructed documents can be identified as „dead" nodes in the original tree.
    - Why are there differences in the original tree and the reconstructed one?
      Differences occure because …
    - Run any other file of tree1.py to tree7a.py to check the results

The Repair Process (reconstructTree_a.py)

This script fixes an invalid tree using local operations.

- What does the script do when it finds an invalid edge?
  ☐ Ignores it
  ☐ Swaps parent and child
  ☐ Rebuilds the entire tree
- What is a "bypass" operation?
  It removes…
- When does the script stop running?
  ☐ After a fixed number of steps
  ☐ When the tree is valid and stable
  ☐ When all leaves are hypothetical

When an overflow node is detected, the code always creates a brand-new hypothetical node (combining two children at random or by minimum distance) and reattaches them under the hypothetical.

Strategic Improvements (reconstructTree_b.py)

- What is the main improvement in version *b*?
  ☐ Simpler code
  ☐ Avoiding unnecessary hypothetical nodes
  ☐ More visual output
- How does the algorithm decide whether to insert a hypothetical node or not?
  It checks if…
- Why is it important to minimize hypothetical nodes?
  ☐ For computational efficiency
  ☐ To increase biological plausibility
  ☐ To keep reconstructions interpretable
  ☐ All of the above

When an overflow node is detected, the code first checks whether one child's sequence could validly mutate directly into the other child's sequence. • If yes, it reattaches one child under the other (no hypothetical needed). • Otherwise, it falls back to creating a new hypothetical (as in version *a*)

Evolutionary Strategy in reconstructTree.py

Both reconstructTree_a.py and reconstructTree_b.py follow a "single-tree iterative repair" pattern.

- How does this single-tree strategy differs from the population-based strategy in reconstructTree.py.

  reconstructTree_a.py and reconstructTree_b.py start with … while
- Briefly state one advantage and one disadvantage of using a population-based ("evolutionary") approach compared to a single-tree iterative approach.
- Test out different parameters for population_size, generations, remove_fraction.

Components of reconstructTree.py

These are very large files. So let's look closer …

- •What function or code snippet in reconstructTree.py is responsible for creating the initial population of random trees?
- •Approximately how many trees are in that initial population by default? (Hint: look for a variable like POP_SIZE or a loop that generates a fixed number of trees.)
- •Which function implements the fitness (i.e., "score") of a candidate tree in reconstructTree.py?
- •List two factors that this fitness function penalizes or rewards. For example, does it penalize invalid mutation edges? Does it use Levenshtein distance?
  Sketch out the high-level steps of the evolutionary loop inside reconstructTree.py. For each generation, what are the main stages (e.g., evaluate, select, mutate)?

Fill out the table:

Feature / Behavior	reconstructTree.py	reconstructTree_a.py	reconstructTree_b.py
Uses a population of multiple candidate trees	☐	☐	☐
Develops trees over many generations with fitness, selection and mutation	☐	☐	☐
Creates exactly one random tree and repairs it in a loop until no more changes are required	☐	☐	☐
Always inserts a new hypothetical node on overflow	☐	☐	☐
Tries direct hanging (child → child) first before inserting a hypothetical node	☐	☐	☐
Removes redundant hypothetical nodes after each repair step	☐	☐	☐
Calculates Levenshtein distances to evaluate mutation edges	☐	☐	☐
Returns during the "Generation … Best Fitness:…" from	☐	☐	☐
Terminated when there are no more invalid edges/overflows/hypotheticals	☐	☐	☐