Starting at top-left of \( nxm \) grid, moving only down or right, how many ways to reach bottom-right?
Starting at top-left of \( n \times m \) grid, moving only down or right, how many ways to reach bottom-right?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recursive form?
Starting at top-left of \(nxm\) grid, moving only down or right, how many ways to reach bottom-right?

\[
\]
Starting at top-left of n × m grid, moving only down or right, how many ways to reach bottom-right?

\[ A[r, c] \]

- \[ A[r-1, c] \]
- \[ A[r, c-1] \]
Starting at top-left of nxm grid, moving only down or right, how many ways to reach bottom-right?
Starting at top-left of nxm grid, moving only down or right, how many ways to reach bottom-right?

\[ A[r,c] \]

\[ A[r-1,c] \]

\[ A[r-2,c] \]

\[ A[r-3,c] \]

\[ A[r-1,c-1] \]

\[ A[r-2,c-1] \]

\[ A[r-2,c-2] \]

\[ A[r-1,c-2] \]

\[ A[r-1,c-3] \]

\[ A[r,c-1] \]

\[ A[r-2,c-1] \]

\[ A[r-2,c-2] \]

\[ A[r-1,c-2] \]

\[ A[r-1,c-3] \]
Starting at top-left of nxm grid, moving only down or right, how many ways to reach bottom-right?

\[
A[r,c] = \begin{cases} 
A[r-1,c] + A[r,c-1] & \text{if } r > 1 \text{ and } c > 1 \\
1 & \text{if } r = 1 \text{ or } c = 1 \\
0 & \text{otherwise}
\end{cases}
\]

\[
\min \{r,c\} \text{ full levels } \Omega(2^n) \text{ for nxn}
\]
Starting at top-left of nxm grid, moving only down or right, how many ways to reach bottom-right?

\[ A[r,c] \]

**repetitive subproblems**

want to avoid repetition

\[ A[r-1,c] \]

\[ A[r,c-1] \]

\[ [r-2,c-1] \]

\[ [r-3,c] \]

\[ [r-2,c] \]

\[ [r-2,c-1] \]

\[ [r-1,c-2] \]

\[ [r-3,c-1] \]

\[ [r-2,c-1] \]

\[ [r-1,c] \]

\[ [r-2,c-1] \]

\[ [r-1,c-2] \]

\[ [r-1,c-2] \]

\[ [r-1,c-3] \]

\[ [r,1] \]

\[ \min\{r,c\} \] full levels

\[ \Omega(2^n) \] for nxn
Starting at top-left of nxm grid, moving only down or right, how many ways to reach bottom-right?
Starting at top-left of nxm grid, moving only down or right, how many ways to reach bottom-right?
Starting at top-left of \( nxm \) grid, moving only down or right, how many ways to reach bottom-right?
Starting at top-left of nxm grid, moving only down or right, how many ways to reach bottom-right?
Starting at top-left of \( nxm \) grid, moving only down or right, how many ways to reach bottom-right?
How many times will we recurse in a unique way?
How many times will we recurse in a unique way?

\[ A[r,c] \rightarrow r \cdot c \text{ distinct subproblems} \]
**MEMOIZATION** (making memos)

For this problem, mxn table


<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recursion:
- First find \( A[r-1,c] \) up.
- Then find \( A[r,c-1] \) left.
**Memoization** (making memos)

For this problem, $m \times n$ table


<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Recursion:**
- First find $A[r-1, c] \uparrow$
- Then find $A[r, c-1] \downarrow$
MEMOIZATION (making memos)

For this problem, m x n table


<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recursion:
- first find \( A[r-1,c] \) up
- then find \( A[r,c-1] \) left
MEMOIZATION (making memos)

For this problem, an $m \times n$ table

MEMOIZATION (making memos)

For this problem, mxn table


Recursion:
- first find \( A[r-1,c] \)
- then find \( A[r,c-1] \)
MEMOIZATION (making memos)

For this problem, m x n table


Recursion:
- first find \( A[r-1,c] \)
- then find \( A[r,c-1] \)
**MEMOIZATION**  (making memos)

For this problem, m x n table  \[ A[r,c] = A[r-1,c] + A[r,c-1] \]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r,c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Recursion:**
- First find \( A[r-1,c] \) ↑
- Then find \( A[r,c-1] \) ←
MEMOIZATION (making memos)

For this problem, m x n table


Recursion:
- First find \( A[r-1, c] \) up
- Then find \( A[r, c-1] \) left
MEMOIZATION (making memos)

For this problem, m x n table


Recursion:
- first find \( A[r-1,c] \) ↑
- then find \( A[r,c-1] \) ←
**MEMOIZATION**  (making memos)

For this problem, an $m \times n$ table


<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recursion:
- first find $A[r-1,c]$
- then find $A[r,c-1]$
MEMOIZATION  (making memos)

For this problem, m x n table

\[
\]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recursion:
- first find \( A[r-1,c] \)
- then find \( A[r,c-1] \)
**MEMOIZATION**  (making memos)

For this problem, mxn table

\[
\]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recursion:
- First find \(A[r-1, c]\)
- Then find \(A[r, c-1]\)

[Diagram showing the table and recursive process]
**MEMOIZATION** (making memos)

For this problem, m x n table


Recursion:
- first find $A[r-1, c]$ ↑
- then find $A[r, c-1]$ ←

$$\Theta(n \cdot m)$$

time & space
Starting at top-left of nxm grid, moving only down or right, how many ways to reach bottom-right?

**DYNAMIC PROGRAMMING** *(bottom-up : base cases first)*

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Starting at top-left of nxm grid, moving only down or right, how many ways to reach bottom-right?

**Dynamic Programming** (bottom-up: base cases first)

\[
\]

Fill any cell as long as what it depends on is full.
Starting at top-left of nxm grid, moving only down or right, how many ways to reach bottom-right?

**Dynamic Programming** (bottom-up: base cases first)

\[
\]

Fill any cell as long as what it depends on is full.
Starting at top-left of nxm grid, moving only down or right, how many ways to reach bottom-right?

**Dynamic Programming** (bottom-up: base cases first)

\[
\]

Fill any cell as long as what it depends on is full.
Starting at top-left of nxm grid, moving only down or right, how many ways to reach bottom-right?

**DYNAMIC PROGRAMMING** (bottom-up : base cases first)

\[
\]

fill any cell as long as what it depends on is full
Starting at top-left of nxm grid, moving only down or right, how many ways to reach bottom-right?

**Dynamic Programming** (bottom-up: base cases first)

\[
\]

Fill any cell as long as what it depends on is full.
Starting at top-left of $n \times m$ grid, moving only down or right, how many ways to reach bottom-right?

**Dynamic Programming** (bottom-up: base cases first)


<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Starting at top-left of \( nxm \) grid, moving only down or right, how many ways to reach bottom-right?

**Dynamic Programming** (bottom-up: base cases first)

\[
\]

![Dynamic Programming Grid](image)

Fill any cell as long as what it depends on is full.
Dynamic Programming - Longest Increasing Subsequence

23, 3, 5, 18, 10, 101, 12, 14, 4, 105
Dynamic Programming - Longest Increasing Subsequence

23, 3, 5, 18, 10, 101, 12, 14, 4, 105
Dynamic Programming - Longest Increasing Subsequence

23, 3, 5, 18, 10, 101, 12, 14, 4, 105
Dynamic Programming - Longest Increasing Subsequence

23, 3, 5, 18, 10, 101, 12, 14, 4, 105
Dynamic Programming - Longest Increasing Subsequence

$S: \ 23, \ 3, \ 5, \ 18, \ 10, \ 101, \ 12, \ 14, \ 4, \ 105$

$L(S) = 3, 5, 10, 12, 14, 105 \quad |L(S)| = 6$
Dynamic Programming - Longest Increasing Subsequence

$S: \quad 23, 3, 5, 18, 10, 101, 12, 14, 4, 105$

$L(S) = 3, 5, 10, 12, 14, 105 \quad |L(S)| = 6$

Could try including/excluding every element:

$2^n$ subsequences to check
Dynamic Programming - Longest Increasing Subsequence

\[ S: \ 23, \ 3, \ 5, \ 18, \ 10, \ 101, \ 12, \ 14, \ 4, \ 105 \]

\[ L(S) = 3, \ 5, \ 10, \ 12, \ 14, \ 105 \quad |L(S)| = 6 \]

For dynamic programming we would like

- a recursive expression w/ repeated subproblems
- an easy, fast way to use solved subproblems
\[ |L_{n-1}| = 2 \]

\[ |L_n| = 1 + \max_{\{\text{all } j \text{ s.t. } S[j] < S[n]\}} |L_j| \]

look at all \( L_j \) (\( j < n \))
\[ |L_n| = 1 + \max_{\{\text{all } j \text{ s.t. } S[j] < S[n]\}} |L_j| \]

Recursion:  
\[ L_n \]

BAD  
\[ L_{n-1} L_{n-2} \ldots L_1 \]

\[ L_{n-2} L_{n-3} \ldots \text{ etc} \]
\[ |L_n| = 1 + \max_{\text{all } j \text{ s.t. } S[j] < S[n]} |L_j| \]

Recursion: \( L_n \)

BAD

\( L_{n-1}, L_{n-2}, \ldots, L_1, L_{n-2}, L_{n-3}, \ldots, \text{ etc} \)

Dyn. Prog: Build solutions, "bottom up"

When it's time to solve \(|L_k|\) we have stored all \(|L_j| (j < k)\) in an array.
\[ 23, 3, 5, 18, 10, 101, 12, 14, 4 \]

\[ \frac{1}{IL_1 I} \]

\[ |L_n| = 1 + \max \{ \text{all } j \text{ st. } S[j] < S[n] \} |L_j| \]

Recursion: \( L_n \)

BAD

\( L_{n-1}, L_{n-2}, \ldots, L_1, \) etc

Dyn. Prog: Build solutions, "bottom up"

When it's time to solve \( |L_k| \) we have stored all \( |L_j| \) (\( j < k \)) in an array.
\[ \begin{align*} &23, 3, 5, 18, 10, 101, 12, 14, 4 \\ &1, 1 \\ &|L_2| \\ &|L_n| = 1 + \max_{\text{all } j \text{ s.t. } S[j] < S[n]} |L_j| \\ \end{align*} \]

Recursion: \( L_n \)

BAD

\[ L_{n-1}, L_{n-2}, \ldots, L_1, \ldots, L_{n-2}, L_{n-3}, \ldots \]

Dyn. Prog.: Build solutions, "bottom up"

When it's time to solve \(|L_k|\) we have stored all \(|L_j|\) (\(j < k\)) in an array.

etc
23, 3, 5, 18, 10, 101, 12, 14, 4

\[ |L_n| = 1 + \max_{\text{all } j \text{ s.t. } S[j] < S[n]} |L_j| \]

Recursion: \( L_n \)

BAD

\( L_{n-1}, L_{n-2}, \ldots, L_1, L_{n-2}, L_{n-3}, \ldots, \text{ etc} \)

Dyn. Prog.: Build solutions, "bottom up"

When it's time to solve \( |L_k| \) we have stored all \( |L_j| (j < k) \) in an array.
\[ |L_n| = 1 + \max_{\{\text{all } j \text{ s.t. } S[j] < S[n]\}} |L_j| \]

Recursion: \( L_n \)

BAD

\[ L_{n-1} L_{n-2} \ldots L_1 \]

\[ L_{n-2} L_{n-3} \ldots \]

etc

Dyn. Prog: Build solutions, "bottom up"

When it's time to solve \( |L_k| \) we have stored all \( |L_j| \) (j<k) in an array.
\[ |L_n| = 1 + \max \{ \text{all } j \text{ s.t. } S[j] < S[n] \} |L_j| \]

Recursion: \( L_n \)

BAD

\[ L_{n-1}, L_{n-2}, \ldots, L_1, \]

\[ L_{n-2}, L_{n-3}, \ldots, \text{etc} \]

Dyn. Prog: Build solutions, "bottom up"

When it's time to solve \( |L_k| \) we have stored all \( |L_j| \) \((j \neq k)\) in an array.
23, 3, 5, 18, 10, 101, 12, 14, 4
1 1 2 3 3 4

\[ |L_n| = 1 + \max_{\{ \text{all } j \text{ s.t. } S[j] < S[n] \}} |L_j| \]

Recursion: \( L_n \)
BAD

\( L_{n-1}, L_{n-2}, \ldots, L_1, \)

\( L_{n-2}, L_{n-3}, \ldots, \) etc

Dyn. Prog: Build solutions, "bottom up"

When it's time to solve \(|L_k|\) we have stored all \(|L_j|\) (\(j < k\)) in an array.
23, 3, 5, 18, 10, 101, 12, 14, 4
1 1 2 3 3 4 4

\[ |L_n| = 1 + \max \{ \text{all } j \text{ s.t. } S[j] < S[n] \} |L_j| \]

Recursion: \( L_n \) 

BAD

\[ L_{n-1} L_{n-2} \ldots L_1 \]

\[ L_{n-2} L_{n-3} \ldots \] etc

Dyn. Prog: Build solutions, "bottom up" 

When it's time to solve \( |L_k| \) we have stored all \( |L_j| \) (\( j < k \)) in an array.
23, 3, 5, 18, 10, 101, 12, 14, 4

$$|L_n| = 1 + \max_{\{\text{all } j \text{ s.t. } S[j] < S[n]\}} |L_j|$$

Recursion: $L_n$

BAD

$\ldots$ $L_{n-2}$ $L_{n-3}$ $L_1$

$\ldots$ $L_{n-2}$ $L_{n-3}$ etc

Dyn. Prog.: Build solutions, "bottom up"

When it's time to solve $|L_k|$ we have stored all $|L_j|$ ($j \leq k$) in an array.
$23, 3, 5, 18, 10, 101, 12, 14, 4$

$1, 1, 2, 3, 3, 4, 4, 5, 2 \rightarrow \text{Score may decrease}$

$$|L_n| = 1 + \max \{ \text{all } j \text{ s.t. } S[j] < S[n] \} |L_j|$$

**Recursion:** $L_n$

BAD

$L_{n-1}, L_{n-2}, \ldots, L_1$

$L_{n-2}, L_{n-3}, \ldots, \text{etc}$

**Dyn. Prog:** Build solutions, "bottom up"

When it's time to solve $|L_k|$ we have stored all $|L_j|$ ($j<k$) in an array.
23, 3, 5, 18, 10, 101, 12, 14, 4
1 1 2 3 3 4 4 5 2 → Score may decrease

\[ |L_n| = 1 + \max_{\{\text{all } j \text{ s.t. } S[j] < S[n]\}} |L_j| \]

Recursion: \( L_n \)

BAD

\( L_{n-1}, L_{n-2}, \ldots, L_1, L_{n-2}, L_{n-3}, \ldots, \text{etc} \)

Dyn. Prog: Build solutions, "bottom up"

When it's time to solve \(|L_k|\) we have stored all \(|L_j|\) (\(j < k\)) in an array.

time? space?
23, 3, 5, 18, 10, 101, 12, 14, 4 → Score may decrease

\[ |L_n| = 1 + \max_{\{ \text{all } j \text{ s.t. } S[j] < S[n] \}} |L_j| \]

Recursion: \( L_n \)

BAD

\( L_{n-1}, L_{n-2}, \ldots, L_1 \)

\( L_{n-2}, L_{n-3}, \ldots, \text{etc} \)

Dyn. Prog: Build solutions, "bottom up" When it's time to solve \( |L_k| \) we have stored all \( |L_j| (j < k) \) in an array.

\[ T(k) = \Theta(k) \]

\[ T(n) = \sum_{k=1}^{n} T(k) = \Theta(n^2) \]

Space = \( \Theta(n) \)
\[ T(n) = \Theta(n^2) \]
\[ \text{Space} = \Theta(n) \]

\[ |L_n| = 1 + \max_{\{\text{all } j \text{ s.t. } S[j] < S[n]\}} |L_j| \]
LONGEST COMMON SUBSEQUENCE
&
DYNAMIC PROGRAMMING
LONGEST COMMON SUBSEQUENCE & DYNAMIC PROGRAMMING

\[
\begin{align*}
X: & \quad n \quad A \quad B \quad C \quad B \quad D \quad A \quad B \\
Y: & \quad m \leq n \quad B \quad D \quad C \quad A \quad B \quad A
\end{align*}
\]
Longest Common Subsequence

&

Dynamic Programming

\[ X : \begin{array}{cccccc}
A & B & C & B & D & A & B \\
\end{array} \]

\[ Y : \begin{array}{cccccc}
B & D & C & A & B & A \\
\end{array} \]

\[ |LCS(X,Y)| = 4 \]
LONGEST COMMON SUBSEQUENCE
&
DYNAMIC PROGRAMMING

$X: \begin{cases} A & B & C & B & D & A & B \end{cases}
\end{array}
\begin{array}{l}
Y: \begin{cases} B & D & C & A & B & A \end{cases}
\end{array}$

$|LCS(X,Y)| = 4$
LONGEST COMMON SUBSEQUENCE
&
DYNAMIC PROGRAMMING

\[
X : \underbrace{A \ B \ C \ B \ D \ A \ B}_{m \leq n}
\]
\[
Y : \underbrace{B \ D \ C \ A \ B \ A}_{n}
\]

\[|\text{LCS}(x, y)| = 4\]
LONGEST COMMON SUBSEQUENCE

&

DYNAMIC PROGRAMMING

\[
\begin{align*}
X &: \{A, B, C, B, D, A, B\} \\
y &: \{B, D, C, A, B, A, B, A\}
\end{align*}
\]

\[|LCS(x, y)| = 4\]

Brute force to find LCS:
for every subsequence of \(y\)

\[\Theta(2^m)\]
Longest Common Subsequence & Dynamic Programming

\[ X: \underbrace{A B C B D A B}_{n} \]
\[ Y: \underbrace{B D C A B A}_{m \leq n} \]

\[ |LCS(x, y)| = 4 \]

Brute force to find LCS:
for every subsequence of \( Y \)
check if it exists in \( X \)

\[ \Theta(2^m) \leq O(n): \text{easy} \]
LONGEST COMMON SUBSEQUENCE

&

DYNAMIC PROGRAMMING

\[
X: \underbrace{A B C B D A B}_{n} \\
Y: \underbrace{B D C A B A}_{m \leq n}
\]

\[|\text{LCS}(x, y)| = 4\]

Brute force to find LCS:

- for every subsequence of \( Y \), check if it exists in \( X \)

\[\Theta(2^m)\]

\[O(n \cdot 2^m)\]
Finding \( |LCS| \)

\[ c(i,j) = |LCS(x[1...i], y[1...j])| \]
Finding \(|\text{LCS}|\)

\[ c(i, j) = |\text{LCS}(x[1...i], y[1...j])| = \begin{cases} 
C(i-1, j-1) + 1 & \text{if } x[i] = y[j] 
\end{cases} \]
Finding $|LCS|$:

$$c(i,j) = |LCS(x[1...i], y[1...j])| = \begin{cases} 
C(i-1,j-1) + 1 & \text{if } x[i] = y[j] 
\end{cases}$$
Finding $|LCS|$ 

$$c(i,j) = |LCS(X[1...i], Y[1...j])| = \begin{cases} 
c(i-1, j-1) + 1 & \text{if } x[i] = y[j] 
\end{cases}$$
Finding $|\text{LCS}|$

$c(i, j) = |\text{LCS}(x[1...i], y[1...j])| = \begin{cases} C(i-1, j-1) + 1 & \text{if } x[i] = y[j] \\ \end{cases}$

Slide last match over: just as good
Finding $|LCS|$:

$$c(i,j) = |LCS(x[1...i], y[1...j])| = \begin{cases} 
C(i-1,j-1) + 1 & \text{if } x[i] = y[j] \\
\max \{c(i,j-1), c(i-1,j)\} & \text{otherwise}
\end{cases}$$
Finding $|LCS|$ 

$$c(i,j) = |LCS(x[1...i], y[1...j])| = \begin{cases} 
  c(i-1, j-1) + 1 & \text{if } x[i] = y[j] \\
  \max\{c(i,j-1), c(i-1,j)\} & \text{otherwise}
\end{cases}$$

$LCS(x[1...i], y[1...j])$ cannot use both $x[i]$ and $y[j]$. 

A B C C D A B
/ / / / / / / / blocked
B D C A B A

A B C C D A B
/ / / / / / / / blocked
B D C A B A
Finding $|\text{LCS}|$

$$c(i,j) = |\text{LCS}(X[1...i], Y[1...j])| = \begin{cases} 
C(i-1,j-1) + 1 & \text{if } X[i] = Y[j] \\
\max\{c(i,j-1), c(i-1,j)\} & \text{otherwise}
\end{cases}$$

$LCS(X[1...i], Y[1...j])$
cannot use both $X[i]$ and $Y[j]$

\[\begin{align*}
A & B & C & C & D & A & B \\
\| & | & | & | & | & | & \\
B & D & C & A & B & A & \times
\end{align*}\]

Hide each

\[\begin{align*}
A & B & C & C & D & A & \times \\
\| & | & | & | & | & | & \\
B & D & C & A & B & A & \times
\end{align*}\]
Finding \( |\text{LCS}| \)

\[ c(i,j) = |\text{LCS}(X[1...i], Y[1...j])| = \begin{cases} 
C(i-1,j-1) + 1 & \text{if } X[i] = Y[j] \\
\max\{c(i,j-1), c(i-1,j)\} & \text{otherwise}
\end{cases} \]

\( \text{LCS}(X[1...i], Y[1...j]) \)
cannot use both \( X[i] \) and \( Y[j] \)

```
A B C C D A B
/ | | |
B D C A B (A
```

```
A B C C D A B
/ | | |
B D C A B (B
```

Hide each and take best result

```
A B C C D A B
/ | | |
B D C A B A
```

```
A B C C D A B
/ | | |
B D C A B A
```

A B C C D A B
/ | | |
B D C A B A

\[ c(i,j) = |LCS(x[1...i], y[1...j])| = \begin{cases} 
  c(i-1, j-1) + 1 & \text{if } x[i] = y[j] \\
  \max\{c(i,j-1), c(i-1,j)\} & \text{otherwise}
\end{cases} \]
\[ c(i,j) = |LCS(X[1...i], Y[1...j])| = \begin{cases} 
C(i-1,j-1) + 1 & \text{if } X[i] = Y[j] \\
\max \{C(i,j-1), C(i-1,j)\} & \text{otherwise}
\end{cases} \]

"Optimal substructure": optimal solutions of subproblems are part of the original problem solution.
\[ c(i,j) = \left| \text{LCS}(X[1\ldots i], Y[1\ldots j]) \right| = \begin{cases} c(i-1, j-1) + 1 & \text{if } X[i] = Y[j] \\ \max \{ c(i,j-1), c(i-1,j) \} & \text{otherwise} \end{cases} \]

"Optimal substructure": optimal solutions of subproblems are part of the original problem solution.

\[ \text{LCS}(X, Y, i, j) \]

return \( c_{ij} \)
\[ c(i, j) = \left| \text{LCS}(x[1...i], y[1...j]) \right| = \begin{cases} c(i-1, j-1) + 1 & \text{if } x[i] = y[j] \\ \max\{c(i, j-1), c(i-1, j)\} & \text{otherwise} \end{cases} \]

"Optimal substructure": optimal solutions of subproblems are part of the original problem solution.

\[
\text{LCS}(x, y, i, j)
\]

if \( x[i] = y[j] \) then \( c_{ij} \leftarrow \text{LCS}(x, y, i-1, j-1) + 1 \)

return \( c_{ij} \)
\[ c(i,j) = \begin{cases} 
\text{LCS}(X[1\ldots i], Y[1\ldots j]) & \text{if } x[i] = y[j] \\
C(i-1, j-1) + 1 & \text{otherwise}
\end{cases} \]

"Optimal substructure": optimal solutions of subproblems are part of the original problem solution.

\[
\text{LCS}(X, Y, i, j) \quad \text{\textbackslash ignoring base case: if } i \text{ or } j = 0 \text{ then } c_{ij} = 0 \\
\text{if } x_i = y_j \text{ then } c_{ij} \leftarrow \text{LCS}(X, Y, i-1, j-1) + 1 \\
\text{else } c_{ij} \leftarrow \max\{\text{LCS}(X, Y, i, j-1), \text{LCS}(X, Y, i-1, j)\} \\
\text{return } c_{ij}
\]
LCS(X, Y, i, j)
  if \( X_i = Y_j \) then \( c_{ij} \leftarrow LCS(X, Y, i-1, j-1) + 1 \)
  else \( c_{ij} \leftarrow \max\{LCS(X, Y, i, j-1), LCS(X, Y, i-1, j)\} \)
  return \( c_{ij} \)
LCS(X, Y, i, j)
if \( X_i = Y_j \) then \( c_{ij} \leftarrow LCS(X, Y, i-1, j-1) + 1 \)
else \( c_{ij} \leftarrow \max\{LCS(X, Y, i, j-1), LCS(X, Y, i-1, j)\} \)
return \( c_{ij} \)

\[
\text{worst case: always get } X_i \neq Y_j
\]

For example: \( n=7, m=6 \)

```
LCS(X, Y, 7, 6)

6, 6

6, 5

6, 6

5, 5

4, 6

4, 5

\[
\text{ex: } n=7, m=6
\]

6, 3

6, 3

LCS(X, Y, 7, 6)
```
LCS(X, Y, i, j)

if \( X_i = Y_j \) then \( c_{ij} \leftarrow LCS(X, Y, i-1, j-1) + 1 \)

else \( c_{ij} \leftarrow \max\{LCS(X, Y, i, j-1), LCS(X, Y, i-1, j)\} \)

return \( c_{ij} \)

\[
\text{worst case: always get } X_i \neq Y_j\]

\[
\text{ex: } n=7, m=6
\]

\[
\text{# full levels} \quad > \min\{m, n\}
\]

\[
\text{work} = \Omega(2^n)
\]

if \( m = n \)
\[ \text{LCS}(X, Y, i, j) \]

- if \( X_i = Y_j \) then \( c_{ij} \leftarrow \text{LCS}(X, Y, i-1, j-1) + 1 \)
- else \( c_{ij} \leftarrow \max\{\text{LCS}(X, Y, i, j-1), \text{LCS}(X, Y, i-1, j)\} \)

- return \( c_{ij} \)

\[ \text{LCS}(X, Y, 7, 6) \]

- worst case: always get \( X_i \neq Y_j \)
- ex: \( n=7, m=6 \)
- \#full levels \( > \min\{m, n\} \)
- work = \( \Omega(2^n) \)

lots of repeated work

\( \Leftrightarrow \) \#distinct subproblems = ?
LCS(X,Y,i,j)
if \(X_i = Y_j\) then \(c_{ij} \leftarrow \text{LCS}(X,Y,i-1,j-1) + 1\)
else \(c_{ij} \leftarrow \max\{\text{LCS}(X,Y,i,j-1), \text{LCS}(X,Y,i-1,j)\}\)
return \(c_{ij}\)

\[\text{LCS}(X,Y,7,6)\]
\[6,6 \quad 7,5\]
\[5,6 \quad 6,5 \quad 6,5 \quad 7,4\]
\[4,6 \quad 5,5 \quad 5,5 \quad 5,5 \quad 6,4 \quad 6,4 \quad 7,3 \quad 6,3,6,3\]

\{ worst case: always get \(X_i \neq Y_j\) \}
\[\text{ex: } n=7, m=6\]

- Full levels: \(\min\{m,n\}\)
- Work: \(\Omega(2^n)\)
- If \(mn\)

\(\Rightarrow \) #distinct subproblems = \(m \cdot n\)

lots of repeated work
LCS($X,Y,i,j$)
if $X_i = Y_j$ then $c_{ij} \leftarrow LCS(X,Y,i-1,j-1) + 1$
else $c_{ij} \leftarrow \max\{LCS(X,Y,i,j-1), LCS(X,Y,i-1,j)\}$
return $c_{ij}$

worst case: always get $X_i \neq Y_j$

ex: $n=7, m=6$

$\text{LCS}(X,Y,7,6)$

Repeated subproblems + optimal substructure $\Rightarrow$ try dynamic programming

$\#\text{full levels} > \min\{m,n\}$

work = $\Omega(2^n)$ if $mn$

lots of repeated work

$\Rightarrow \#\text{distinct subproblems} = m \cdot n$
\[ LCS(X, Y, i, j) \]
\[
\text{if } X_i = Y_j \text{ then } c_{ij} \leftarrow LCS(X, Y, i-1, j-1) + 1
\]
\[
\text{else } c_{ij} \leftarrow \max\{LCS(X, Y, i, j-1), LCS(X, Y, i-1, j)\}
\]
\text{return } c_{ij}
LCS(X, Y, i, j)
  if X_i = Y_j then c_{ij} ← LCS(X, Y, i−1, j−1) + 1
  else c_{ij} ← \text{max}\{LCS(X, Y, i, j−1), LCS(X, Y, i−1, j)\}
  return c_{ij}

**Memoization**

Make "memos" of solutions (to subproblems)
\[
\text{LCS}(X, Y, i, j)
\]
\[
\text{if } X_i = Y_j \text{ then } c_{ij} \leftarrow \text{LCS}(X, Y, i-1, j-1) + 1
\]
\[
\text{else } c_{ij} \leftarrow \max\{\text{LCS}(X, Y, i, j-1), \text{LCS}(X, Y, i-1, j)\}
\]
\[
\text{return } c_{ij}
\]

Let \( c[1...m, 1...n] \) be a \( m \times n \) table of -1's.

\text{Memoization}

Make "memos" of solutions (to subproblems)
\[
\text{LCS}(X,Y,i,j)
\]
\[
\text{if } X_i = Y_j \text{ then } c_{ij} \leftarrow \text{LCS}(X,Y,i-1,j-1) + 1
\]
\[
\text{else } c_{ij} \leftarrow \max\{\text{LCS}(X,Y,i,j-1), \text{LCS}(X,Y,i-1,j)\}
\]
\[
\text{return } c_{ij}
\]

\textbf{Memoization}

\text{Make "memos" of solutions}
\text{(to subproblems)}

Let \(c[1..m,1..n]\) be a \(m \times n\) table of \(-1\)'s.

Whenever we need to know \(c_{ij}\)

If it's the first time \ldots \ldots then calculate it
LCS(X,Y,i,j)
if Xᵢ = Yⱼ then cᵢⱼ ← LCS(X,Y,i-1,j-1) + 1
else cᵢⱼ ← max{LCS(X,Y,i,j-1), LCS(X,Y,i-1,j)}
return cᵢⱼ

Let C[1...m,1...n] be a m×n table of -1’s.
whenever we need to know cᵢⱼ
if it’s the first time (C[i,j] = -1) then calculate it
else look it up

**Memoization**
Make “memos” of solutions (to subproblems)
LCS(X, Y, i, j)
  if $X_i = Y_j$ then $c_{ij} \leftarrow \text{LCS}(X, Y, i-1, j-1) + 1$
  else $c_{ij} \leftarrow \max\{\text{LCS}(X, Y, i, j-1), \text{LCS}(X, Y, i-1, j)\}$
  return $c_{ij}$

Let $C[1...m, 1...n]$ be a $m \times n$ table of -1’s. Whenever we need to know $c_{ij}$
  if it’s the first time ($C[i,j] = -1$) then calculate it
  else look it up

Memoization
Make “memos” of solutions (to subproblems)

if $\min(i, j) = 0$ then return 0
\[
\text{LCS}(X, Y, i, j) \\
\text{if } X_i = Y_j \text{ then } c_{ij} \leftarrow \text{LCS}(X, Y, i-1, j-1) + 1 \\
\text{else } c_{ij} \leftarrow \max\{\text{LCS}(X, Y, i, j-1), \text{LCS}(X, Y, i-1, j)\} \\
\text{return } c_{ij}
\]

\textbf{Memoization} \\
Make "memos" of solutions (to subproblems)

Let \(C[1...m, 1...n]\) be a \(m \times n\) table of -1's. Whenever we need to know \(c_{ij}\), if it's the first time \(C[i, j] = -1\) then calculate it else look it up

\[
\text{LCS}(X, Y, i, j) \\
\text{if } \min{i, j} = 0 \text{ then return 0} \\
\text{if } C[i, j] = -1 \text{ then } \backslash \text{ first time}
\]
LCS(X, Y, i, j)
if $X_i = Y_j$ then $C_{i,j} \leftarrow LCS(X, Y, i-1, j-1) + 1$
else $C_{i,j} \leftarrow \max \{LCS(X, Y, i, j-1), LCS(X, Y, i-1, j)\}$
return $C_{i,j}$

Memoization
Make "memos" of solutions (to subproblems)

Let $C[1...m, 1...n]$ be a $m \times n$ table of -1's.
whenever we need to know $C_{i,j}$
if it's the first time ($C[i,j] = -1$) then calculate it
else look it up

LCS(X, Y, i, j)
if $\min \{i, j\} = 0$ then return 0
if $C[i, j] = -1$ then \(\backslash\) first time
if $X_i = Y_j$ then $C[i, j] \leftarrow LCS(X, Y, i-1, j-1) + 1$
else $C[i, j] \leftarrow \max \{LCS(X, Y, i, j-1), LCS(X, Y, i-1, j)\}$
LCS(X, Y, i, j)
  if \(X_i = Y_j\) then \(c_{ij} \leftarrow \text{LCS}(X, Y, i-1, j-1) + 1\)
  else \(c_{ij} \leftarrow \max\{\text{LCS}(X, Y, i, j-1), \text{LCS}(X, Y, i-1, j)\}\)
  return \(c_{ij}\)

**Memoization**

Make "memos" of solutions (to subproblems)

Let \(C[1...m, 1...n]\) be a \(m \times n\) table of -1's.
whenever we need to know \(c_{ij}\)
if it's the first time \((C[i, j] = -1)\) then calculate it
else look it up

LCS(X, Y, i, j)
if \(\min(i, j) = 0\) then return 0
if \(C[i, j] = -1\) then \(\parallel\) first time
  if \(X_i = Y_j\) then \(C[i, j] \leftarrow \text{LCS}(X, Y, i-1, j-1) + 1\)
  else \(C[i, j] \leftarrow \max\{\text{LCS}(X, Y, i, j-1), \text{LCS}(X, Y, i-1, j)\}\)
return \(C[i, j]\) \(\parallel\) look up
LCS(X, Y, i, j)
if \(X_i = Y_j\) then \(c_{ij} \leftarrow \text{LCS}(X, Y, i-1, j-1) + 1\)
else \(c_{ij} \leftarrow \max\{\text{LCS}(X, Y, i, j-1), \text{LCS}(X, Y, i-1, j)\}\)
return \(c_{ij}\)

**Memoization**

Make "memos" of solutions (to subproblems)

\[\text{Let } C[1\ldots m, 1\ldots n] \text{ be a } m \times n \text{ table of } -1\text{'s.}\]

whenever we need to know \(c_{ij}\)
if it's the first time \((C[i, j] = -1)\) then calculate it
else look it up
LCS(X, Y, i, j)
  if X_i = Y_j then c_{i,j} ← LCS(X, Y, i-1, j-1) + 1
  else c_{i,j} ← max\{LCS(X, Y, i, j-1), LCS(X, Y, i-1, j)\}
  return c_{i,j}

\[\Theta(mn) \text{ time \& space}\]

\[\text{LCS}(X, Y, i, j)\]
  if \(i = 0\) or \(j = 0\) then return 0
  if \(C[i, j] = -1\) then \(\ll\) first time
    if \(X_i = Y_j\) then \(C[i, j] ← LCS(X, Y, i-1, j-1) + 1\)
    else \(C[i, j] ← \max\{LCS(X, Y, i, j-1), LCS(X, Y, i-1, j)\}\)
  return \(C[i, j]\) \(\ll\) look up

\[\text{Memoization}\]
  Make "memos" of solutions (to subproblems)

Let \(C[1...m, 1...n]\) be a \(m \times n\) table of -1's. Whenever we need to know \(c_{i,j}\)
  if it's the first time \((C[i, j] = -1)\) then calculate it
  else look it up
Memoization
Make "memos" of solutions (to subproblems)

Top-down

LCS(X, Y, 7, 6)

6, 6

5, 6

4, 6

5, 5

5, 5

5, 5

6, 4

5, 5

6, 4

6, 4

7, 3

6, 3

6, 3
Memoization
Make "memos" of solutions (to subproblems)

LCS(X, Y, 7, 6)

6, 6  7, 5

5, 6  6, 5  6, 5  7, 4

4, 6  5, 5  5, 5  6, 4  5, 5  6, 4  6, 4  7, 3

4, 6  5, 5  6, 4  6, 4  7, 3

6, 3  6, 3
Memoization
Make "memos" of solutions (to subproblems)

Top-down
Memoization
Make "memos" of solutions (to subproblems)

Top-down

LCS(x, y, 7, 6)

6, 6
5, 6
4, 6

5, 5
6, 5
5, 5
6, 4
5, 5
6, 4
6, 4
6, 3
7, 5
7, 4
7, 3
7, 3
7, 6
Dynamic Programming
Dynamic Programming  
bottom-up

A B C B D A B

• • • • • • • • • •

B •
D •
C •
A •
B •
A •

base cases
**Dynamic Programming**

```
A B C B D A B
D
C
A
B
A
```

green # = max of {above, left}

when letters in column & row of #
don't match
**Dynamic Programming**

```
   A B C B D A B
   o o o o o o o o
   B 1 o o o o o o
   D o o o o o o o
   C o o o o o o o
   A o o o o o o o
   B o o o o o o o
   A o o o o o o o
```

Red # : $1 + \text{diag}#$

- when letters in column & row of # match

Green # : $\max\{\text{above, left}\}$

- when letters in column & row of # don't match
Dynamic Programming

$$\begin{array}{cccccc}
A & B & C & B & D & A & B \\
\circ & \circ & \circ & \circ & \circ & \circ & \circ \\
B & 1 & 1 & \circ & \circ & \circ & \circ \\
D & \circ & \circ & \circ & \circ & \circ & \circ \\
C & \circ & \circ & \circ & \circ & \circ & \circ \\
A & \circ & \circ & \circ & \circ & \circ & \circ \\
B & \circ & \circ & \circ & \circ & \circ & \circ \\
A & \circ & \circ & \circ & \circ & \circ & \circ \\
\end{array}$$

red \#: \ 1 + \text{diag}\# \\
when \ letters \ in \ column \ & \ row \ of \ # \ match

green \#: \ \max \ of \ {\text{above, left}} \\
when \ letters \ in \ column \ & \ row \ of \ # \ don't \ match
**Dynamic Programming**

- A B C B D A B
- 0 0 0 0 0 0 0 0

```
  B 0 0 1 1 1
  D 0
  C 0
  A 0
  B 0
  A 0
```

- **Red #**: $1 + \text{diag #}$
  - when letters in column & row of # match
- **Green #**: max of \{above, left\}
  - when letters in column & row of # don't match
Dynamic Programming

red # : 1 + diag #
when letters in column & row of # match

green # : max of {above, left} when letters in column & row of # don't match
Dynamic Programming

A B C B D A B

red \#: 1 + diag\# when letters in column & row of \# match

green \#: max of \{above, left\} when letters in column & row of \# don't match
**Dynamic Programming**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Red #**:

1. When letters in column & row of # match
2. Max of {above, left} when letters in column & row of # don't match
Dynamic Programming

red \# \text{ : } 1 + \text{ diag}\#

when letters in column \& row of \# match

green \# \text{ : } \max \{ \text{above, left} \}
when letters in column \& row of \# don't match
Dynamic Programming

\[
\begin{array}{cccccc}
A & B & C & B & D & A & B \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
B & 0 & 0 & 1 & 1 & 1 & 1 \\
D & 0 & 0 & 1 & 1 & 1 & 1 \\
C & 0 & 0 & 1 & 2 & 2 & 2 \\
A & 0 & 1 & 1 & 1 & 1 & 1 \\
B & 0 & 1 & 1 & 1 & 1 & 1 \\
A & 0 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

red # : 1 + diag #
when letters in column & row of # match

green # : max of \{above, left\}
when letters in column & row of # don't match
Dynamic Programming

A B C B D A B
0 0 0 0 0 0 0 0
B 0 0 1 1 1 1 1 1
D 0 0 1 1 1 2 2 2
C 0 0 1 2 2
A 0 1 1 2
B 0 1 2
A 0 1

red # : 1 + diag#
when letters in column & row of # match

green # : max of {above, left}
when letters in column & row of # don't match
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

**Red #:** $1 + \text{diag}#$

When letters in column & row of # match

**Green #:** $\max$ of {above, left}#

When letters in column & row of # don't match
Dynamic Programming

**Red #**: $1 + \text{diag}

When letters in column & row of # match

**Green #**: $\max\{\text{above, left}\}$

When letters in column & row of # don't match

Trace from $C_{mn}$ to $C_{11}$ to get LCS
Dynamic Programming

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Red #: 1 + diag#
when letters in column & row of # match

Green #: max of {above, left}
when letters in column & row of # don't match

Trace from C_{mn} to C_{11} to get LCS
follow mandatory paths;
optional branches: multiple solutions
**Dynamic Programming**

- **Red #**: $1 + \text{diag#}
  - When letters in column & row of # match
- **Green #**: max of \{above, left\}
  - When letters in column & row of # don't match

Trace from $C_{mn}$ to $C_{11}$ to get LCS

- Follow mandatory paths;
- Optional branches: multiple solutions

LCS Examples:
- BDAB
- BCAB
- BCBA
Dynamic Programming

When letters in column & row of # match

green #: \text{max of \{above, left\}}

when letters in column & row of # don't match

Trace from C_{mn} to C_{11} to get LCS

follow mandatory paths;
optimal branches: multiple solutions

\Theta(mn) time & space (+1 trace)
Dynamic Programming

A B C B D A B

D C A B A

red # : 1 + diag#
when letters in column & row of # match

green # : max of {above, left}^
when letters in column & row of # don't match

Trace from C_{mn} to C_{11} to get LCS
follow mandatory paths;
optional branches : multiple solutions

Θ(mn) time & space (+1 trace)

Save space: \min\{m, n\}
Dynamic Programming

\[
\begin{array}{cccccc}
A & B & C & B & D & A & B \\
B & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 1 \\
D & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\
C & \\
A & \\
B & \\
A & \\
\end{array}
\]

**red #**: \(1 + \text{diag}^#\)
- when letters in column & row of # match

**green #**: \(\max\{\text{above, left}\}\)
- when letters in column & row of # don't match

Trace from \(C_{mn}\) to \(C_{11}\) to get LCS
- follow mandatory paths
- optional branches: multiple solutions

\(\Theta(mn)\) time & space (+1 trace)

Save space: \(\min\{m, n\}\)
Dynamic Programming

A B C B D A B

B
D 0 0 1 1 1 2 2 2
C 0 0 1 2 2 2 2 2
A 0 0 0 0 0 0 0 0

red # : 1 + diag#
when letters in column & row of # match

green # : max of {above, left}#
when letters in column & row of # don't match

Trace from Cmn to C11 to get LCS
follow mandatory paths;
optimal branches: multiple solutions
Θ(mn) time & space (+1 trace)
Save space: min{m,n}
Dynamic Programming

A B C B D A B

B
D
C
A

etc

\[ \begin{array}{ccccccc}
& & & & & & \\
B & 0 & 1 & 2 & 2 & 3 & 3 & 3 & 4 \\
A & 0 & 1 & 2 & 2 & 3 & 3 & 4 & \text{4} \\
\end{array} \]

red \# : 1 + diag\# 

when letters in column & row of \# match

green \# : max of \{above, left\} 

when letters in column & row of \# don't match

Trace from \( C_{mn} \) to \( C_{11} \) to get LCS 

→ follow mandatory paths; 
optional branches: multiple solutions

\( \Theta(mn) \) time & space (+1 trace)

get \( |\text{LCS}| \) but not LCS 

Save space: \( \min\{m,n\} \)