

# Dynamic Programming

# Administrivia

- HW3 is out, and due Oct. 23
  - You can work in groups of 3 now if you choose
- Prelim review next Tuesday (Oct. 23)
  - Come with questions!
- Prelim in-class next Thursday (Oct. 25)

# Dynamic Programming

- Useful technique to solve problems that have an “optimal substructure.”
  - i.e. an optimal solution to a problem can be built from optimal solutions to subproblems
  - Ex.  $\text{fib}(n-1)$  and  $\text{fib}(n-2)$  can be used to calculate  $\text{fib}(n)$
- Dynamic Programming also requires “overlapping subproblems.”
  - i.e. there is shared work in the recursive calls
  - Ex.  $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$  <- notice that  $\text{fib}(n-1)$  can be expanded to also need  $\text{fib}(n-2)$
  - Note: if subproblems don't overlap, you may still be able to develop a “Divide and Conquer” algorithm

```
def fib(n):  
    if n == 0 or n == 1:  
        return n  
    return fib(n-1) + fib(n-2)
```

```
mem = {0:0, 1:1}  
def fib(n):  
    if n not in mem:  
        mem[n] = fib(n-1) + fib(n-2)  
    return mem[n]
```

# DP Example: Longest Common Subsequence

- Define a subsequence of a string  $s$  to be a string  $s'$  where all characters of  $s'$  appear in  $s$  and are in the same order in both  $s$  and  $s'$ .
  - Example: MTA, H, ATTN, HAT are all subsequences of MANHATTAN, but TAM is not
- Problem statement: given two strings  $s$  and  $t$ , find the longest subsequence common to both strings.
  - Example: if our strings are ITHACA and MANHATTAN, the LCS would be HAA
- Brute force: enumerate all subsequences of  $s$  and check if each is a subsequence of  $t$ .
  - Runtime complexity:  $O(2^n)$

# DP Example: Longest Common Subsequence

- Does this problem have an optimal substructure?
- Observation #1:
  - Consider the case where  $s$  and  $t$  end in the same letter. Example: MANHATTAN and MADMEN
    - Secretly: by inspection we can see the  $LCS(\text{MANHATTAN}, \text{MADMEN}) = \text{MAN}$
  - Since we know they both end in N, let's guess that  $LCS(\text{MANHATTAN}, \text{MADMEN})$  ends in N
  - Consider  $LCS(\text{MANHATTA}, \text{MADME})$  - by inspection this equals MA
  - Therefore  $LCS(\text{MANHATTA}, \text{MADME}) + \text{N} = \text{MAN} = LCS(\text{MANHATTAN}, \text{MADMEN})$
  - More generally,

$$\text{If } s_n = t_m,$$

$$LCS(s_1 \dots s_n, t_1 \dots t_m) = LCS(s_1 \dots s_{n-1}, t_1 \dots t_{m-1}) + t_m$$

# DP Example: Longest Common Subsequence

- Observation #2:
  - Consider the case where  $s$  and  $t$  do NOT end in the same letter. Example: MANHATTAN and ITHACA
  - Case 1:  $LCS(\text{MANHATTAN}, \text{ITHACA})$  does NOT end in N
    - If so, we don't need it, so  $LCS(\text{MANHATTAN}, \text{ITHACA}) = LCS(\text{MANHATTA}, \text{ITHACA})$
  - Case 2:  $LCS(\text{MANHATTAN}, \text{ITHACA})$  ends in N
    - If so, we don't need the A at the end of ITHACA, so  $LCS(\text{MANHATTAN}, \text{ITHACA}) = LCS(\text{MANHATTAN}, \text{ITHAC})$
  - But... we don't know which case is true *a priori*
  - So, generally:

If  $s_n \neq t_m$ ,

$$LCS(s_1 \dots s_n, t_1 \dots t_m) = \max(LCS(s_1 \dots s_{n-1}, t_1 \dots t_m) + LCS(s_1 \dots s_n, t_1 \dots t_{m-1}))$$

# DP Example: Longest Common Subsequence

- Observation #3:
  - If at least one of  $s$  or  $t$  is the empty string, then  $LCS(s, t)$  is also the empty string

# DP Example: Longest Common Subsequence

$$LCS(s_1 \dots s_n, t_1 \dots t_m) = \begin{cases} \text{“ ”} & \text{if } n = 0 \text{ or } m = 0 \\ LCS(s_1 \dots s_{n-1}, t_1 \dots t_{m-1}) + t_m & \text{if } s_n = t_m \\ \max(LCS(s_1 \dots s_{n-1}, t_1 \dots t_m), LCS(s_1 \dots s_n, t_1 \dots t_{m-1})) & \text{otherwise} \end{cases}$$

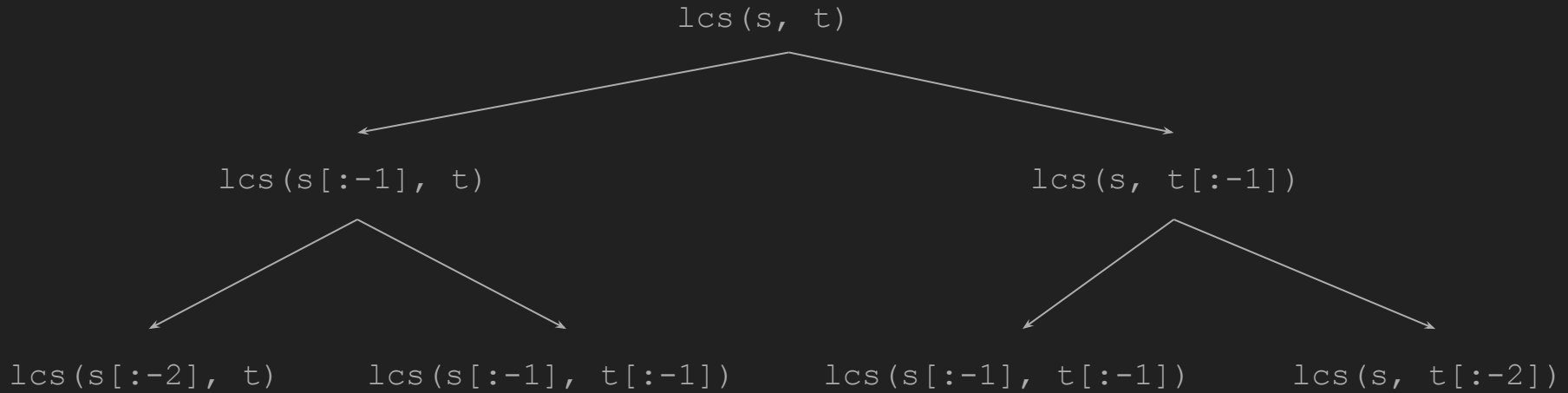
Does this problem have an optimal substructure? Yes!



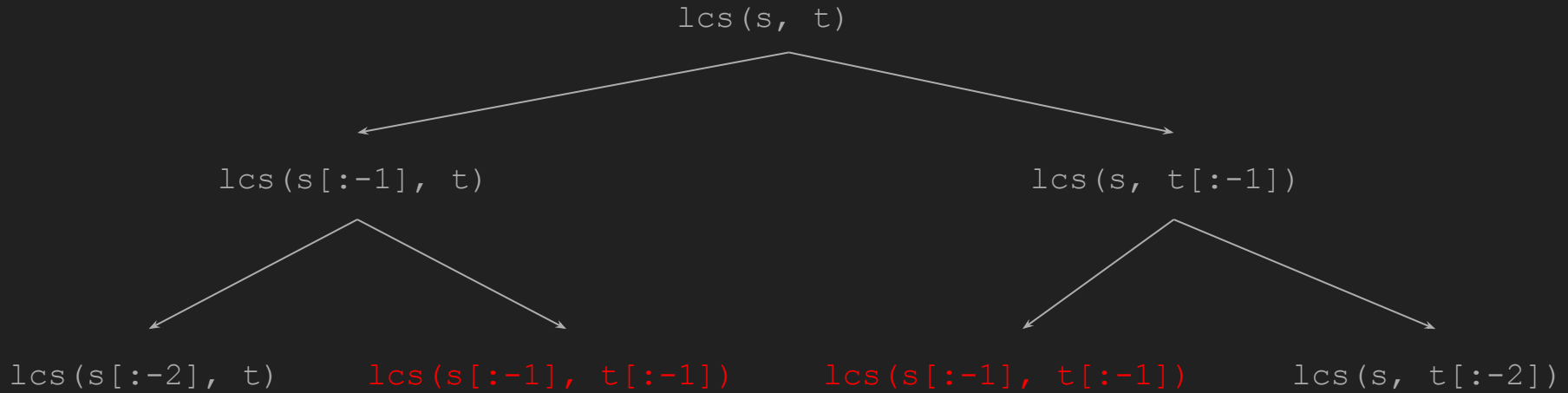
# LCS: Naive Implementation

```
def lcs(s, t):  
    if len(s) == 0 or len(t) == 0:  
        return ""  
  
    if s[-1] == t[-1]:  
        return lcs(s[:-1], t[:-1]) + t[-1]  
  
    tmp1 = lcs(s[:-1], t)  
    tmp2 = lcs(s, t[:-1])  
  
    return tmp1 if len(tmp1) > len(tmp2) else tmp2
```

# LCS: Naive Implementation



# LCS: Naive Implementation



Runtime complexity:  $O(2^n)$

# LCS: Recursive Implementation with Memoization

```
mem = {}
def lcs(s, t):
    if (s, t) in mem:
        return mem[(s, t)]
    if len(s) == 0 or len(t) == 0:
        return ""
    if s[-1] == t[-1]:
        mem[(s, t)] = lcs(s[:-1], t[:-1]) + t[-1]
    else:
        tmp1 = lcs(s[:-1], t)
        tmp2 = lcs(s, t[:-1])
        mem[(s, t)] = tmp1 if len(tmp1) > len(tmp2) else tmp2
    return mem[(s, t)]
```

# LCS: Alternative implementation with “table-filling”

	X	A	G	W	T
X					
A					
G					
T					

# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""					
G	""					
T	""					

# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""					
G	""					
T	""					

The table above illustrates the initial state of a dynamic programming table for finding the Longest Common Subsequence (LCS) between the strings "XAGWT" and "XAGWT". The columns represent the characters of the first string (X, A, G, W, T) and the rows represent the characters of the second string (X, A, G, T). The top-left cell (row "", column "") is shaded gray, indicating it is the starting point for the table-filling process. Arrows point from the cell (row A, column X) to the cell (row "", column X) and from the cell (row A, column X) to the cell (row A, column A), suggesting the next steps in the table-filling process.

# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	""			
G	""					
T	""					

The diagram illustrates a dynamic programming table for finding the Longest Common Subsequence (LCS) between the strings "XAGWT" and "XAGWT". The table is a 5x5 grid. The columns are labeled with the characters of the first string: "", X, A, G, W, T. The rows are labeled with the characters of the second string: "", A, G, T. The cells containing "" represent the base cases where the LCS length is 0. The cell at row 'A', column 'X' contains an arrow pointing up to the cell at row 'A', column 'A', and an arrow pointing left to the cell at row 'A', column 'X'. This indicates that the LCS length for "AX" is 1, derived from the LCS length for "A" (which is 1) and "X" (which is 0).



# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""				
G	""					
T	""					

The table illustrates the dynamic programming table for finding the Longest Common Subsequence (LCS) between the strings "XAGWT" and "XAGWT". The table is a 5x5 grid. The top row and left column are labeled with the characters of the strings. The cell at row 'A', column 'X' is shaded gray. Arrows indicate the backpointers: a vertical arrow from the cell (A, X) to the cell (A, A), a horizontal arrow from the cell (A, X) to the cell (A, G), and a diagonal arrow from the cell (A, X) to the cell (A, W).

# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A			
G	""					
T	""					

The diagram illustrates a dynamic programming table for finding the Longest Common Subsequence (LCS) between the strings "XAGWT" and "XAGWT". The table is a 5x5 grid. The columns are labeled with the characters of the first string: "", X, A, G, W, T. The rows are labeled with the characters of the second string: "", A, G, T. The cell at row 'A', column 'A' contains the character 'A'. The cell at row 'X', column 'X' contains the character 'X'. The cell at row 'A', column 'X' contains the character 'X'. The cell at row 'A', column 'A' contains the character 'A'. The cell at row 'G', column 'G' contains the character 'G'. The cell at row 'W', column 'W' contains the character 'W'. The cell at row 'T', column 'T' contains the character 'T'. Arrows indicate the path of the LCS: a vertical arrow points from the cell (A, X) to (X, X), and a diagonal arrow points from (X, X) to (A, A).

# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A	""		
G	""					
T	""					

The table illustrates the dynamic programming table for finding the Longest Common Subsequence (LCS) between the strings "XAGWT" and "XAGWT". The table is a 5x5 grid. The columns are labeled with the characters of the first string: "", X, A, G, W, T. The rows are labeled with the characters of the second string: "", A, G, T. The cells contain the LCS value for the corresponding prefixes. The cell at row 'A', column 'G' is shaded gray. Arrows indicate the backpointers: a vertical arrow from (A, G) to (A, A), a horizontal arrow from (A, G) to (A, X), a diagonal arrow from (A, G) to (X, A), and a vertical arrow from (X, A) to (X, X).

# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A	A		
G	""					
T	""					

The diagram illustrates a dynamic programming table for finding the Longest Common Subsequence (LCS) between the strings "XAGWT" and "XAGWT". The table is a 5x5 grid. The columns are labeled with the characters of the first string: "", X, A, G, W, T. The rows are labeled with the characters of the second string: "", A, G, T. The cells contain either the empty string "" or the character 'A'. Arrows indicate the path of the LCS: a vertical arrow from the cell (A, X) to (A, A), a horizontal arrow from (A, X) to (A, ""), a vertical arrow from (A, G) to (A, A), a horizontal arrow from (A, G) to (A, A), and a diagonal arrow from (A, A) to (A, X).

# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A	A	A	A
G	""					
T	""					

The diagram illustrates a dynamic programming table for finding the Longest Common Subsequence (LCS) between the strings "XAGWT" and "XAGWT". The table is a 5x5 grid. The columns are labeled with the characters of the first string: "", X, A, G, W, T. The rows are labeled with the characters of the second string: "", A, G, T. The cells contain either the empty string "" or the character A. Arrows indicate the path of the LCS: a diagonal arrow from (A, X) to (A, A), and vertical arrows from (A, A) to (A, G), (A, G) to (A, W), and (A, W) to (A, T). The cell (A, A) is the starting point of the LCS.

# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A	A	A	A
G	""					
T	""					

The diagram illustrates the table-filling process for the Longest Common Subsequence (LCS) problem. The table is a 4x6 grid with columns labeled "" (empty string), X, A, G, W, and T, and rows labeled "", A, G, and T. The cell at row A, column X is shaded gray. Arrows indicate the direction of the fill: vertical arrows point up from the row below, and horizontal arrows point left from the column to the right. A diagonal arrow points from the cell (A, X) to the cell ( "", X).

# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A	A	A	A
G	""	""				
T	""					

The diagram illustrates the table-filling process for the Longest Common Subsequence (LCS) problem. The table is a 4x6 grid with columns labeled "" (empty string), X, A, G, W, and T, and rows labeled "", A, G, and T. The cells contain the LCS value for the corresponding prefixes. Arrows indicate the direction of the update: vertical arrows point up from the cell below to the cell above, and horizontal arrows point left from the cell to the right. A diagonal arrow points from the cell (A, X) to the cell (A, A).

# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A	A	A	A
G	""	""				
T	""					

The diagram illustrates the table-filling process for the Longest Common Subsequence (LCS) problem. The table is a 4x6 grid with columns labeled "" (empty string), X, A, G, W, and T, and rows labeled "", A, G, and T. The cells contain the LCS value for the corresponding prefixes. Arrows indicate the direction of the update: vertical arrows point up from the cell below, and horizontal arrows point left from the cell to the right. A diagonal arrow points from the cell (A, X) to the cell (A, A). The cell (G, A) is shaded gray, indicating it is the current cell being filled.



# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A	A	A	A
G	""	""	A			
T	""					

The diagram illustrates the table-filling process for the Longest Common Subsequence (LCS) problem. The table is a 4x6 grid with columns labeled "" (empty string), X, A, G, W, and T, and rows labeled "", A, G, and T. The cells contain the LCS value for the corresponding prefixes. Arrows indicate the direction of the update: vertical arrows point up from the cell below to the cell above, and horizontal arrows point left from the cell to the right. A diagonal arrow points from the cell (A, X) to the cell ( "", X).

# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A	A	A	A
G	""	""	A			
T	""					

The diagram illustrates the table-filling process for the Longest Common Subsequence (LCS) problem. The table is a 4x6 grid with columns labeled "" (empty string), X, A, G, W, and T, and rows labeled "", A, G, and T. The cells contain the LCS value for the corresponding prefixes. Arrows indicate the direction of the update: vertical arrows point up from the cell below, horizontal arrows point left from the cell to the right, and diagonal arrows point from the bottom-right cell to the top-left cell. The cell (G, G) is shaded gray, indicating it is the current cell being filled. The LCS values are: (row, col) = (0,0) to (0,6) are ""; (1,0) to (1,6) are ""; (2,0) is "", (2,1) is "", (2,2) is A, (2,3) is A, (2,4) is A, (2,5) is A; (3,0) is "", (3,1) is "", (3,2) is A, (3,3) is empty; (4,0) is "", (4,1) to (4,6) are empty.

# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A	A	A	A
G	""	""	A	AG		
T	""					

The diagram illustrates the table-filling process for finding the Longest Common Subsequence (LCS) between the strings "XAGWT" and "XAGWT". The table is a 5x5 grid where the top row and left column represent the prefixes of the strings. The cells contain the LCS for those prefixes. Arrows indicate the direction of the fill: vertical arrows point up from the cell below to the cell above, and horizontal arrows point left from the cell to the right. Diagonal arrows point from the bottom-right cell to the top-left cell, indicating the start of the fill. The LCS values are: "" for all prefixes of length 1; "A" for prefixes of length 2 and 3; "AG" for the prefix of length 4; and "" for the prefix of length 5.

# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A	A	A	A
G	""	""	A	AG	AG	AG
T	""					

The diagram illustrates the table-filling process for the Longest Common Subsequence (LCS) problem. The table is a 4x6 grid with columns labeled "", X, A, G, W, T and rows labeled "", A, G, T. The cells contain the LCS string for the prefixes of the two sequences. Arrows indicate the transitions between cells during the filling process:

- From (row "", col "") to (row "", col X): diagonal arrow (no match).
- From (row "", col X) to (row A, col X): vertical arrow (match).
- From (row "", col X) to (row A, col A): diagonal arrow (no match).
- From (row A, col "") to (row A, col X): horizontal arrow (no match).
- From (row A, col X) to (row A, col A): vertical arrow (match).
- From (row A, col X) to (row A, col G): horizontal arrow (no match).
- From (row A, col A) to (row A, col G): horizontal arrow (no match).
- From (row A, col A) to (row G, col A): vertical arrow (match).
- From (row A, col G) to (row G, col G): horizontal arrow (no match).
- From (row A, col G) to (row A, col W): horizontal arrow (no match).
- From (row A, col G) to (row A, col T): horizontal arrow (no match).
- From (row G, col "") to (row G, col X): horizontal arrow (no match).
- From (row G, col X) to (row G, col A): horizontal arrow (no match).
- From (row G, col X) to (row G, col G): horizontal arrow (no match).
- From (row G, col A) to (row G, col G): horizontal arrow (no match).
- From (row G, col A) to (row G, col W): horizontal arrow (no match).
- From (row G, col A) to (row G, col T): horizontal arrow (no match).
- From (row G, col G) to (row G, col W): horizontal arrow (no match).
- From (row G, col G) to (row G, col T): horizontal arrow (no match).
- From (row G, col G) to (row T, col G): vertical arrow (match).
- From (row G, col W) to (row T, col W): vertical arrow (match).
- From (row G, col T) to (row T, col T): vertical arrow (match).

# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	"" ↑	""	"" ↑	"" ↑	"" ↑
A	"" ←	"" ↑	A ↑	A ←	A ←	A ←
G	"" ←	"" ↑	A ↑	AG ↑	AG ←	AG ←
T	"" ←	"" ↑	A ↑	AG ↑	AG ←	

The diagram illustrates the table-filling process for the Longest Common Subsequence (LCS) problem. The grid shows the characters of the strings being compared: "" (empty string), X, A, G, W, and T. The cells contain the LCS value for the prefixes of the two strings. Arrows indicate the direction of the transition from one cell to the next, showing how the LCS value is updated based on the characters of the two strings. For example, the LCS value for "" and "" is "", and for "" and X is "", indicating no match. The LCS value for A and A is A, and for G and G is AG, showing that the common subsequence is built up as the strings are processed.

# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A	A	A	A
G	""	""	A	AG	AG	AG
T	""	""	A	AG	AG	

The diagram illustrates the table-filling process for the Longest Common Subsequence (LCS) problem. The table is a 5x6 grid with columns labeled "", X, A, G, W, T and rows labeled "", A, G, T. The cells contain the LCS string for the prefixes of the two strings. Arrows indicate the direction of the transition from the previous cell to the current one: vertical arrows for matches, horizontal arrows for insertions, and diagonal arrows for deletions. The cell (T, T) is shaded gray, indicating it is the final state of the table.

# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A	A	A	A
G	""	""	A	AG	AG	AG
T	""	""	A	AG	AG	AGT

The diagram illustrates the table-filling process for the Longest Common Subsequence (LCS) problem. The table is a 5x6 grid with columns labeled "", X, A, G, W, T and rows labeled "", A, G, T. The cells contain the LCS string for the prefixes of the two sequences. Arrows indicate the direction of the transition from one cell to the next, showing the path of the algorithm. Diagonal arrows indicate matches (e.g., from (A, X) to (A, A), from (G, X) to (G, A), from (T, X) to (T, A), from (A, G) to (A, AG), from (G, G) to (G, AG), from (T, G) to (T, AG)), and horizontal arrows indicate insertions (e.g., from (A, "") to (A, A), from (G, "") to (G, A), from (T, "") to (T, A), from (A, A) to (A, AG), from (G, A) to (G, AG), from (T, A) to (T, AG)). A vertical arrow from (A, A) to (A, AG) and a horizontal arrow from (G, AG) to (G, AGT) indicate the final steps of the algorithm.

# LCS: Alternative implementation with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A	A	A	A
G	""	""	A	AG	AG	AG
T	""	""	A	AG	AG	AGT

The diagram illustrates the table-filling process for the Longest Common Subsequence (LCS) problem. The table is a 4x6 grid with columns labeled "" (empty string), X, A, G, W, and T, and rows labeled "", A, G, and T. The cells contain the LCS string for the corresponding prefixes. Arrows indicate the direction of the transition from one cell to the next, showing how the LCS is built up. The final LCS, "AGT", is highlighted in green in the bottom-right cell.



# LCS: Iterative Implementation with “table filling”

```
def lcs(s, t):  
    matrix = [[""] for x in range(len(t)+1)] for y in range(len(s)+1)]  
    for i in range(1, len(s)+1):  
        for j in range(1, len(t)+1):  
            if s[i-1] == t[j-1]:  
                matrix[i][j] = matrix[i-1][j-1] + t[j-1]  
            else:  
                tmp1 = matrix[i-1][j]  
                tmp2 = matrix[i][j-1]  
                matrix[i][j] = tmp1 if len(tmp1) > len(tmp2) else tmp2  
    return matrix[len(s)][len(t)]
```

# DP Example: Longest common subsequence

- Iterative “table-filling” runtime complexity
  - Filling in an  $n * m$  grid, so  $O(nm)$
  - Space is worse, because we’re storing the whole string
    - Can improve by only storing the path to the previous call, and reconstruct answer later

# LCS: Space saving with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A	A	A	A
G	""	""	A	AG	AG	AG
T	""	""	A	AG	AG	AGT

The diagram illustrates the space-saving technique in the Longest Common Subsequence (LCS) table-filling algorithm. The table is a 4x6 grid with columns labeled "", X, A, G, W, T and rows labeled "", A, G, T. The cells contain the LCS string for the prefixes of the two sequences. Arrows indicate the direction of the transition from the previous cell to the current one: vertical arrows for matches, horizontal arrows for insertions, and diagonal arrows for deletions. The diagonal arrows are located at (A, X), (G, A), and (T, W).

# LCS: Space saving with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A	A	A	A
G	""	""	A	AG	AG	AG
T	""	""	A	AG	AG	AGT

The diagram illustrates the filling of an LCS table for strings "XAGW" and "ATAGT". The table is a 5x6 grid. The columns are labeled with characters from the first string: "", "X", "A", "G", "W", "T". The rows are labeled with characters from the second string: "", "A", "G", "T".

The table content is as follows:

- Row 1 (Header): "", "X", "A", "G", "W", "T"
- Row 2 (Header): "", "", "", "", "", ""
- Row 3: "", "", "", "A", "A", "A", "A"
- Row 4: "", "", "", "A", "AG", "AG", "AG"
- Row 5: "", "", "A", "AG", "AG", "AGT"

Arrows indicate the path of the longest common subsequence (AGT):

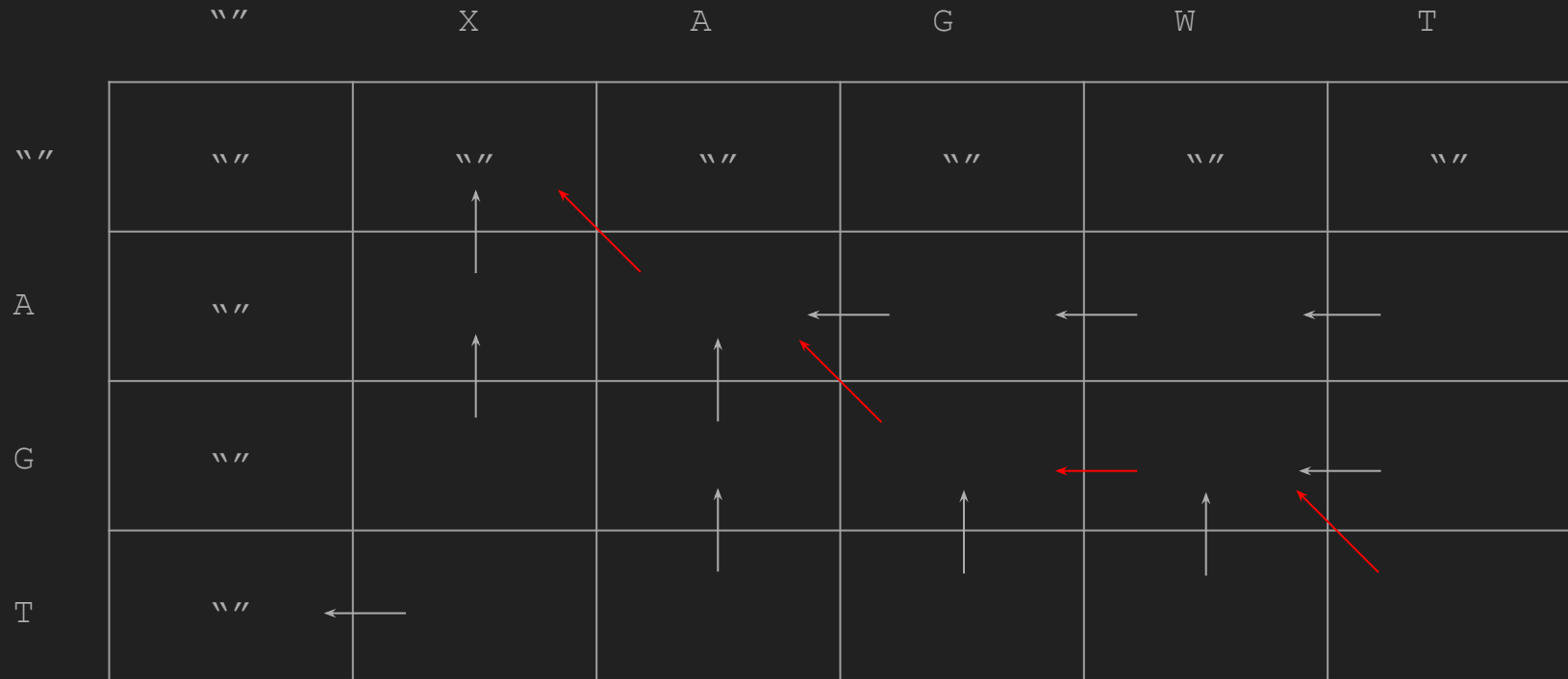
- From (3,3) to (3,4)
- From (3,4) to (3,5)
- From (3,5) to (3,6)
- From (4,3) to (4,4)
- From (4,4) to (4,5)
- From (4,5) to (4,6)
- From (5,2) to (5,3)
- From (5,3) to (5,4)
- From (5,4) to (5,5)
- From (5,5) to (5,6)

# LCS: Space saving with “table-filling”

	""	X	A	G	W	T
""	""	""	""	""	""	""
A	""	""	A	A	A	A
G	""	""	A	AG	AG	AG
T	""	""	A	AG	AG	AGT

The diagram illustrates the space-saving technique in the LCS table-filling algorithm. The table is a 4x6 grid. The columns are labeled with the characters of the second string: "", X, A, G, W, T. The rows are labeled with the characters of the first string: "", A, G, T. The cells contain the longest common subsequence (LCS) for the prefixes of the two strings. Red arrows indicate the path from the bottom-right cell (T, T) containing 'AGT' back to the top-left cell ('', '') containing '': (T, T) to (T, W) containing 'AG', (T, W) to (G, W) containing 'AG', (G, W) to (G, A) containing 'A', (G, A) to (A, A) containing 'A', and (A, A) to (A, X) containing 'A'. White arrows indicate the backpointers used to reconstruct the LCS: (T, T) ← (T, W), (T, W) ← (G, W), (G, W) ← (G, A), (G, A) ← (A, A), (A, A) ← (A, X).

# LCS: Space saving with “table-filling”



# DP Example: Longest common subsequence

- Iterative “table-filling” runtime complexity
  - Filling in an  $n * m$  grid, so  $O(nm)$
  - Space is worse, because we’re storing the whole string
    - Can improve by only storing the path to the previous call, and reconstruct answer later
- Recursive memoization runtime complexity
  - Essentially memoizing values for the cells visited
  - $O(nm)$  still a reasonable upper bound
  - Space can be improved in a similar way
- Practical applications
  - diff
  - version control systems
  - bioinformatics
  - computational linguistics

# LCS application: diff

Sequence 1: A B D F H Y Z

Sequence 2: A B C F H W X Y Z



# LCS application: diff

Sequence 1: A B D F H Y Z

Sequence 2: A B C F H W X Y Z

LCS: A B F H Y Z

diff: D C W X  
- + + +

DP Example:

# DP Example: Dijkstra's Algorithm

- Yes, really!
- Recall: if the shortest path from  $s$  to  $t$  goes through  $k$ , then the subpath from  $s$  to  $k$  is also the shortest path from  $s$  to  $k$ 
  - This is our optimal substructure!
- Dijkstra's is sort of a "table-filling" algorithm
  - Table dimensions are source cells and sink cells
  - Priority queue tells you which order to fill in cells
  - Your "visited set" is the memoized solutions to subproblems

# DP Example: Floyd-Warshall algorithm

- Solution to shortest path problem, like Dijkstra's algorithm
  - Supports negative edges! But still not negative cycles...
  - Dynamic Programming connection is more explicit
- Given: a graph  $g$  with vertices labeled  $\{1, \dots, n\}$ .
- Consider  $shortestPath(i, j, k)$ 
  - Computes the shortest path from  $i$  to  $j$  only using nodes in  $\{1, \dots, k\}$  as intermediate nodes
  - Could be one of two cases:
    - The path does not contain  $k$  (so the path only contains nodes in  $\{1, \dots, k-1\}$ )
    - The path does contain  $k$ , therefore the path is made up of a path from  $i$  to  $k$  plus a path from  $k$  to  $j$ , each of which only contains nodes in  $\{1, \dots, k-1\}$
- If  $w(i, j)$  is the weight of the edge from  $i$  to  $j$ , then:
  - $shortestPath(i, j, 0) = w(i, j)$
  - $shortestPath(i, j, k) = \min(shortestPath(i, j, k-1), shortestPath(i, k, k-1) + shortestPath(k, j, k-1))$