**Longest Common Subsequence Problem**

*Find a longest common subsequence of two strings.*

**Input:** Strings *s* and *t*.

**Output:** A longest common subsequence of *s* and *t*.

A string is a *subsequence* of a string *s* if it can be obtained by deleting some symbols from *s*. For example, GC and GAT are both subsequences of GACT. A string *x* is a common subsequence of strings s and *t* if it is a subsequence of both *s* and *t*. For example, G and AT are both common subsequences of GACT and ATG. Our goal is to find a *longest common subsequence (LCS)* of two strings. Note that two strings may have more than one longest common subsequence.

**Input Format.** The first line of the input contains a string *s*, and the second line of the input contains a string *t*.

**Output Format.** A longest common subsequence of *s* and *t*. (**Note:** you can output any of the solutions.)

**Constraints.** |*s*| ≤ 1,000; |*t*| ≤ 1,000

**SAMPLE DATASET:**

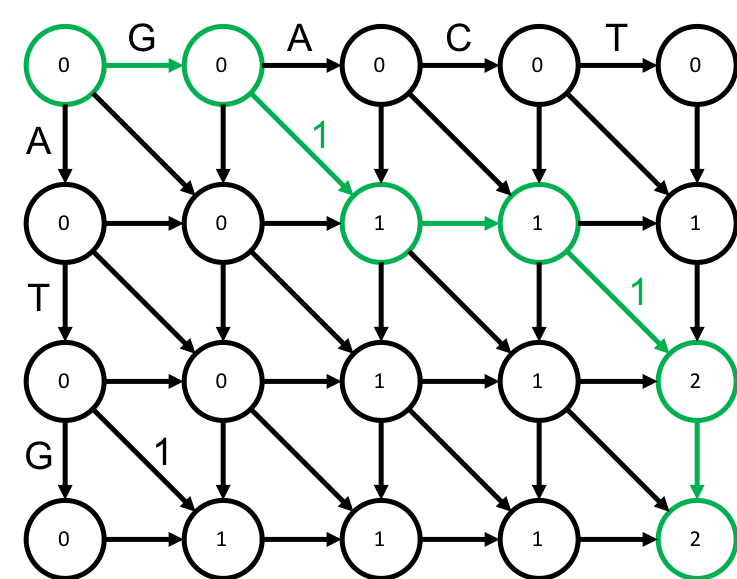
Input:

GACT

ATG

Output:

AT

 A longest common subsequence shared by G**A**C**T** and **AT**G is AT. Although other common substrings exist between *s* and *t*, there are none that are longer. The Figure below represents the longest subsequence as a longest path in the directed graph (shown in green), where diagonal edges corresponding to matching symbols have weight 1 and all other edges have weight 0.

**TEST DATASET 1:**

Input:

ACTGAG

GACTGG

Output:

ACTGG

This dataset checks that you code solves the longest common **subsequence** problem instead of the related longest common **substring** problem. The longest common substring between **ACTG**AG and G**ACTG**G is ACGT, but the longest common subsequence between **ACGT**A**G** and G**ACTGG** is ACTGG.

**TEST DATASET 2:**

Input:

AC

AC

Output:

AC

This simple dataset is used to check if your code correctly reconstructs the longest common subsequence from the backtracking matrix. Common errors include forgetting to reverse the reconstruction before returning (result: CA), terminating reconstruction too early (result: C), and starting reconstruction too late (result: A).

**TEST DATASET 3:**

Input:

GGGGT

CCCCT

Output:

T

This dataset checks that your code correctly considers the last character of each string. Off-by-one errors (can be caused by 0/1 indexing errors) in indexing the input strings could result in the solution erroneously ignoring the final characters of inputs. If your code outputs an empty string it is likely that your implementation includes some off-by-one error.

**TEST DATASET 4:**

Input:

TCCCC

TGGGG

Output:

T

This dataset checks that your code correctly considers the first character of each string. Off-by-one errors (can be caused by 0/1 indexing errors) in indexing the input strings could result in the solution erroneously ignoring the first characters of inputs. If your code outputs an empty string it is likely that your implementation includes some off-by-one error.

**TEST DATASET 5:**

Input:

AA

CGTGGAT

Output:

A

This dataset checks that your code can handle inputs in which the two strings to be aligned are different lengths. If your output is incorrect make sure that your dynamic programming matrix has dimensions or . If your code incorrectly sets the dynamic programming matrix dimensions to or it will not necessarily fail previous datasets since is the same as in all previous test datasets. Make sure that your implementation does not make any assumptions about the sizes of strings *s* and *t*.

**TEST DATASET 6:**

Input:

GGTGACGT

CT

Output:

CT

This dataset checks that your code can handle inputs in which the two strings to be aligned are different lengths. This dataset is similar to Test Dataset 5 except that in this dataset string *s* is longer than string *t*.