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. fib(n-1) and fib(n-2) can be used to calculate fib(n)
- Dynamic Programming also requires “overlapping subproblems.”
    - i.e. there is shared work in the recursive calls
    - Ex. fib(n) = fib(n-1) + fib(n-2)   <- notice that fib(n-1) can be expanded to also need 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(*MANHATTAN, MADMEN*)* = MAN
  - Since we know they both end in N, let's guess that *LCS(*MANHATTAN, MADMEN*)* ends in N
  - Consider *LCS(*MANHATTA, MADME*)* - by inspection this equals MA
  - Therefore *LCS(*MANHATTA, MADME*)* + N = MAN = *LCS(*MANHATTAN, MADMEN*)*
  - More generally,

If $s_n = t_m$,

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

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(*MANHATTAN, ITHACA*)* does NOT end in N
    - If so, we don't need it, so *LCS(*MANHATTAN, ITHACA*)* = *LCS(*MANHATTA, ITHACA*)*
  - Case 2: *LCS(*MANHATTAN, ITHACA*)* ends in N
    - If so, we don't need the A at the end of ITHACA, so *LCS(*MANHATTAN, ITHACA*)* = *LCS(*MANHATTAN, ITHAC*)*
  - But… we don't know which case is true *a priori*
  - So, generally:

If $s_n \neq t_m$,

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

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

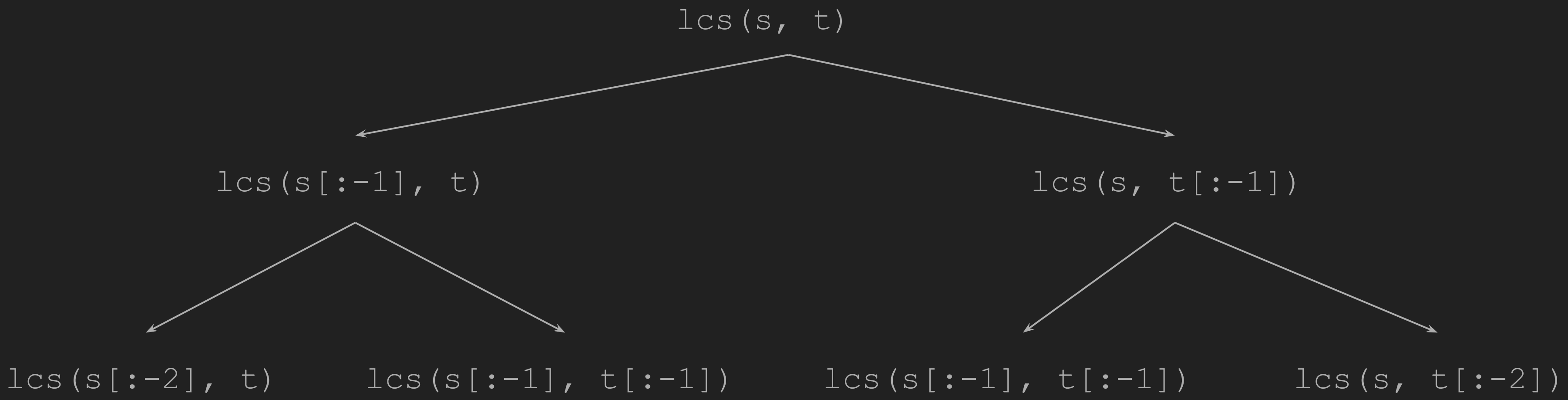
$$LCS(s_1 ... s_n, t_1 ... t_m) = \begin{cases} \text{“ ”} & \text{if } n = 0 \text{ or } m = 0 \\ LCS(s_1 ... s_{n-1}, t_1 ... t_{m-1}) + t_\mathrm{m} & \text{if } s_n = t_m \\ max(LCS(s_1 ... s_{n-1}, t_1 ... t_m), LCS(s_1 ... s_n, t_1 ... t_{m-1})) & \text{otherwise} \end{cases}$$

Does this problem have an optimal substructure? Yes!

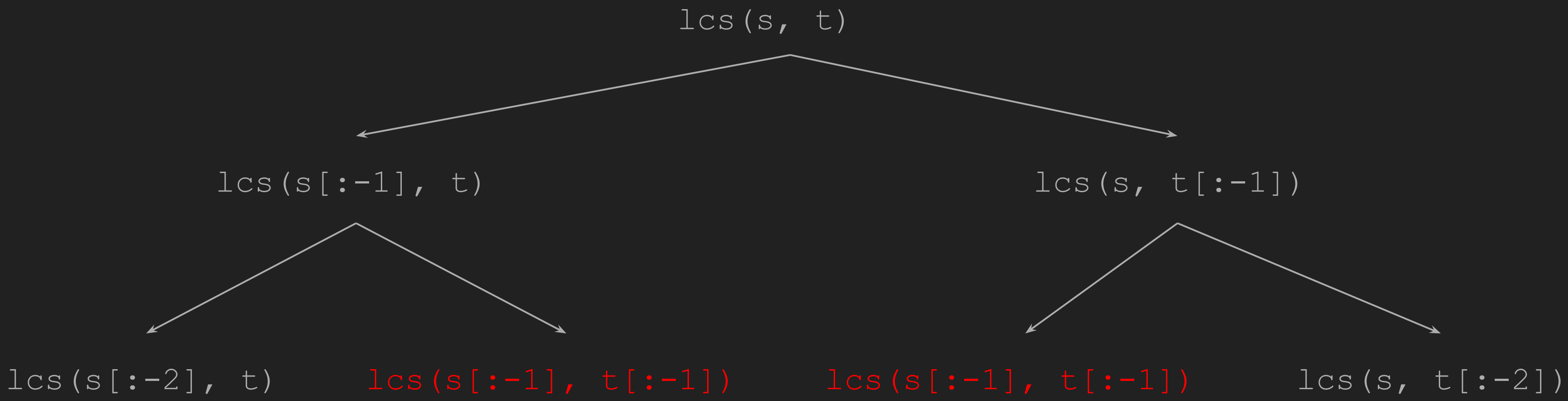
LCS: Naive Implementation

```
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
""						
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	“”					

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	“”					

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A			
G	“”					
T	“”					

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A			
G	“”					
T	“”					

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A		
G	“”					
T	“”					

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”					
T	“”					

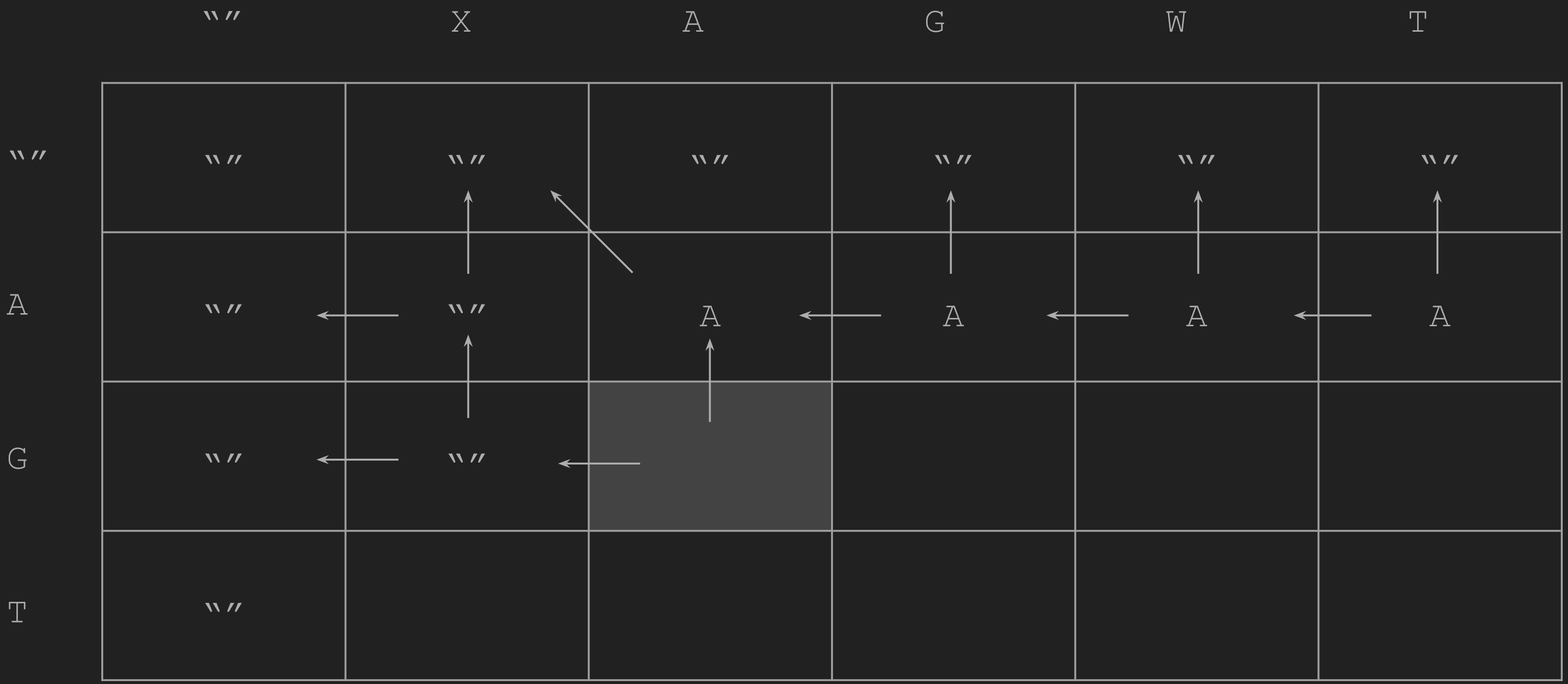
LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”					
T	“”					

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”	“”				
T	“”					

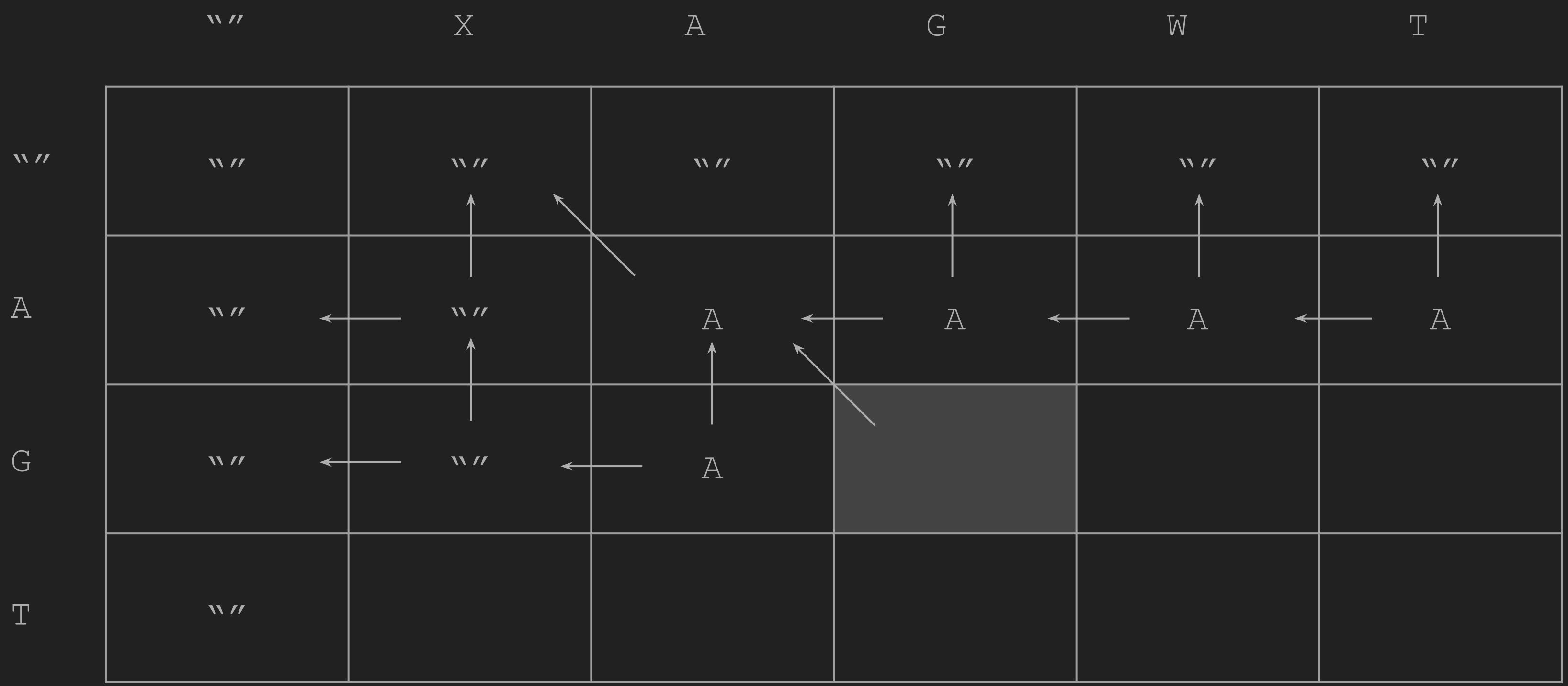
LCS: Alternative implementation with “table-filling”



LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”	“”	A			
T	“”					

LCS: Alternative implementation with “table-filling”



LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”	“”	A	AG		
T	“”					

LCS: Alternative implementation with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”	“”	A	AG	AG	AG
T	“”					

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	

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	

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

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

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

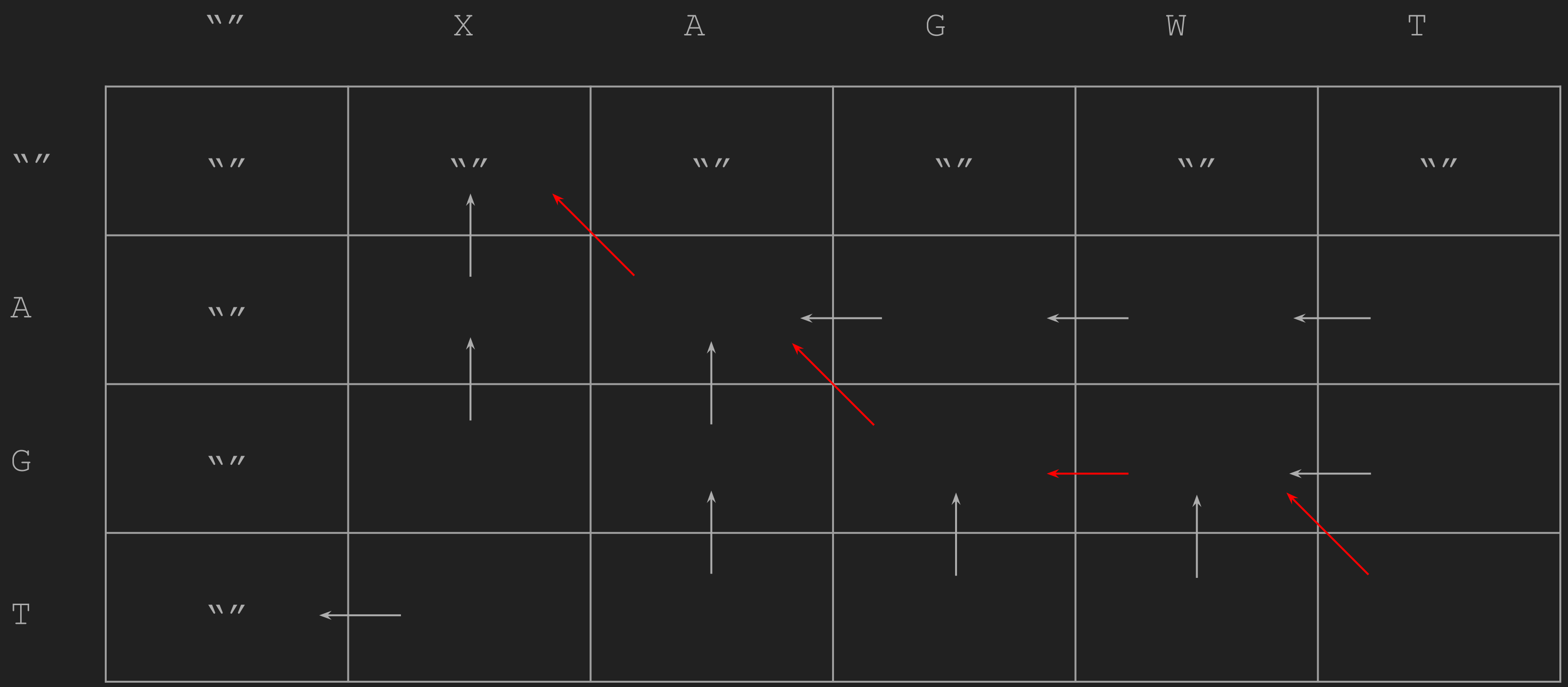
LCS: Space saving with “table-filling”

	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”	“”	A	AG	AG	AG
T	“”	“”	A	AG	AG	AGT

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

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*, than 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, …, n}*.
- Consider *shortestPath(i, j, k)*
  - Computes the shortest path from *i* to *j* only using nodes in {1, …, *k*} as intermediate nodes
  - Could be one of two cases:
    - The path does not contain *k* (so the path only contains nodes in {1, …, *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, …, *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))*