TITLE :DNA Sequence Similarity Checker

Objectives:

- **Computes the LCS using Dynamic Programming and reconstructs the sequence through backtracking.**
- **Validates DNA inputs and builds a styled DP table with highlighted LCS path.**
- **Generates a concise summary including LCS length, sequence, and similarity percentage.**

Introduction:

- **The project focuses on comparing two DNA sequences to identify their similarity.**
- **LCS helps determine the longest subsequence common to both strings while preserving order.**
- **Useful in bioinformatics, genome comparison, sequence alignment, and error detection.**

Algorithms/Technique used:

1. Longest Common Subsequence (Dynamic Programming)

Algorithm Steps:

1. **Create a DP table of size *(m+1)×(n+1)(m+1) \times (n+1)*(m+1)×(n+1).**
2. **Initialize first row/column to 0 (base case: empty string).**
3. **For each cell:**
    a. **If characters match → dp[i][j] = dp[i-1][j-1] + 1**

    b. **Else → dp[i][j] = max(dp[i-1][j], dp[i][j-1])**

4. **Final LCS length is in dp[m][n].**

2. Backtracking Algorithm

1. **Start from bottom-right cell.**
2. **If characters match → move diagonally and add to LCS.**
3. **If mismatch → move to cell with greater value (up or left).**
4. **Reverse collected characters to get LCS.**

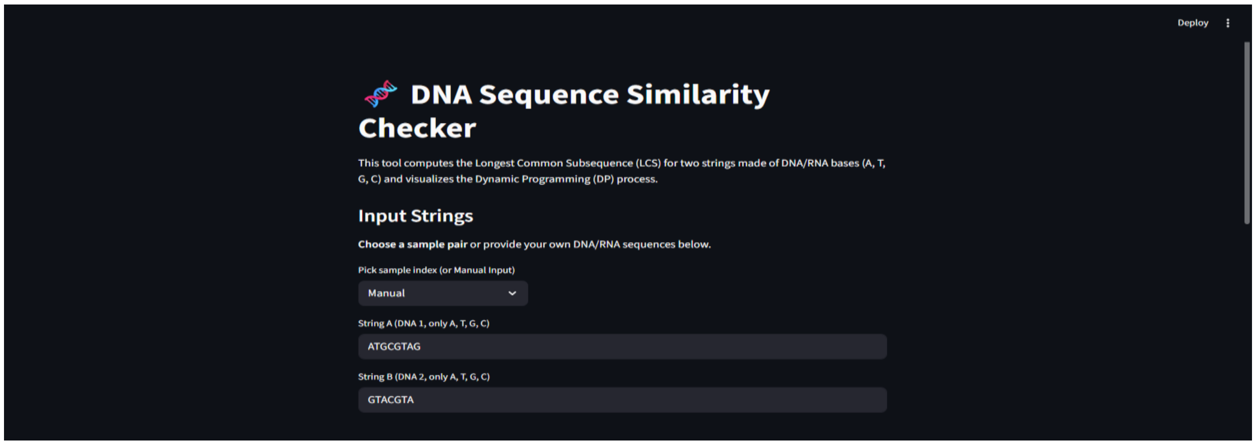
Time Complexity

- **DP Table Filling: O(m × n)**
- **Backtracking: O(m + n)**
- **Total Complexity: O(mn)**

Explanation

- **Every cell is computed once → m×n operations.**
- **Backtracking travels at most m+n steps.**
- **Thus, the algorithm is efficient even for moderately long DNA strings.**

Result:


Deploy

🧬 DNA Sequence Similarity Checker

This tool computes the Longest Common Subsequence (LCS) for two strings made of DNA/RNA bases (A, T, G, C) and visualizes the Dynamic Programming (DP) process.

Input Strings

Choose a sample pair or provide your own DNA/RNA sequences below.

Pick sample index (or Manual Input)

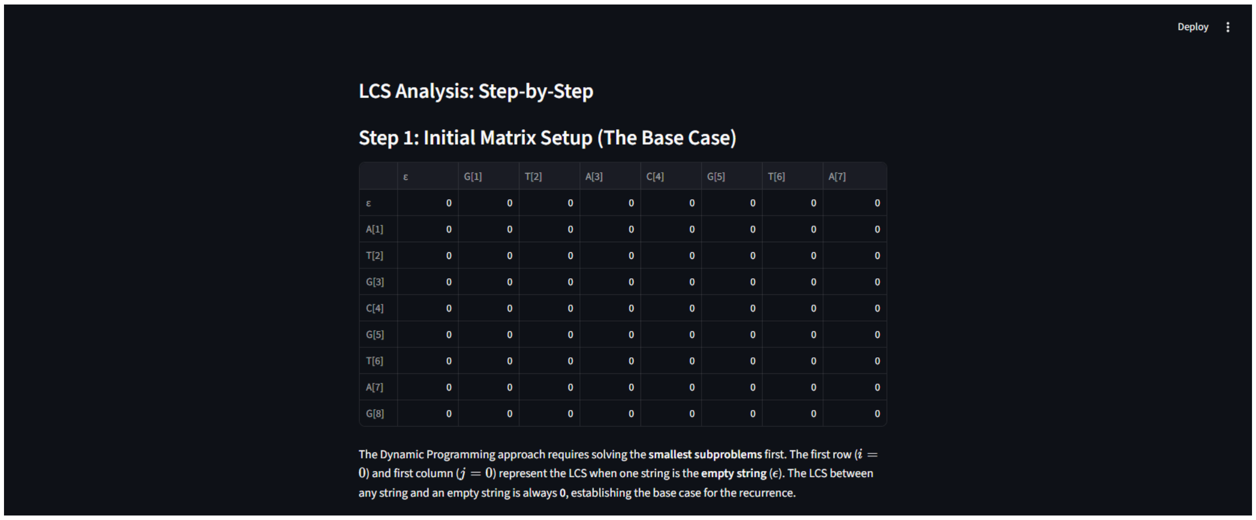
Manual

String A (DNA 1, only A, T, G, C)

ATGCGTAG

String B (DNA 2, only A, T, G, C)

GTACGTA

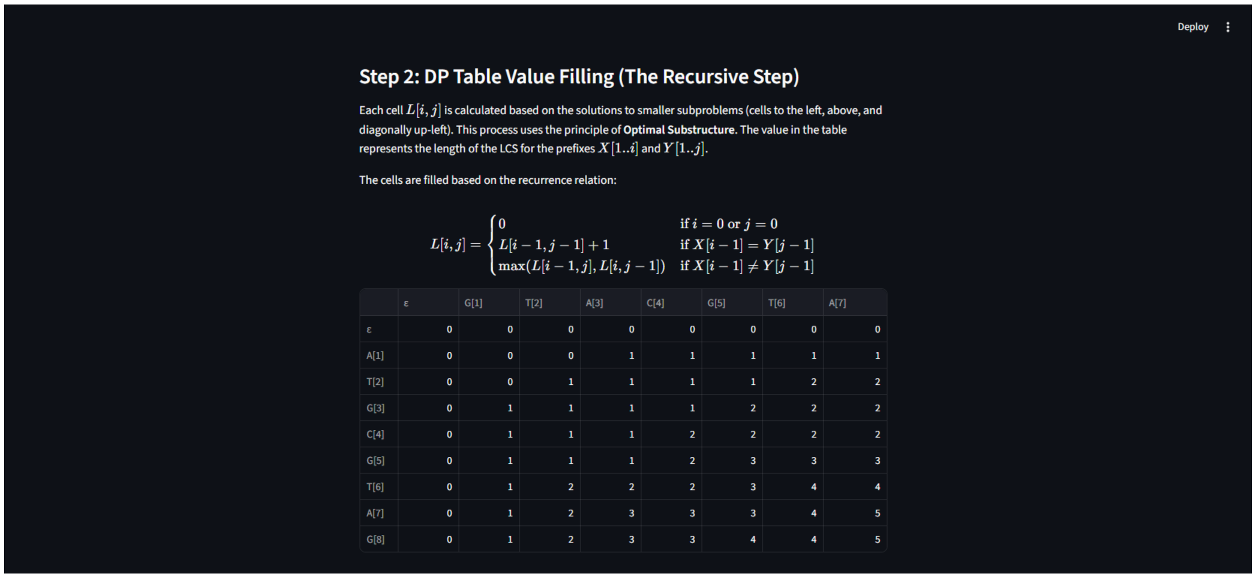

Deploy

LCS Analysis: Step-by-Step

Step 1: Initial Matrix Setup (The Base Case)

	ε	G[1]	T[2]	A[3]	C[4]	G[5]	T[6]	A[7]
ε	0	0	0	0	0	0	0	0
A[1]	0	0	0	0	0	0	0	0
T[2]	0	0	0	0	0	0	0	0
G[3]	0	0	0	0	0	0	0	0
C[4]	0	0	0	0	0	0	0	0
G[5]	0	0	0	0	0	0	0	0
T[6]	0	0	0	0	0	0	0	0
A[7]	0	0	0	0	0	0	0	0
G[8]	0	0	0	0	0	0	0	0

The Dynamic Programming approach requires solving the **smallest subproblems** first. The first row ($i = 0$) and first column ($j = 0$) represent the LCS when one string is the **empty string** (ϵ). The LCS between any string and an empty string is always **0**, establishing the base case for the recurrence.


Deploy

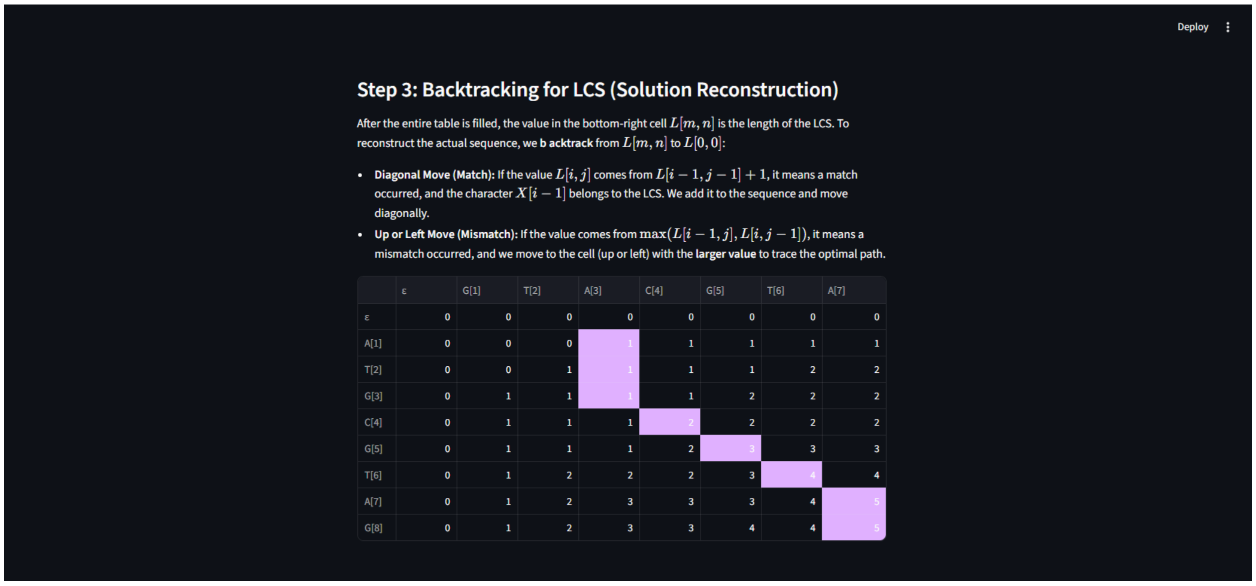
Step 2: DP Table Value Filling (The Recursive Step)

Each cell $L[i, j]$ is calculated based on the solutions to smaller subproblems (cells to the left, above, and diagonally up-left). This process uses the principle of **Optimal Substructure**. The value in the table represents the length of the LCS for the prefixes $X[1..i]$ and $Y[1..j]$.

The cells are filled based on the recurrence relation:

$$L[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L[i-1, j-1] + 1 & \text{if } X[i-1] = Y[j-1] \\ \max(L[i-1, j], L[i, j-1]) & \text{if } X[i-1] \neq Y[j-1] \end{cases}$$

	ε	G[1]	T[2]	A[3]	C[4]	G[5]	T[6]	A[7]
ε	0	0	0	0	0	0	0	0
A[1]	0	0	0	1	1	1	1	1
T[2]	0	0	1	1	1	1	2	2
G[3]	0	1	1	1	1	2	2	2
C[4]	0	1	1	1	2	2	2	2
G[5]	0	1	1	1	2	3	3	3
T[6]	0	1	2	2	2	3	4	4
A[7]	0	1	2	3	3	3	4	5
G[8]	0	1	2	3	3	4	4	5

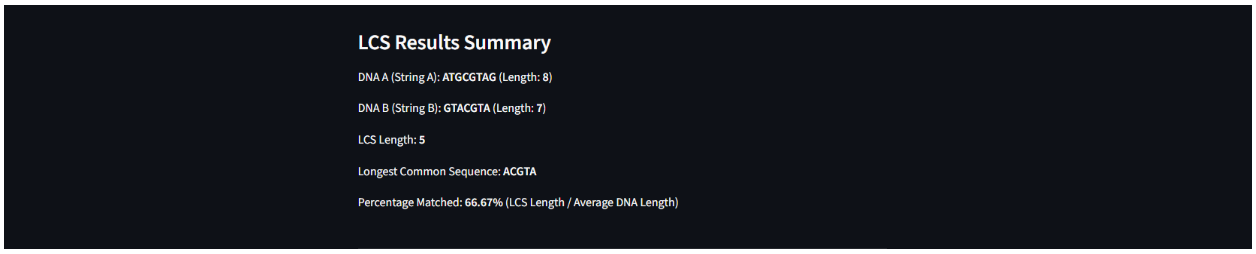


Step 3: Backtracking for LCS (Solution Reconstruction)

After the entire table is filled, the value in the bottom-right cell $L[m, n]$ is the length of the LCS. To reconstruct the actual sequence, we **b acktrack** from $L[m, n]$ to $L[0, 0]$:

- **Diagonal Move (Match):** If the value $L[i, j]$ comes from $L[i-1, j-1]+1$, it means a match occurred, and the character $X[i-1]$ belongs to the LCS. We add it to the sequence and move diagonally.
- **Up or Left Move (Mismatch):** If the value comes from $\max(L[i-1, j], L[i, j-1])$, it means a mismatch occurred, and we move to the cell (up or left) with the **larger value** to trace the optimal path.

	ε	G[1]	T[2]	A[3]	C[4]	G[5]	T[6]	A[7]
ε	0	0	0	0	0	0	0	0
A[1]	0	0	0	1	1	1	1	1
T[2]	0	0	1	1	1	1	2	2
G[3]	0	1	1	1	1	2	2	2
C[4]	0	1	1	1	2	2	2	2
G[5]	0	1	1	1	2	3	3	3
T[6]	0	1	2	2	2	3	4	4
A[7]	0	1	2	3	3	3	4	5
G[8]	0	1	2	3	3	4	4	5



LCS Results Summary

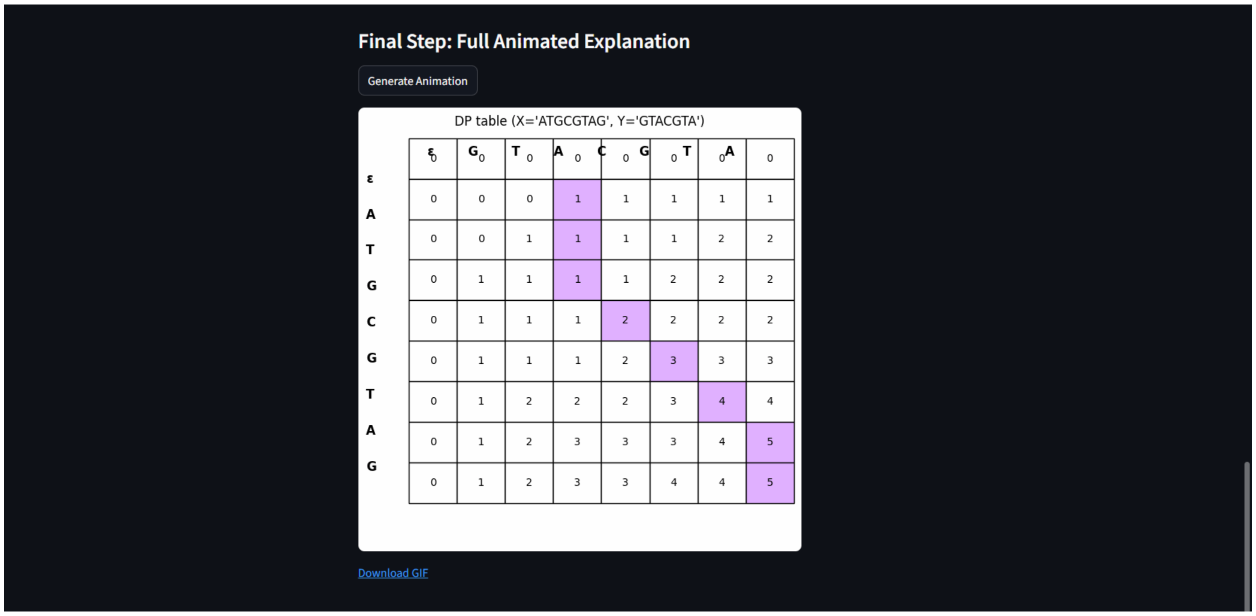
DNA A (String A): **ATGCGTAG** (Length: **8**)

DNA B (String B): **GTACGTA** (Length: **7**)

LCS Length: **5**

Longest Common Sequence: **ACGTA**

Percentage Matched: **66.67%** (LCS Length / Average DNA Length)



Final Step: Full Animated Explanation

DP table (X='ATGCGTAG', Y='GTACGTA')

	ε	G	T	A	C	G	T	A
	0	0	0	0	0	0	0	0
ε	0	0	0	1	1	1	1	1
A	0	0	1	1	1	1	2	2
T	0	1	1	1	1	2	2	2
G	0	1	1	1	2	2	2	2
C	0	1	1	1	2	3	3	3
G	0	1	2	2	2	3	4	4
T	0	1	2	3	3	3	4	5
A	0	1	2	3	3	4	4	5
G								

- **The tool accurately identifies shared subsequences between DNA strings.**
- **Helps visualize how DP builds optimal solutions gradually.**
- **Highlights importance of backtracking in extracting the final sequence.**

Conclusion

- **LCS effectively measures DNA similarity.**
- **Dynamic Programming ensures optimal and efficient computation.**
- **Visualization enhances understanding of internal DP operations.**

Future Scope

- **Extend LCS to full genome alignment (Needleman-Wunsch / Smith-Waterman).**
- **Add support for RNA bases and ambiguous nucleotide symbols.**
- **Integrate with machine-learning models for mutation prediction.**
- **Improve visualization with real-time interactive heatmaps.**