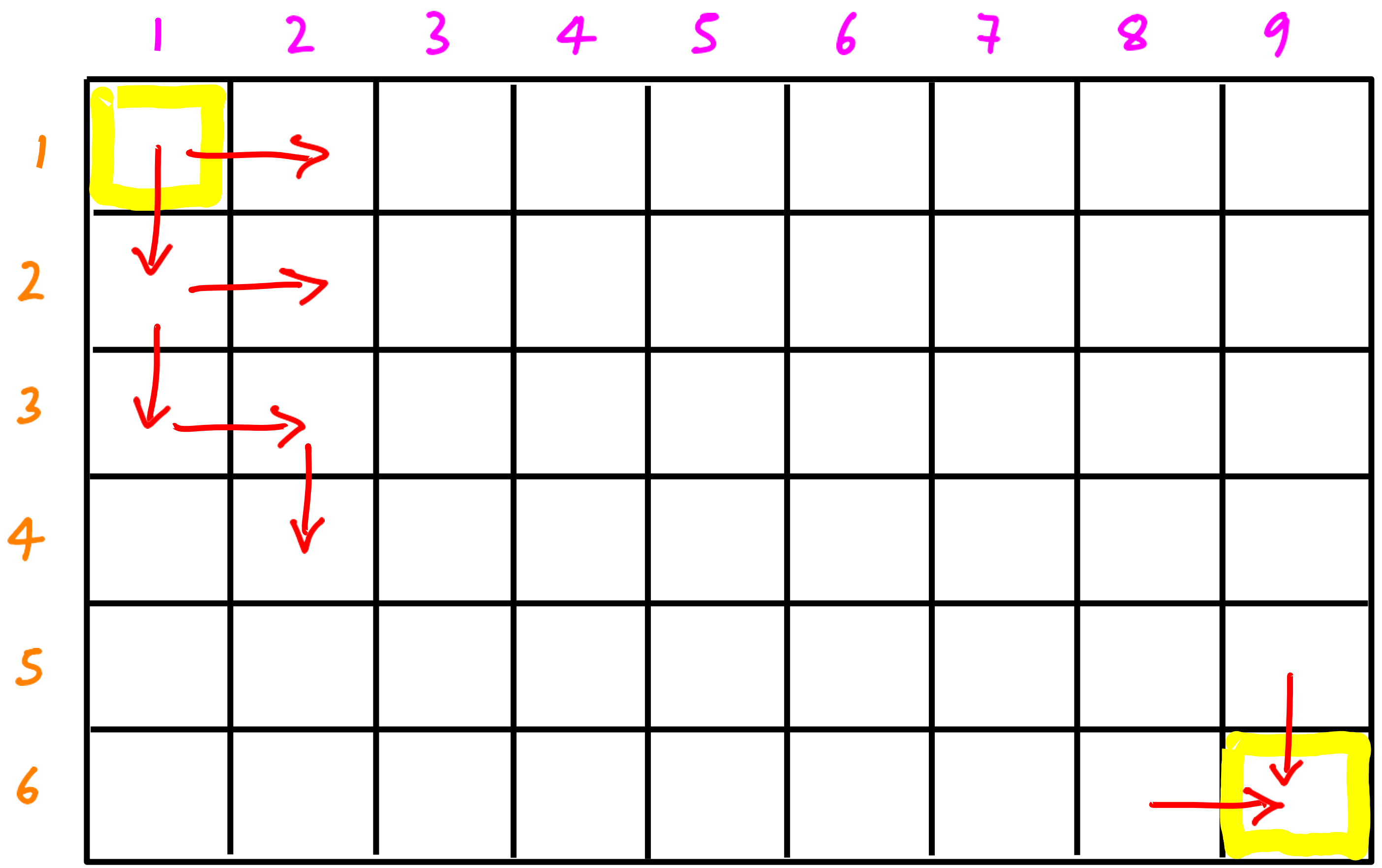
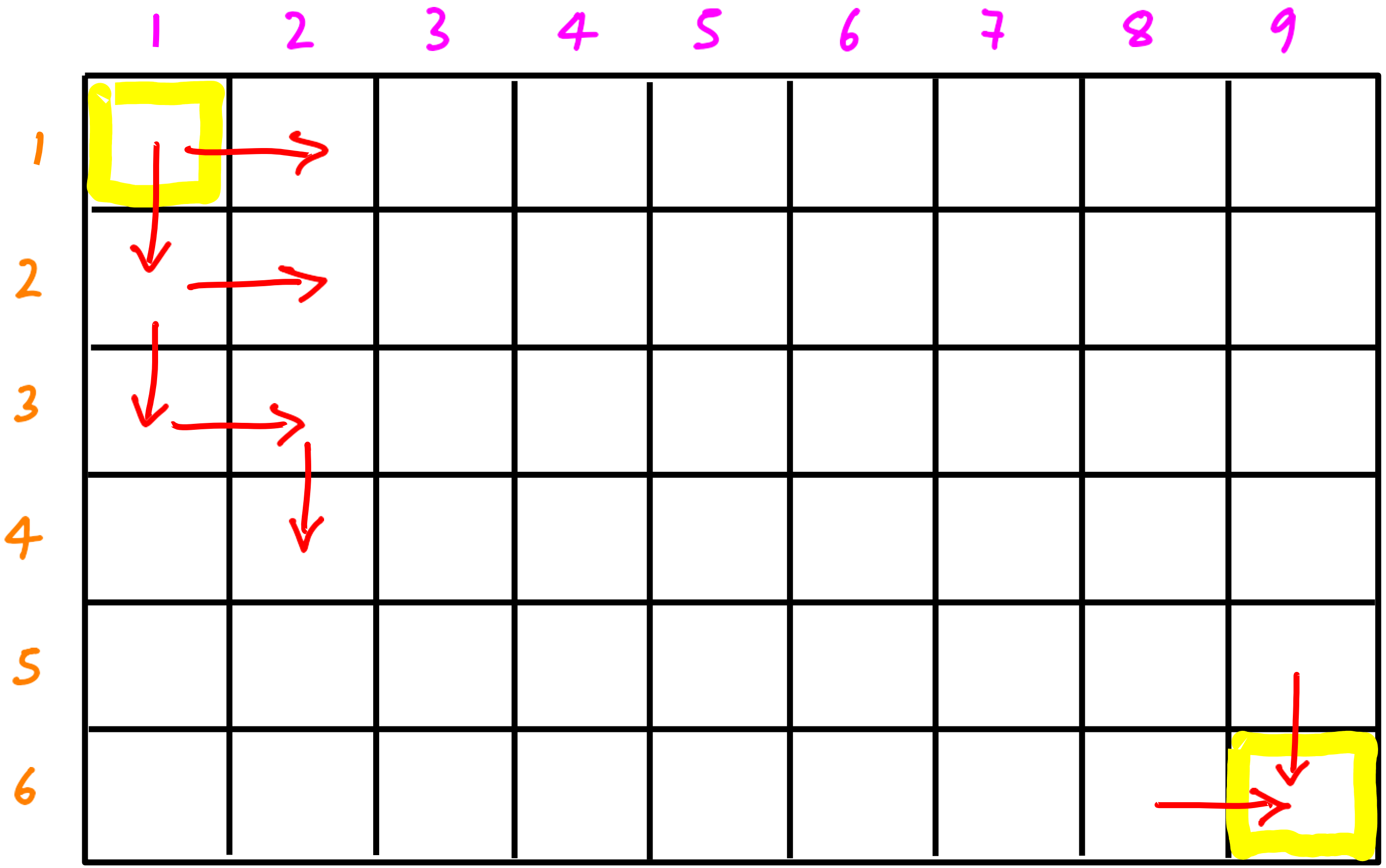
Starting at top-left of $n \times m$ grid, moving only down or right, how many ways to reach bottom-right?

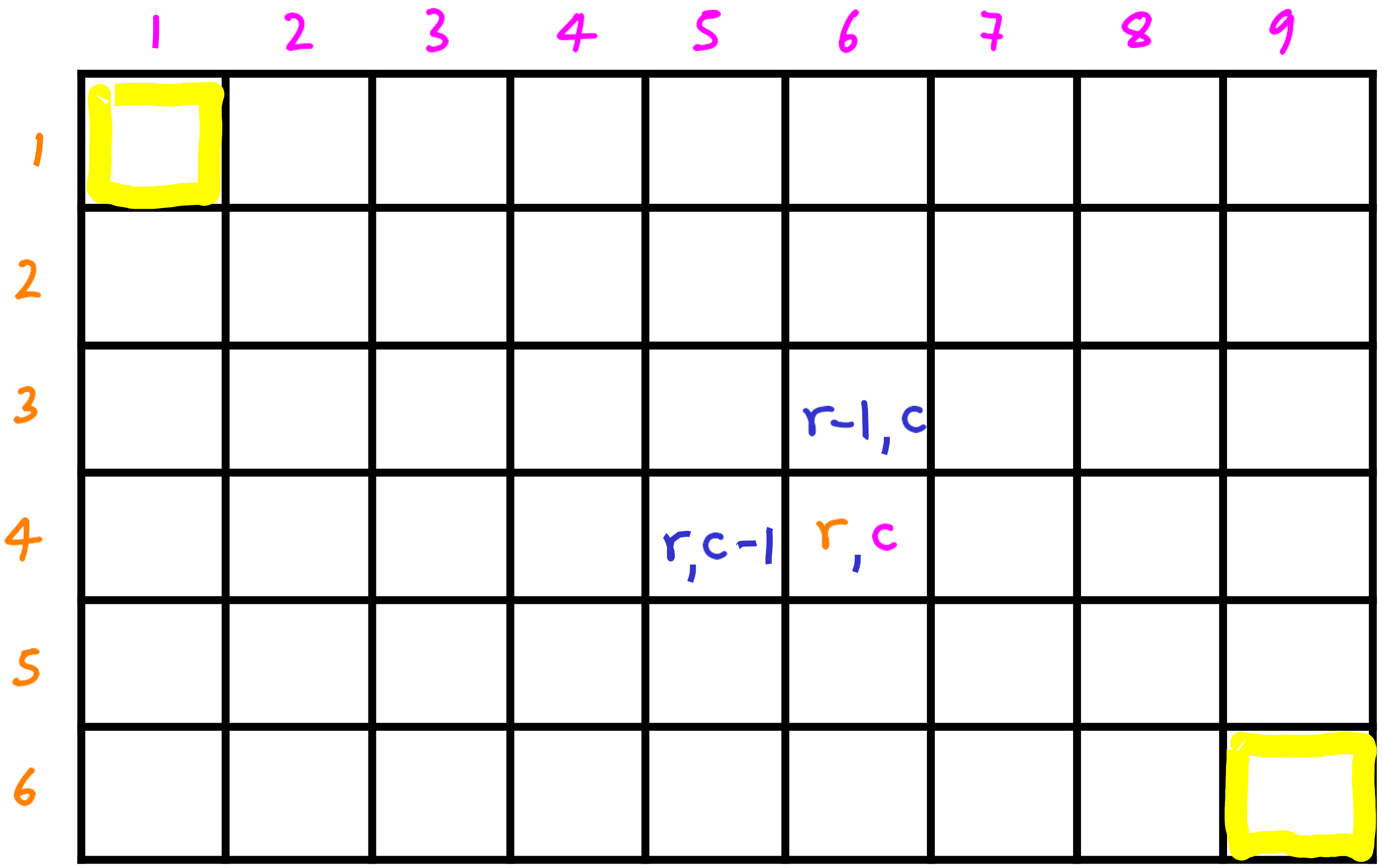


Starting at top-left of $n \times m$ grid, moving only down or right, how many ways to reach bottom-right?

Recursive form?



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



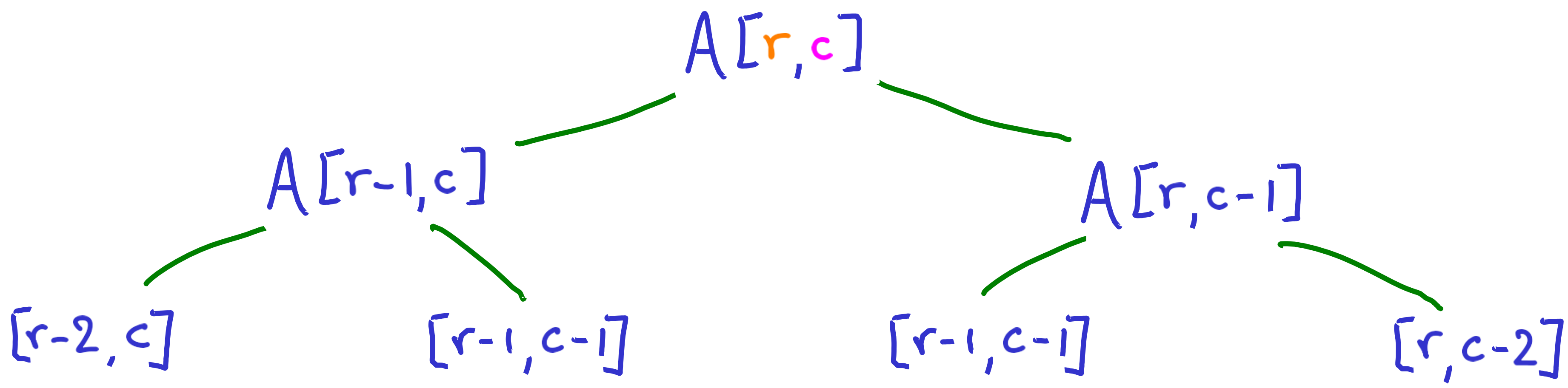
$$A[r,c] = A[r-1,c] + A[r,c-1]$$

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

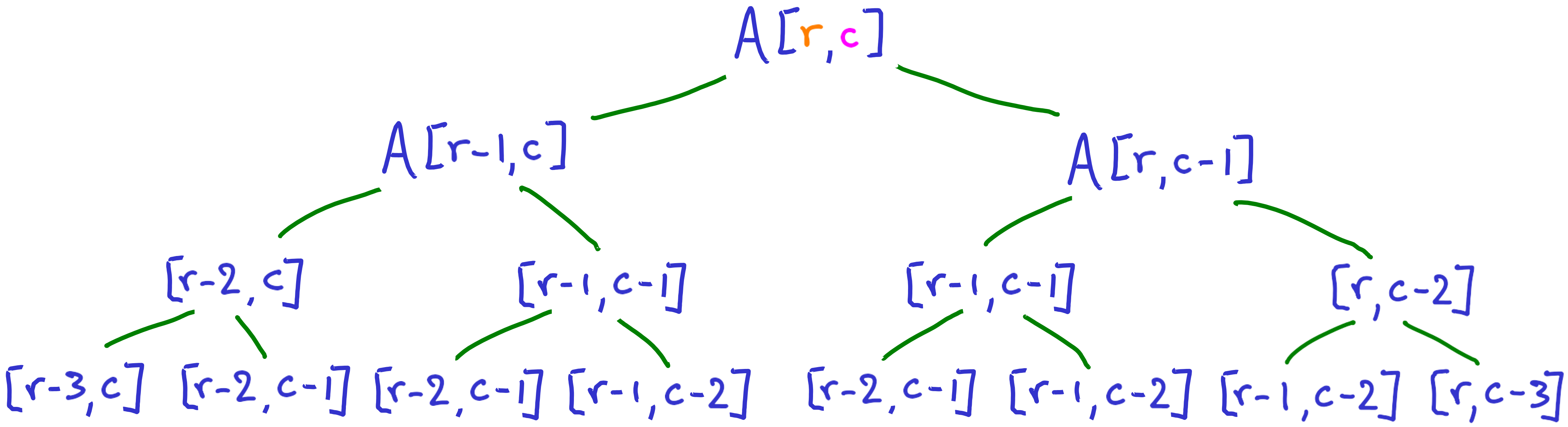
$$A[r,c]$$

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

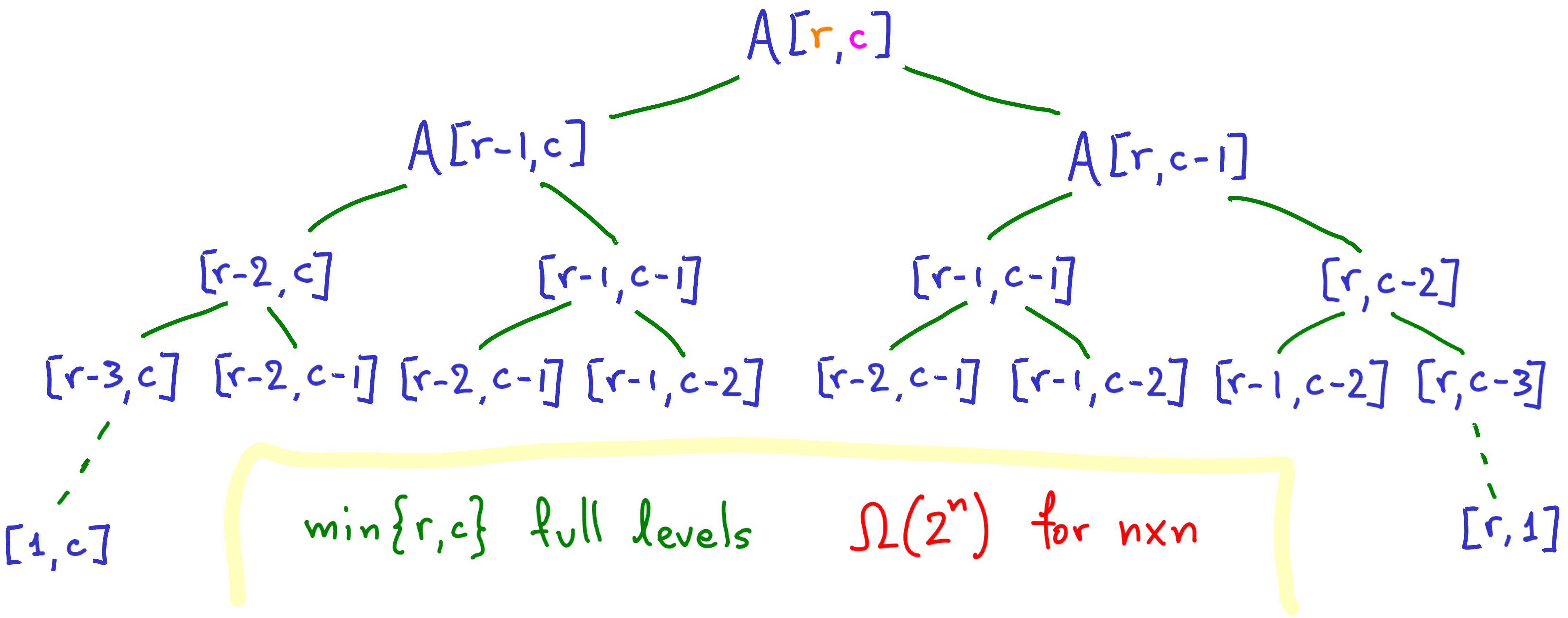
Starting at top-left of $n \times m$ grid, moving only down or right, how many ways to reach bottom-right?



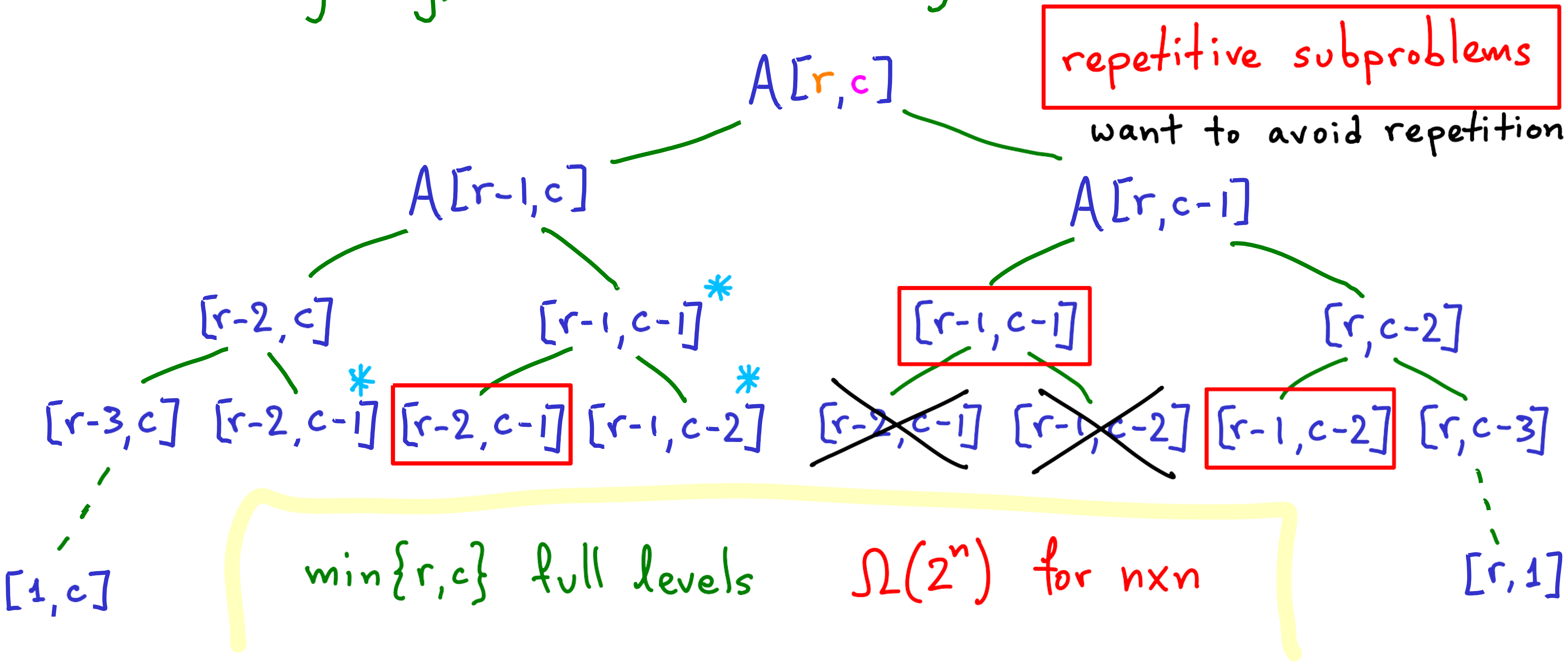
Starting at top-left of $n \times m$ grid, moving only down or right, how many ways to reach bottom-right?



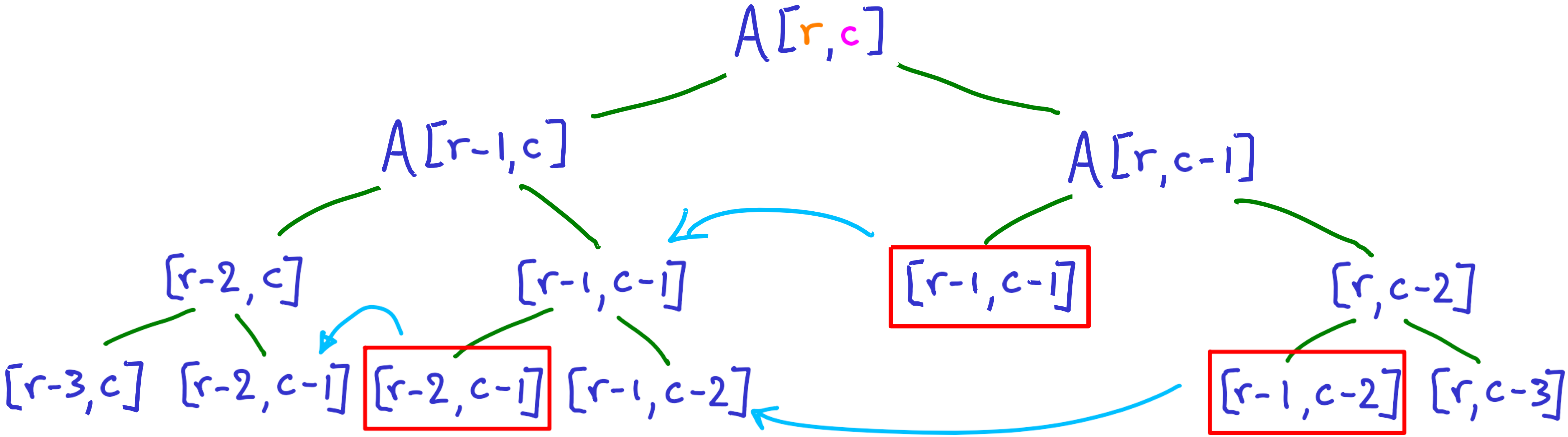
Starting at top-left of $n \times m$ grid, moving only down or right, how many ways to reach bottom-right?



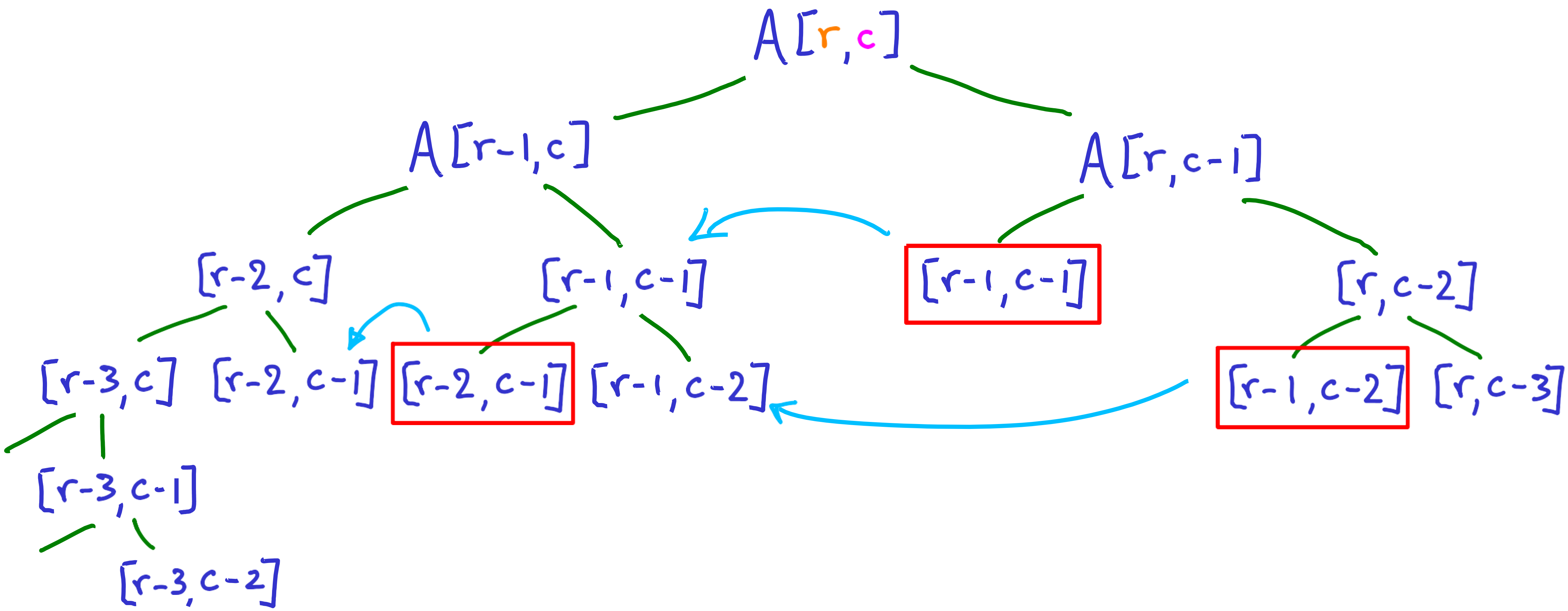
Starting at top-left of $n \times m$ grid, moving only down or right, how many ways to reach bottom-right?



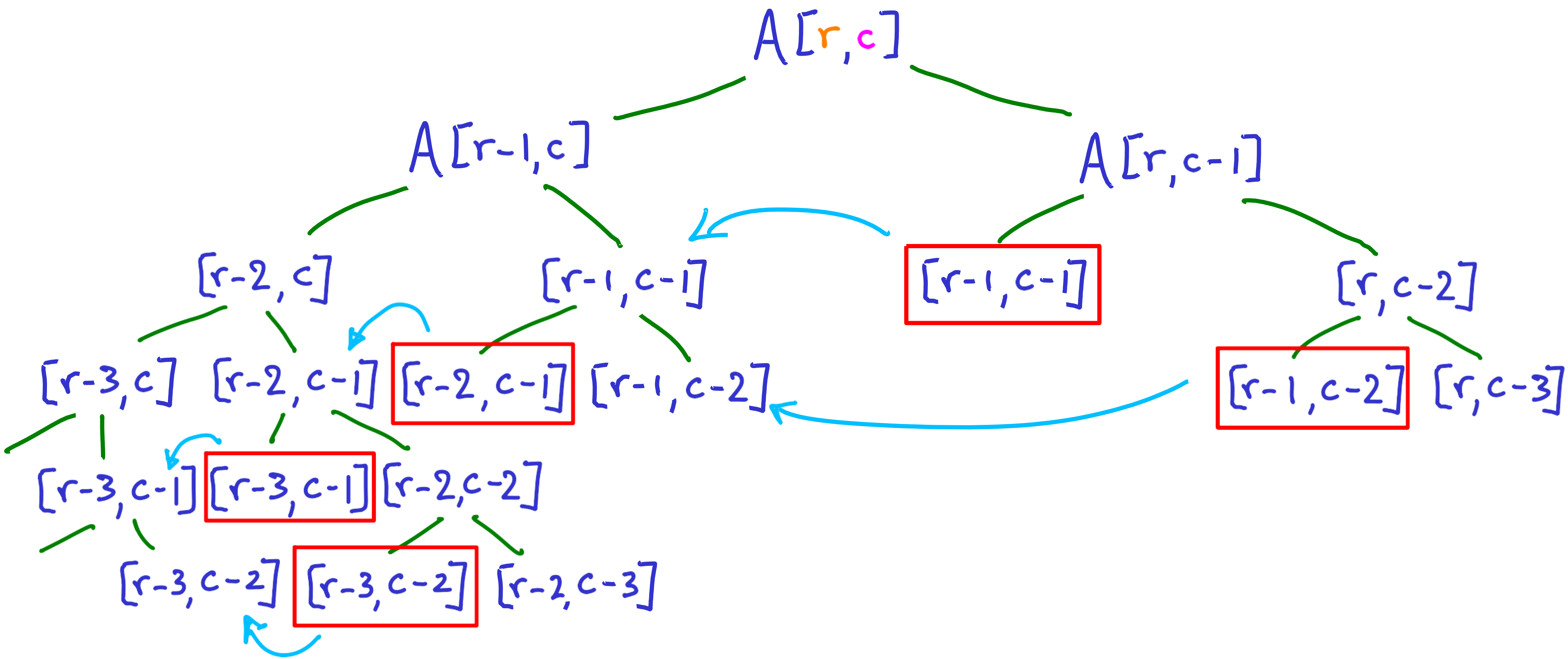
Starting at top-left of $n \times m$ grid, moving only down or right, how many ways to reach bottom-right?



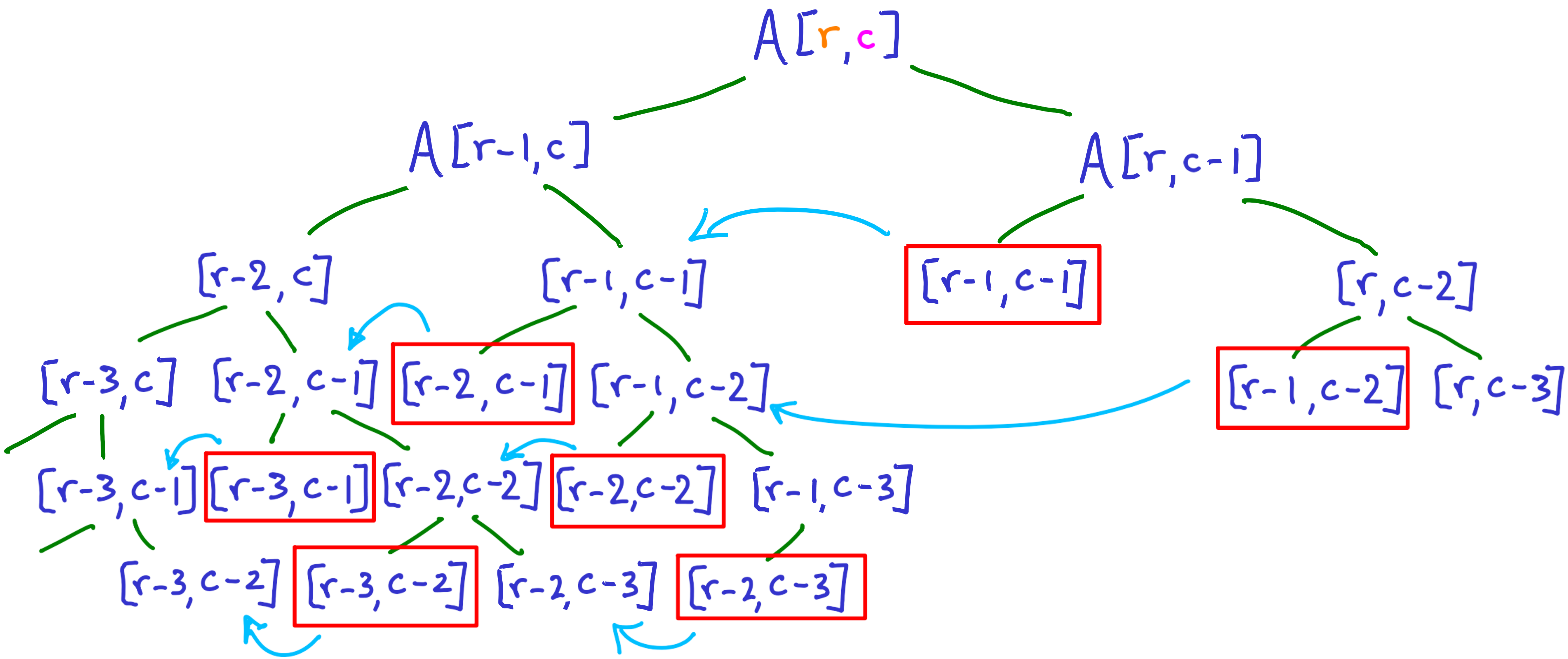
Starting at top-left of $n \times m$ grid, moving only down or right, how many ways to reach bottom-right?



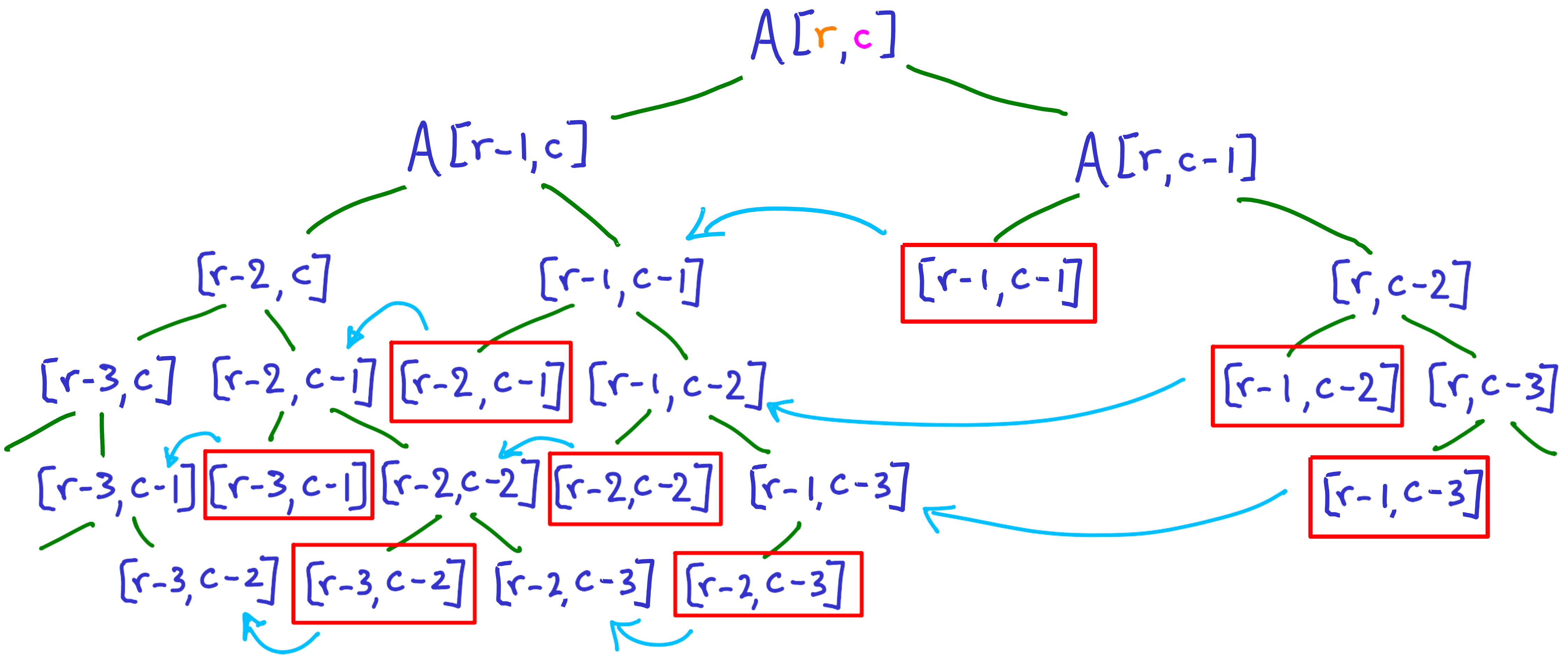
Starting at top-left of $n \times m$ grid, moving only down or right, how many ways to reach bottom-right?



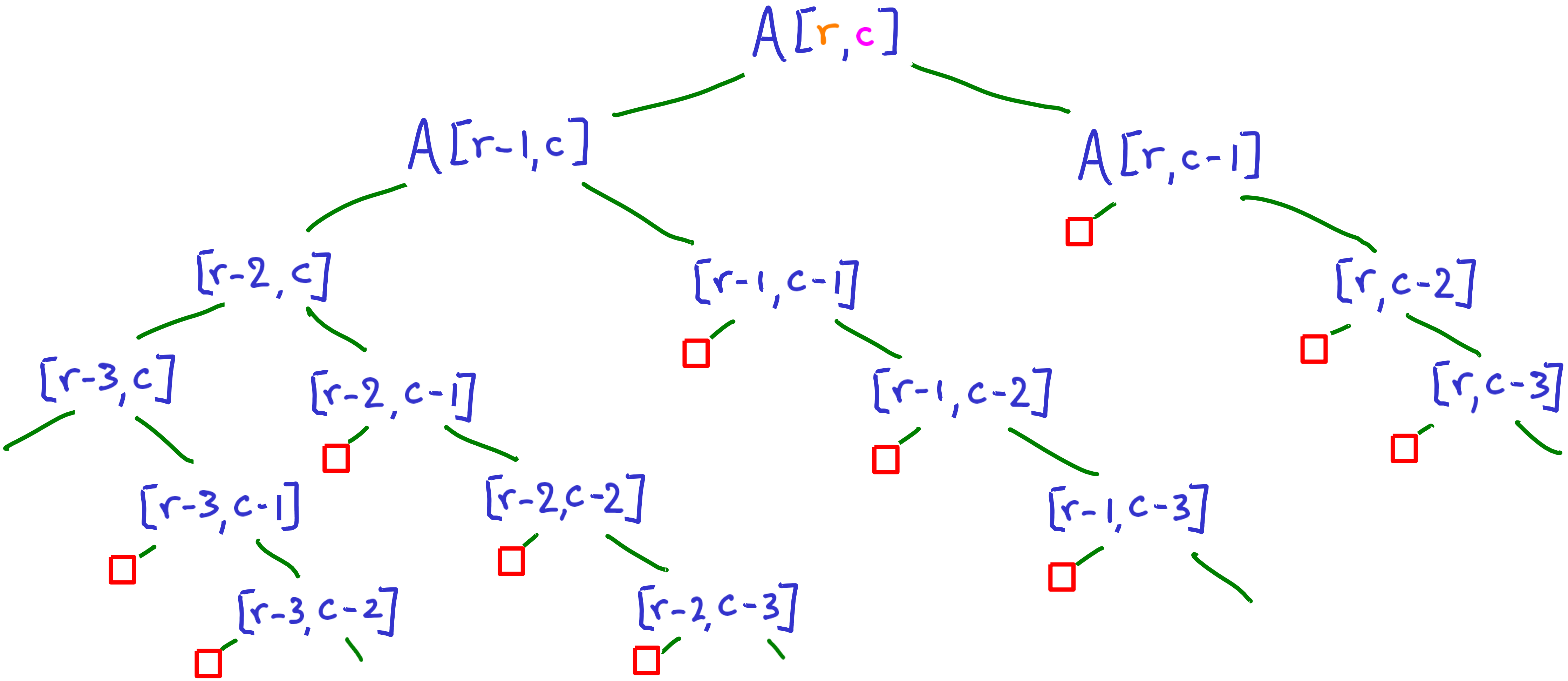
Starting at top-left of n×m grid, moving only down or right, how many ways to reach bottom-right?



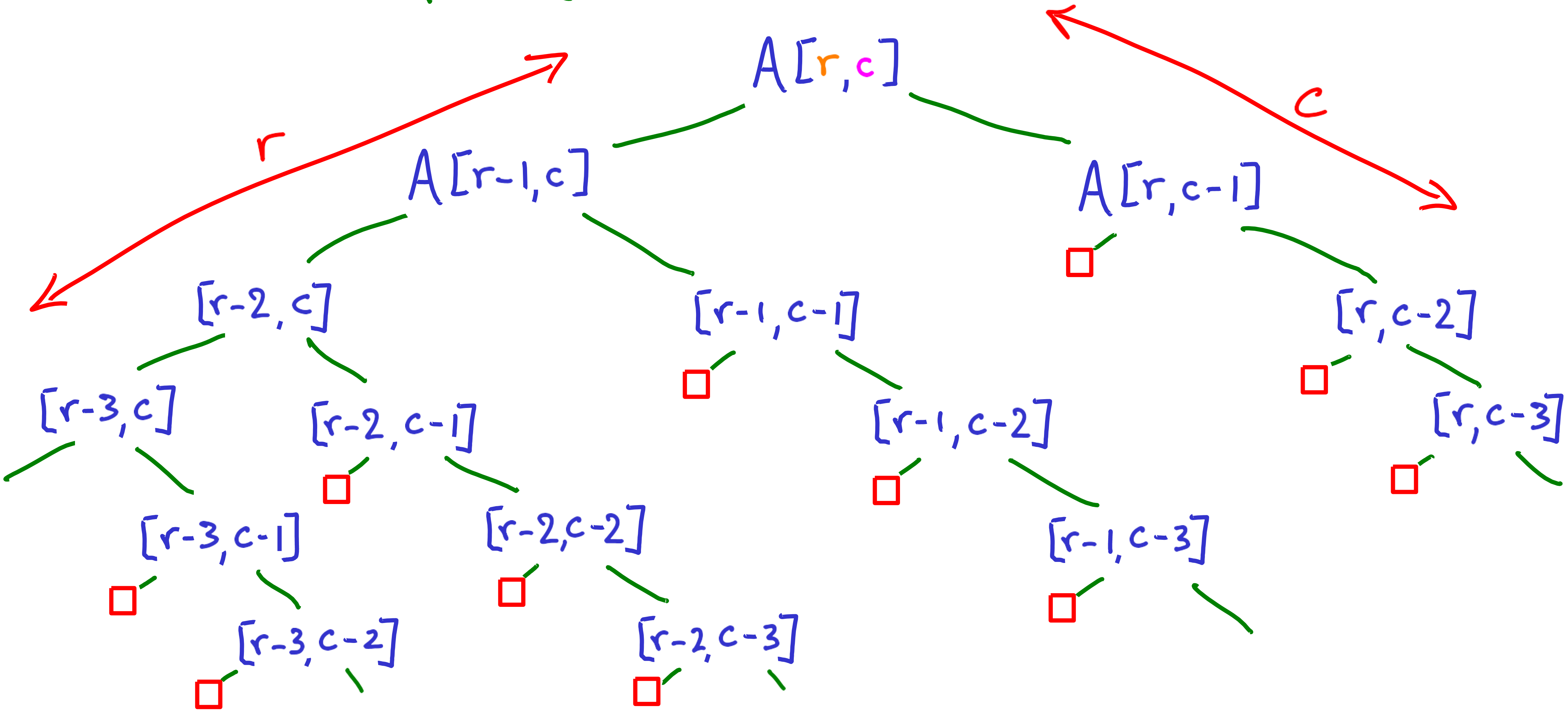
Starting at top-left of $n \times m$ grid, moving only down or right, how many ways to reach bottom-right?



How many times will we recurse in a unique way?



How many times will we recurse in a unique way? → $r \cdot c$ distinct subproblems



MEMOIZATION (making memos)

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

	1	2	3	4	5	6	7	8	9
1									
2									
3									
4									
5									
6									$\boxed{r,c}$

Recursion:

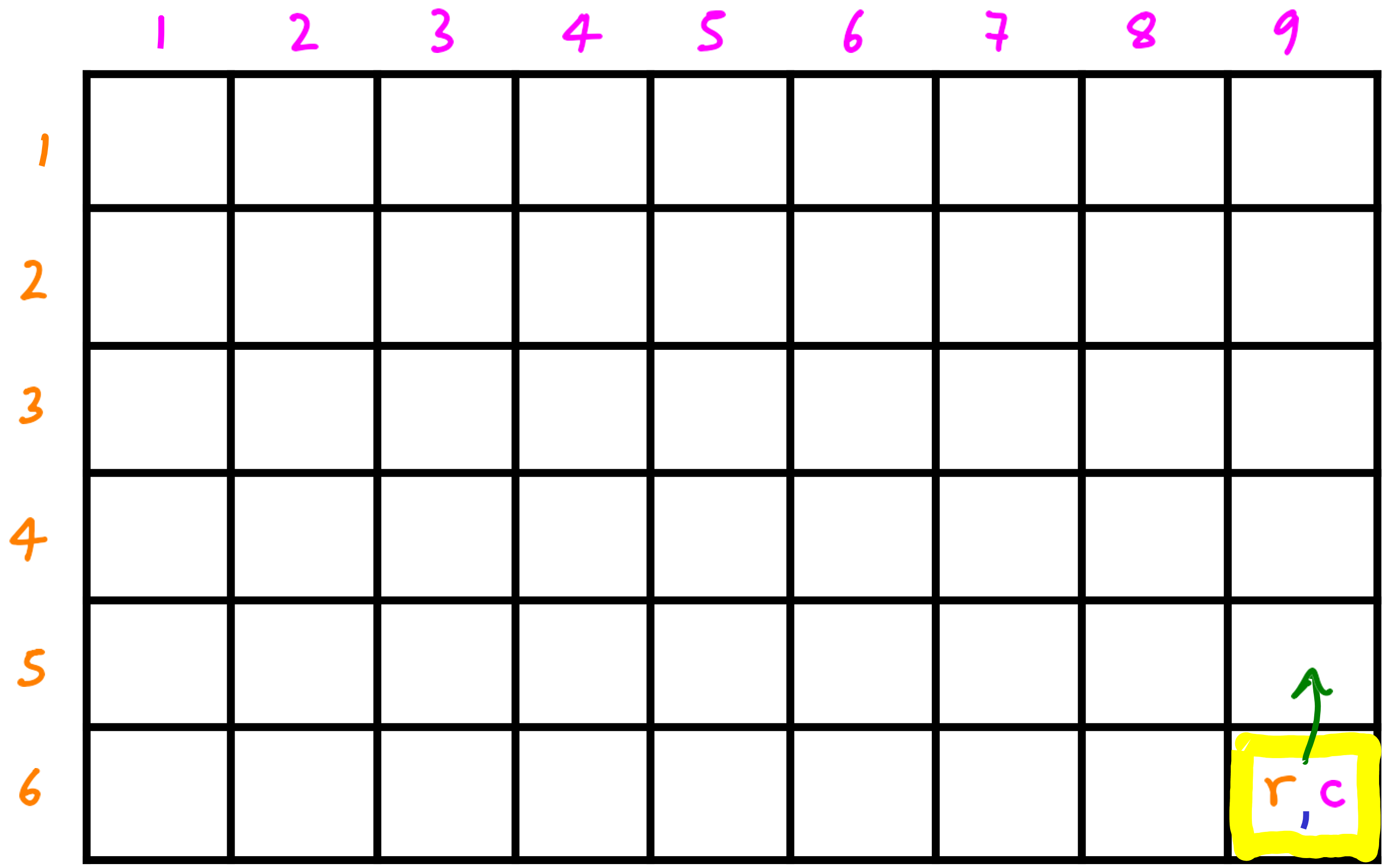
first find $A[r-1,c]$ $\uparrow$

then find $A[r,c-1]$ $\leftarrow$

Memoization (making memos)

For this problem, $m \times n$ table

$$A[r,c] = A[r-1,c] + A[r,c-1]$$



Recursion:

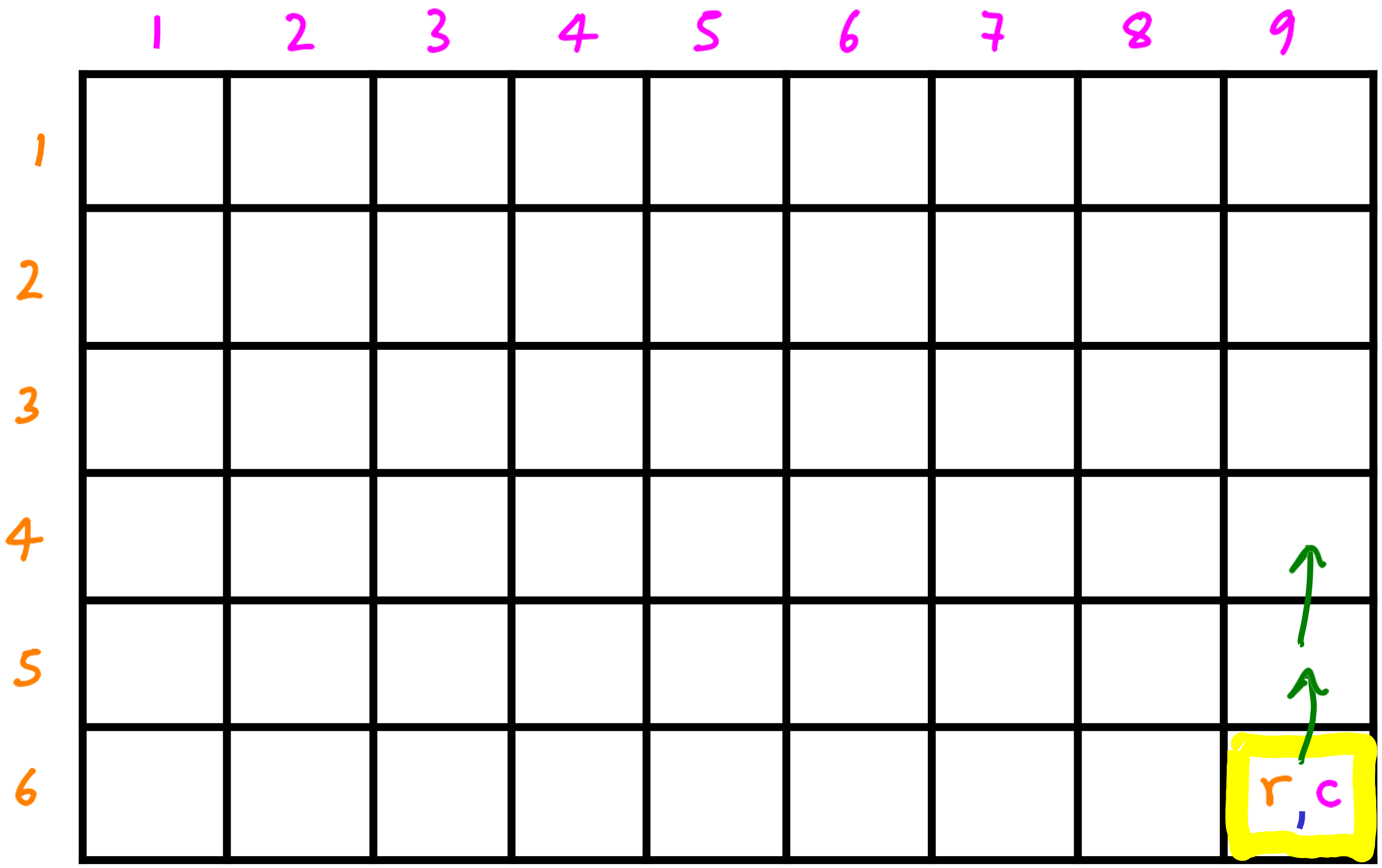
first find $A[r-1,c]$ ↑

then find $A[r,c-1]$ ←

MEMOIZATION (making memos)

For this problem, $m \times n$ table

$$A[r,c] = A[r-1,c] + A[r,c-1]$$



Recursion:

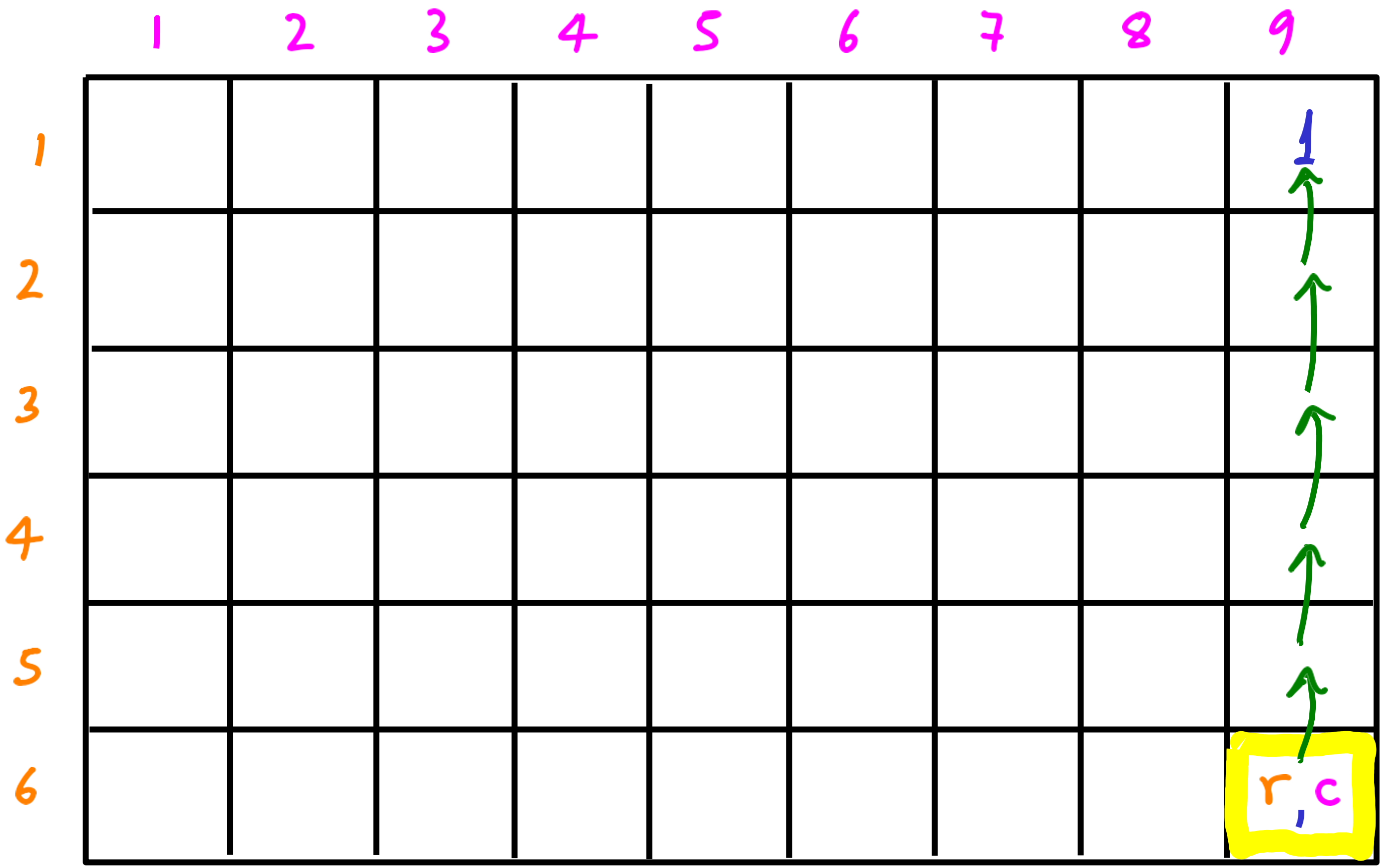
first find $A[r-1,c]$ ↑

then find $A[r,c-1]$ ←

MEMOIZATION (making memos)

For this problem, $m \times n$ table

$$A[r,c] = A[r-1,c] + A[r,c-1]$$



Recursion:

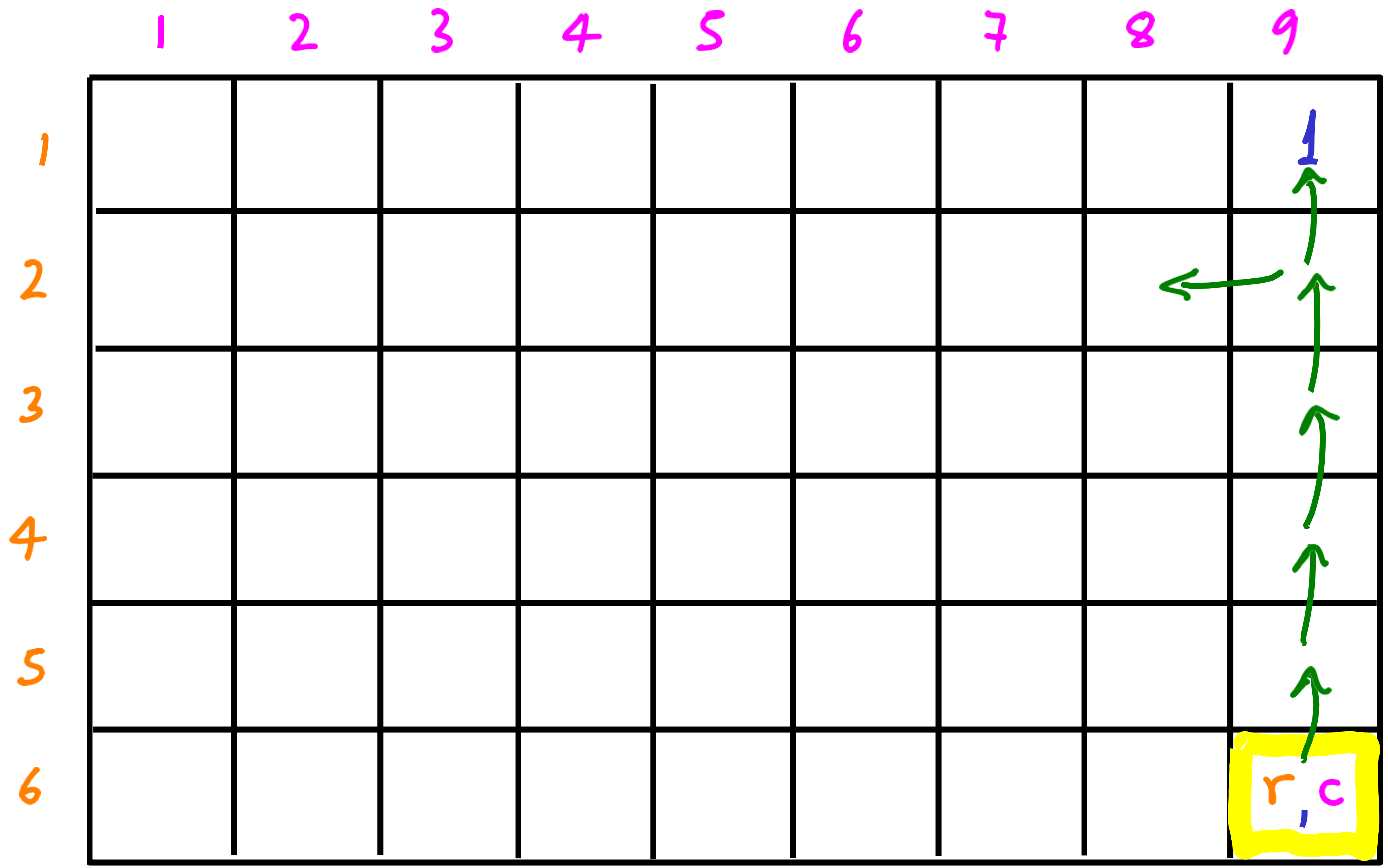
first find $A[r-1,c]$ ↑

then find $A[r,c-1]$ ←

MEMOIZATION (making memos)

For this problem, $m \times n$ table

$$A[r,c] = A[r-1,c] + A[r,c-1]$$



Recursion:

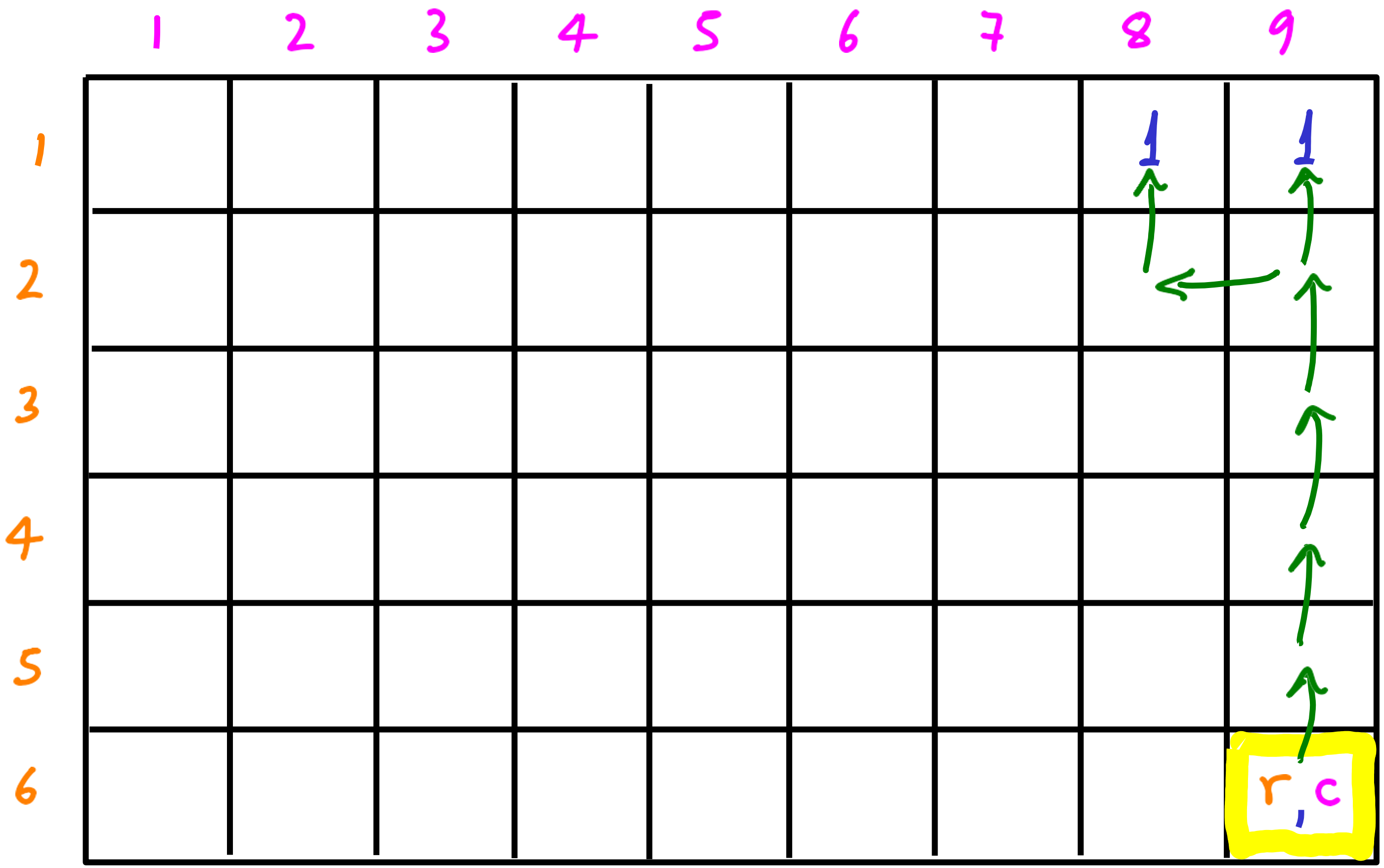
first find $A[r-1,c]$ ↑

then find $A[r,c-1]$ ←

MEMOIZATION (making memos)

For this problem, m x n table

$$A[r,c] = A[r-1,c] + A[r,c-1]$$



Recursion:

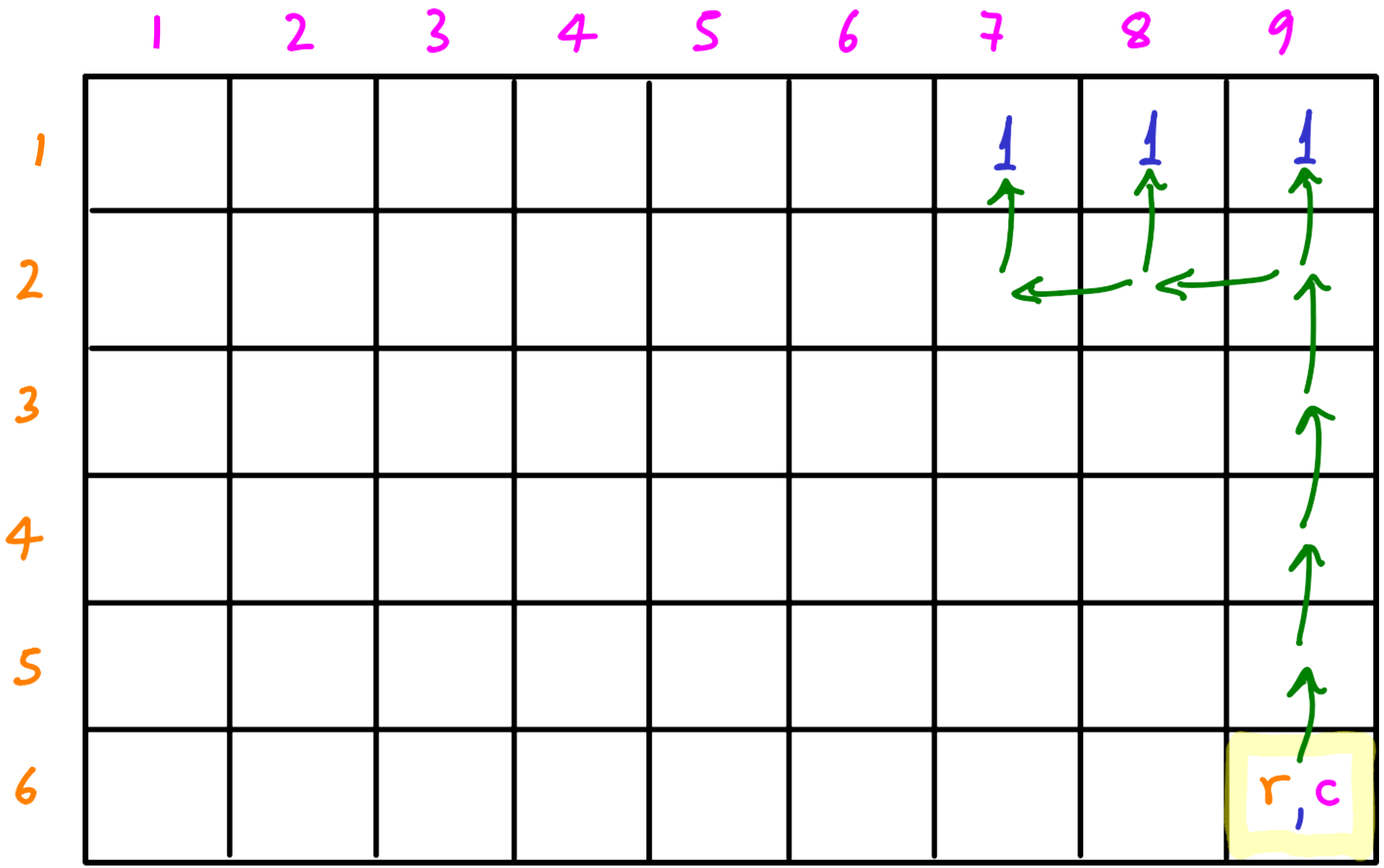
first find $A[r-1,c]$ ↑

then find $A[r,c-1]$ ←

MEMOIZATION (making memos)

For this problem, $m \times n$ table

$$A[r,c] = A[r-1,c] + A[r,c-1]$$



Recursion:

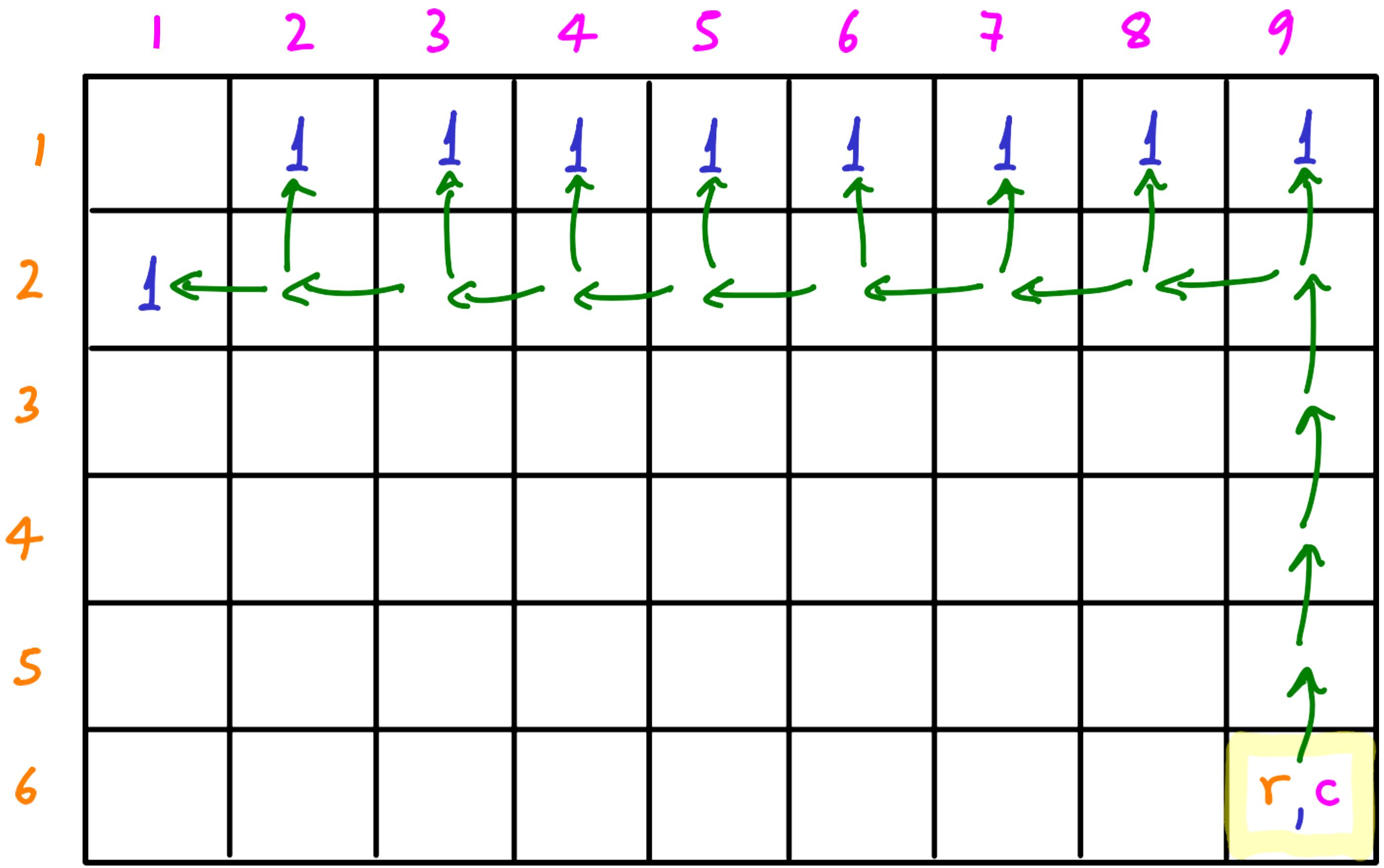
first find $A[r-1,c]$ $\uparrow$

then find $A[r,c-1]$ $\leftarrow$

MEMOIZATION (making memos)

For this problem, m×n table

$$A[r,c] = A[r-1,c] + A[r,c-1]$$



Recursion:

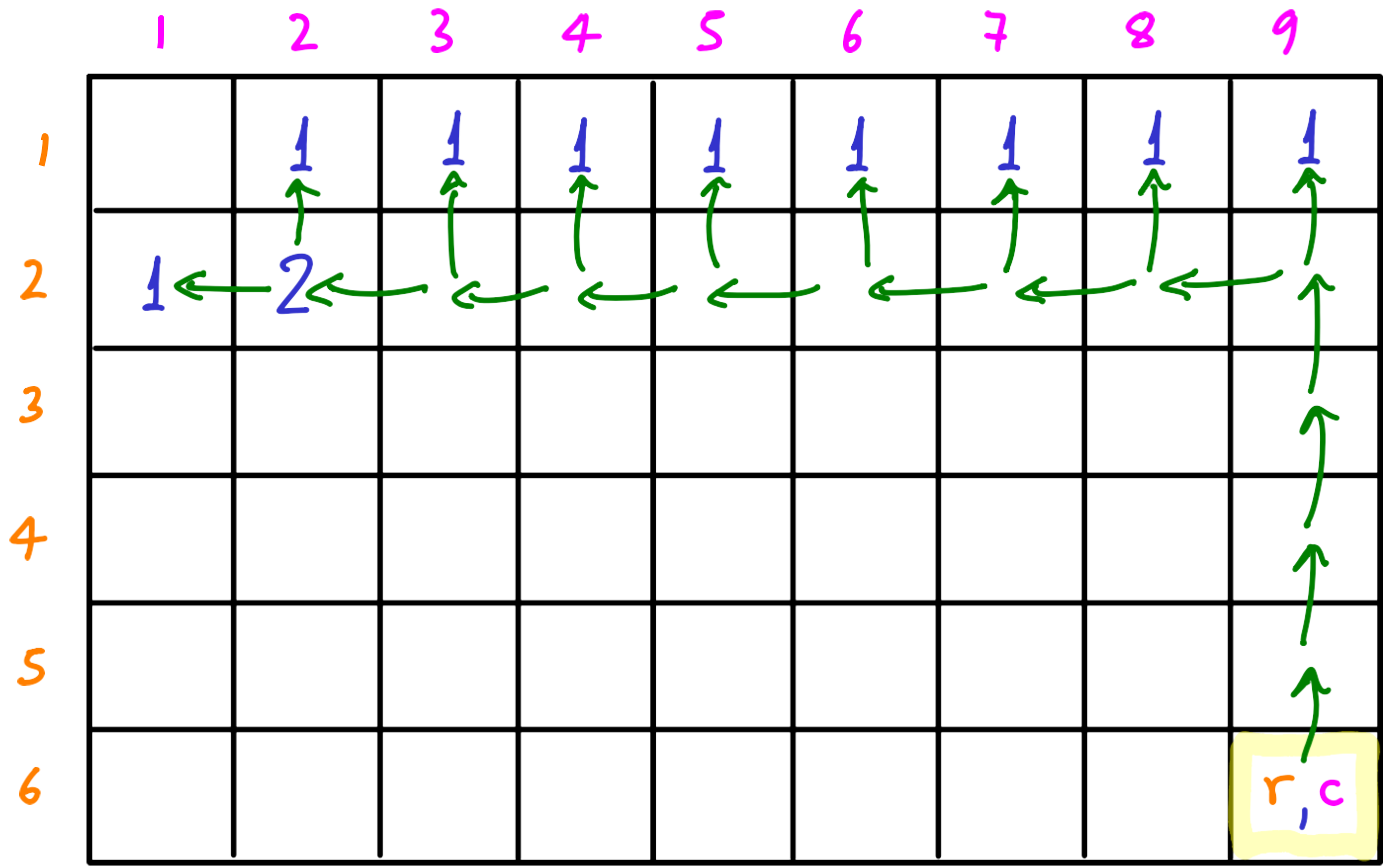
first find $A[r-1,c]$ ↑

then find $A[r,c-1]$ ←

Memoization (making memos)

For this problem, $m \times n$ table

$$A[r,c] = A[r-1,c] + A[r,c-1]$$



Recursion:

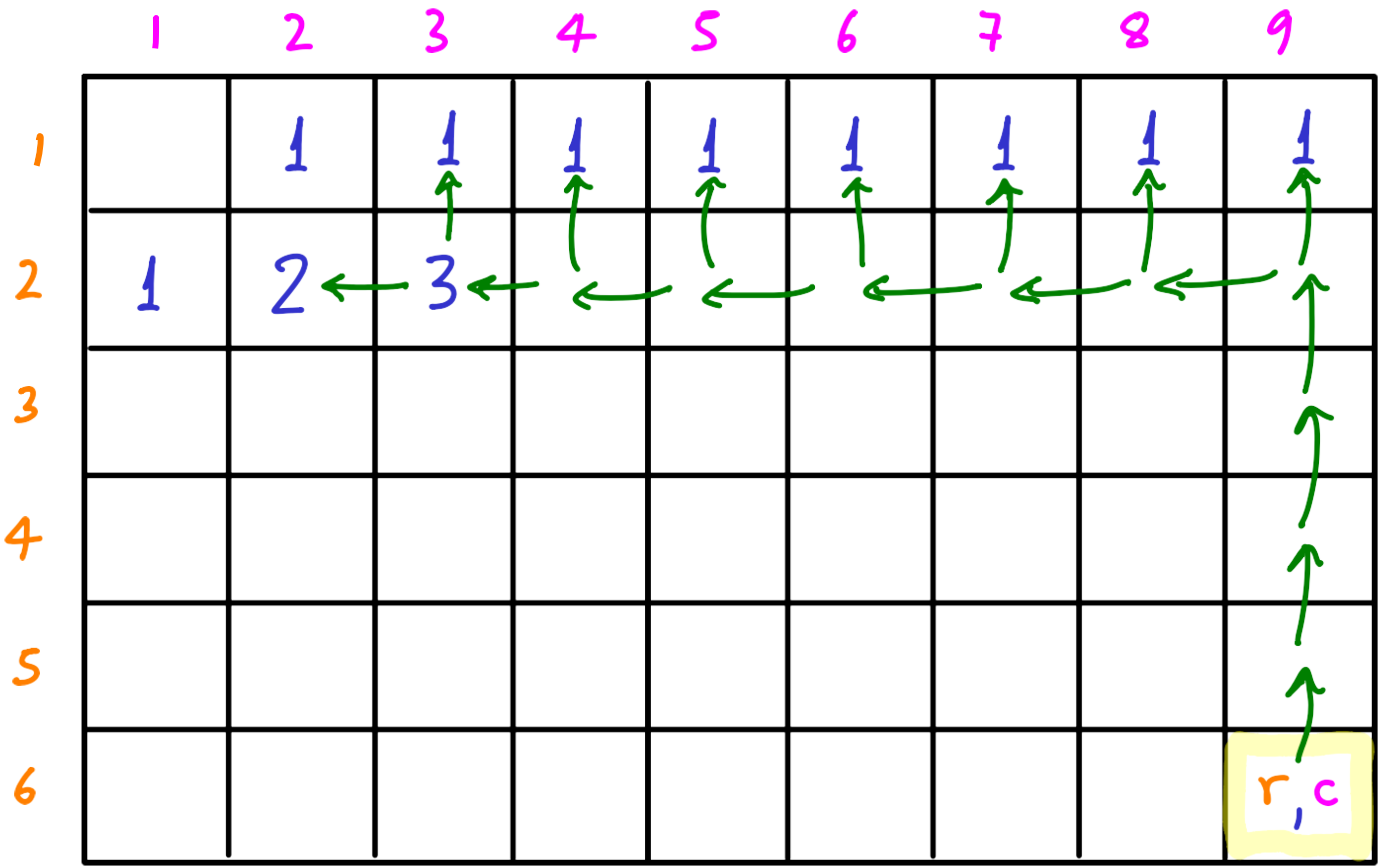
first find $A[r-1,c]$ ↑

then find $A[r,c-1]$ ←

MEMOIZATION (making memos)

For this problem, $m \times n$ table

$$A[r,c] = A[r-1,c] + A[r,c-1]$$



Recursion:

first find $A[r-1,c]$ ↑

then find $A[r,c-1]$ ←

Memoization (making memos)

For this problem, m×n table

$$A[r,c] = A[r-1,c] + A[r,c-1]$$

	1	2	3	4	5	6	7	8	9
1		1	1	1	1	1	1	1	1
2	1	2	3	4	5	6	7	8	9
3									
4									
5									
6									r, c

Recursion:

first find $A[r-1,c]$ $\uparrow$

then find $A[r,c-1]$ $\leftarrow$

MEMOIZATION (making memos)

For this problem, $m \times n$ table

$$A[r,c] = A[r-1,c] + A[r,c-1]$$

	1	2	3	4	5	6	7	8	9
1		1	1	1	1	1	1	1	1
2	1	2	3	4	5	6	7	8	9
3					etc				
4									
5									
6									r, c

Recursion:

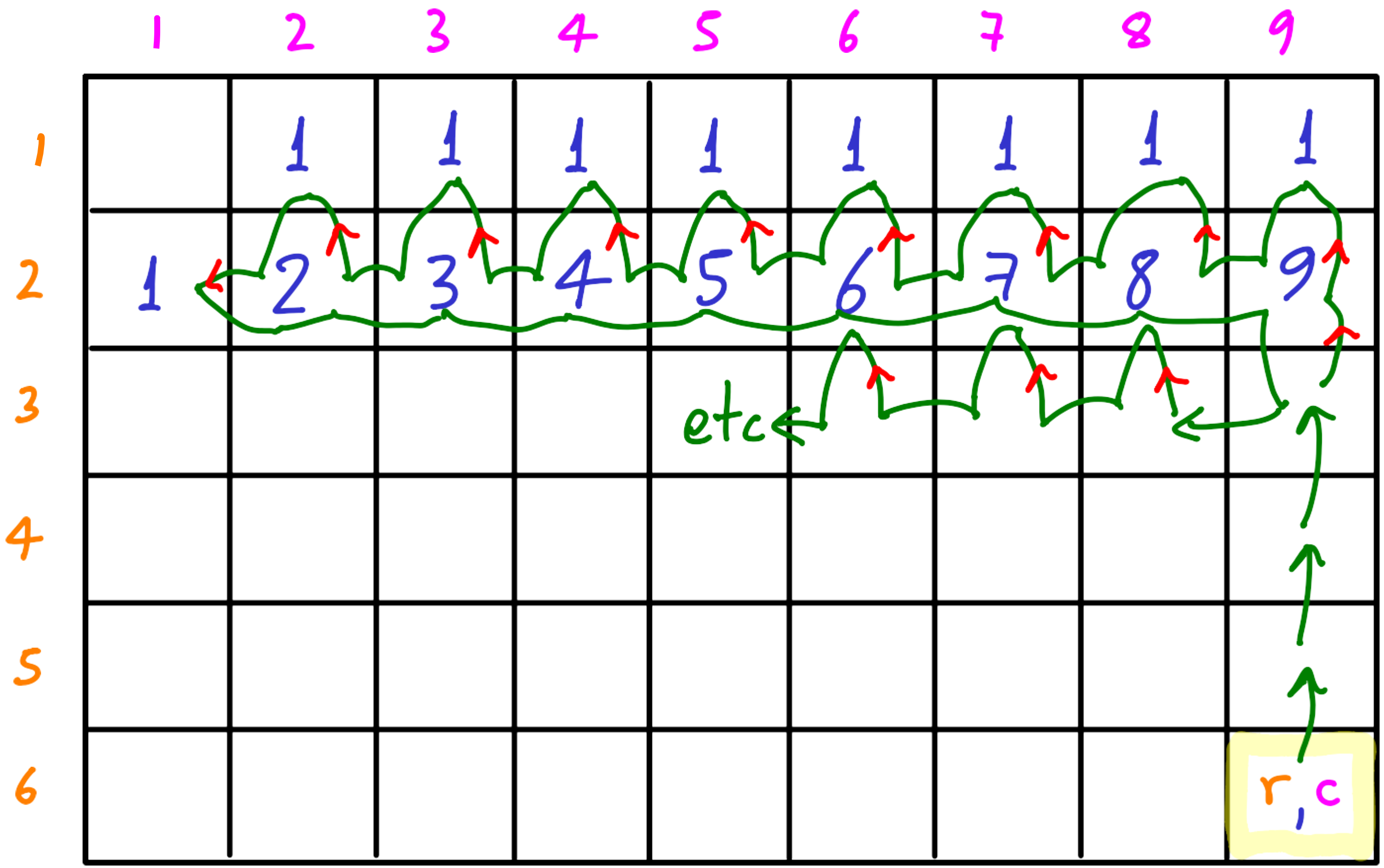
first find $A[r-1,c]$ ↑

then find $A[r,c-1]$ ←

MEMOIZATION (making memos)

For this problem, $m \times n$ table

$$A[r,c] = A[r-1,c] + A[r,c-1]$$



Recursion:

first find $A[r-1,c]$ ↑

then find $A[r,c-1]$ ←

$\Theta(n \cdot m)$

time & space

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)

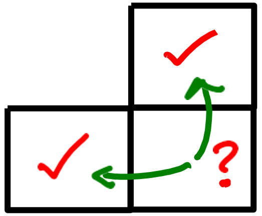
	1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	1	1	1	1
2	1								
3	1								
4	1								
5	1								
6	1								

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)

	1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	1	1	1	1
2	1								
3	1								
4	1								
5	1								
6	1								

$$A[r,c] = \underbrace{A[r-1,c] + A[r,c-1]}$$



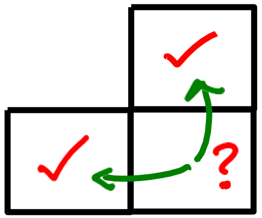
fill any cell as long as what it depends on is full

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

DYNAMIC PROGRAMMING (bottom-up: base cases first)

	1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	1	1	1	1
2	1	2							
3	1								
4	1								
5	1								
6	1								

$$A[r,c] = \underbrace{A[r-1,c] + A[r,c-1]}$$



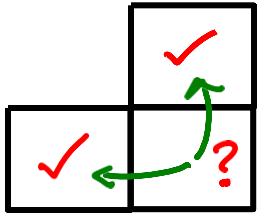
fill any cell as long as what it depends on is full

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)

	1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	1	1	1	1
2	1	2	3						
3	1								
4	1								
5	1								
6	1								

$$A[r,c] = \underbrace{A[r-1,c] + A[r,c-1]}$$



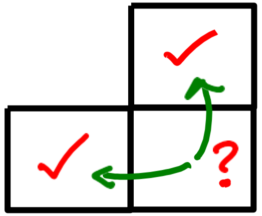
fill any cell as long as what it depends on is full

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)

	1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	1	1	1	1
2	1	2	3	4					
3	1								
4	1								
5	1								
6	1								

$$A[r,c] = \underbrace{A[r-1,c] + A[r,c-1]}$$



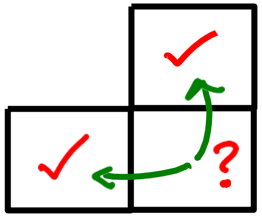
fill any cell as long as what it depends on is full

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)

	1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	1	1	1	1
2	1	2	3	4					
3	1	3							
4	1								
5	1								
6	1								

$$A[r,c] = \underbrace{A[r-1,c] + A[r,c-1]}$$



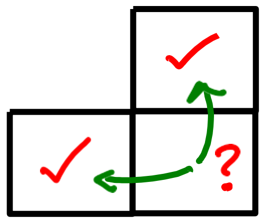
fill any cell as long as what it depends on is full

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)

	1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	1	1	1	1
2	1	2	3	4	5	6	7	8	
3	1	3	6						
4	1	4	10						
5	1	5							
6	1	6							

$$A[r,c] = \underbrace{A[r-1,c] + A[r,c-1]}$$



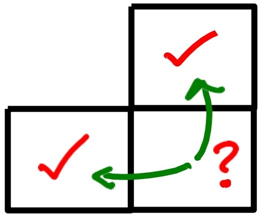
fill any cell as long as what it depends on is full

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)

	1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	1	1	1	1
2	1	2	3	4	5	6	7	8	9
3	1	3	6	10	15	21	28	36	45
4	1	4	10	20	35	56	84	120	165
5	1	5	15	35	70	126	210	330	495
6	1	6	21	56	126	252	462	792	1287

$$A[r,c] = \underbrace{A[r-1,c] + A[r,c-1]}$$



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

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

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

$$\begin{array}{cccccccccc} 1 & 2 & 3 & \dots & & \dots & & & n-1 & n \\ 23, & 3, & 5, & 18, & 10, & 101, & 12, & 14, & 4, & 105 \end{array}$$

$|L_{n-1}| = 2$

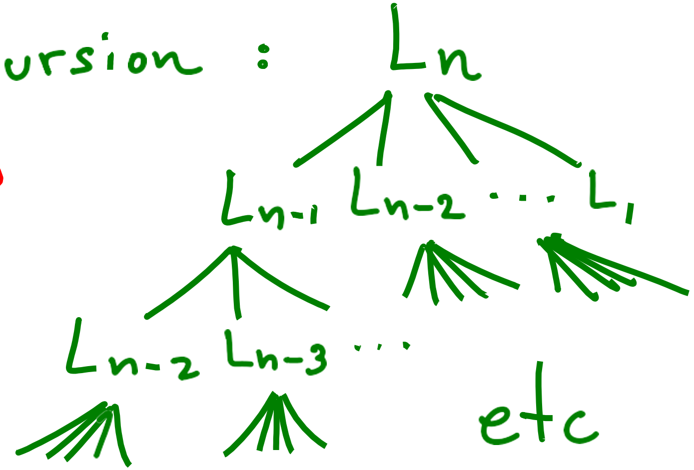
$$|L_n| = 1 + \max_{\{\text{all } j \text{ s.t. } S[j] < S[n]\}} |L_j|$$

look at all L_j $(j<n)$

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

Recursion :

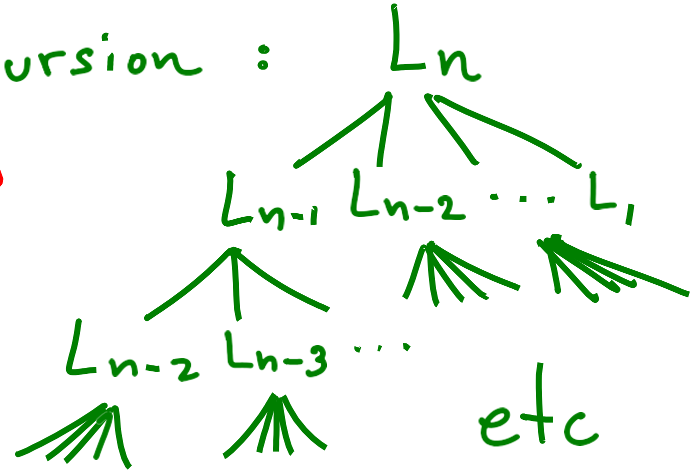
BAD



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

Recursion :

BAD



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

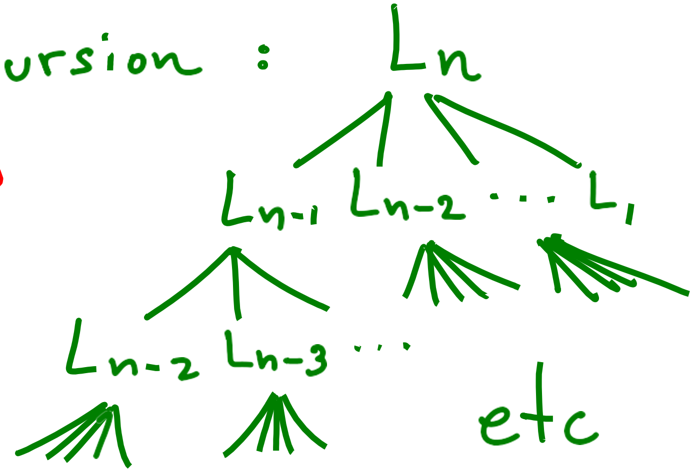
1

$|L_1|$

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

Recursion :

BAD



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

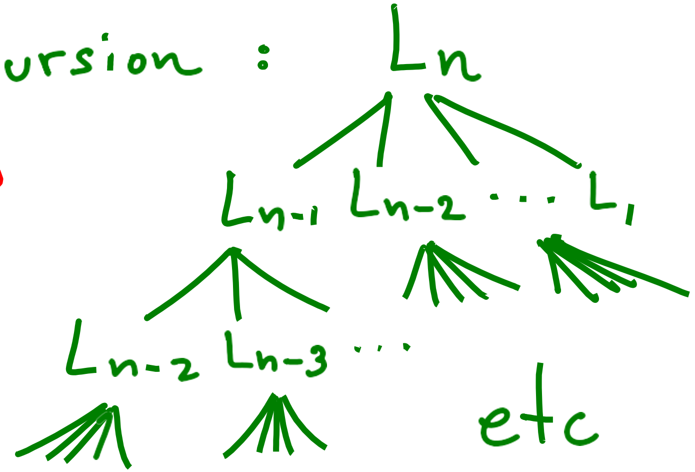
1 1

$|L_2|$

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

Recursion:

BAD



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.

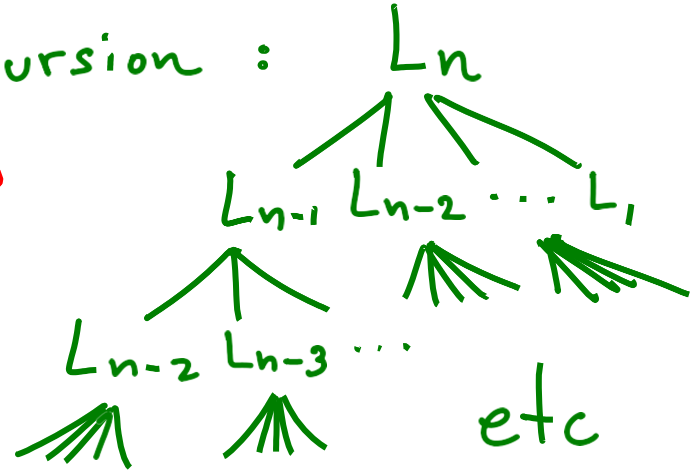
23, 3, 5, 18, 10, 101, 12, 14, 4

1 1 2

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

Recursion :

BAD



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.

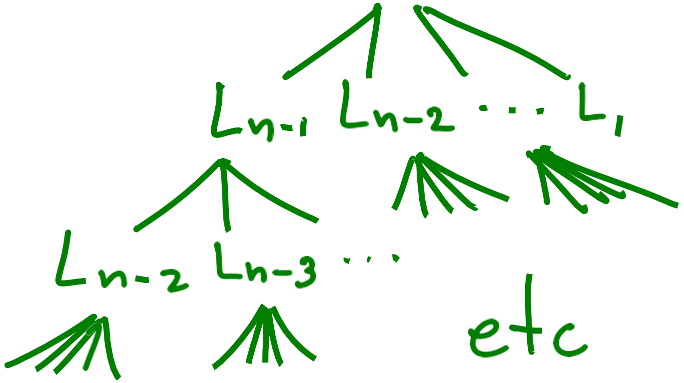
23 , 3 , 5 , 18 , 10 , 101 , 12 , 14 , 4

1 1 2 3

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

Recursion : L_n

BAD



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.

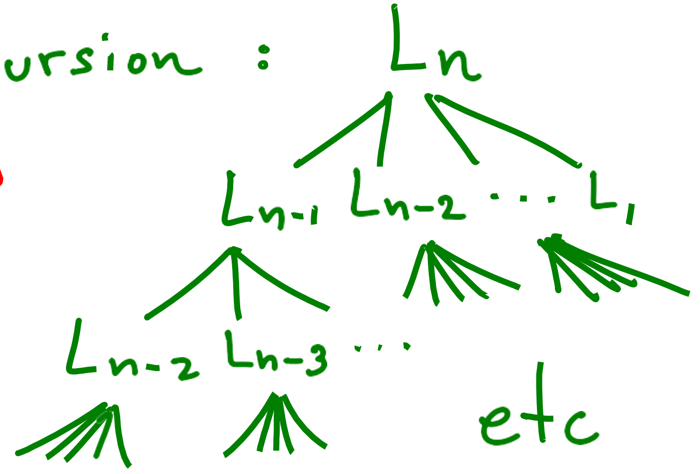
23, 3, 5, 18, 10, 101, 12, 14, 4

1 1 2 3 3

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

Recursion :

BAD



Dyn. Prog: Build solutions "bottom up"

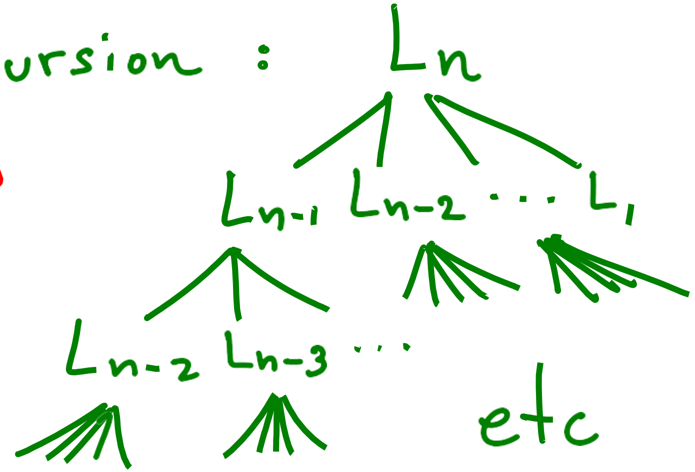
When it's time to solve $|L_k|$ we have stored all $|L_j|$ $(j<k)$ in an array.

23	3	5	18	10	101	12	14	4
1	1	2	3	3	4			

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

Recursion :

BAD



Dyn. Prog: Build solutions "bottom up".

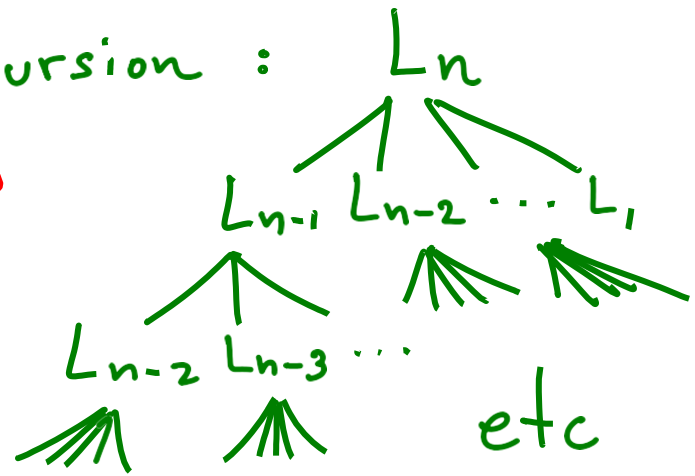
When it's time to solve $|L_k|$ we have stored all $|L_j|$ $(j<k)$ in an array.

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 : BAD



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

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

Recursion : L_n

BAD

L_{n-1} L_{n-2} ... L_1

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 $|L_j|$ $(j<k)$ in an array.

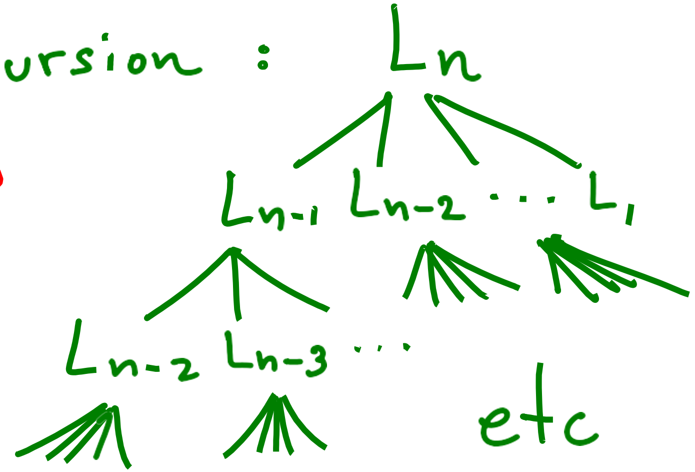
23	3	5	18	10	101	12	14	4
1	1	2	3	3	4	4	5	2

→ Score may decrease

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

Recursion :

BAD



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.

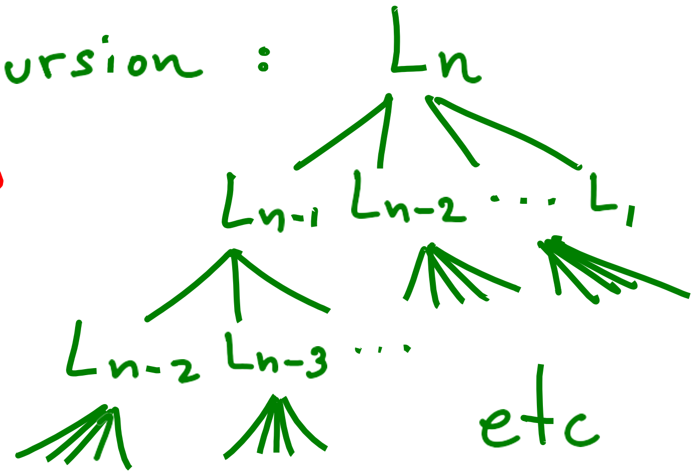
23, 3, 5, 18, 10, 101, 12, 14, 4

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 :

BAD



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?

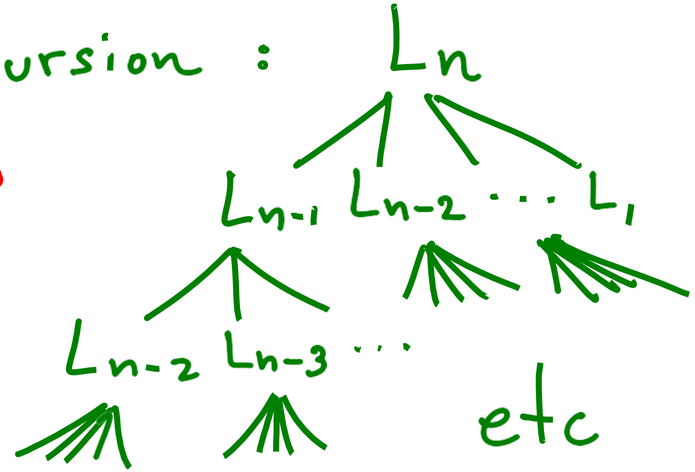
23 , 3 , 5 , 18 , 10 , 101 , 12 , 14 , 4

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 :

BAD



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.

$T(k) = \Theta(k)$

$$T(n) = \sum_{i=1}^{n} T(k) = \Theta(n^2)$$

$\text{Space} = \Theta(n)$

23	3	5	18	10	101	12	14	4
1	1	2	3	3	4	4	5	2

$T(n) = \Theta(n^2)$

$\text{space} = \Theta(n)$

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

LONGEST COMMON SUBSEQUENCE

&

DYNAMIC PROGRAMMING

LONGEST COMMON SUBSEQUENCE

&

DYNAMIC PROGRAMMING

$$X : \overbrace{A\ B\ C\ B\ D\ A\ B}^{n}$$

$$Y : \underbrace{B\ D\ C\ A\ B\ A}_{m \leq n}$$

LONGEST COMMON SUBSEQUENCE

&

DYNAMIC PROGRAMMING

$$X : \overbrace{A\ B\ C\ B\ D\ A\ B}^{n}$$

$$Y : \underbrace{B\ D\ C\ A\ B\ A}_{m \leq n}$$

$$\left.\right\} |LCS(x,y)| = 4$$

LONGEST COMMON SUBSEQUENCE

&

DYNAMIC PROGRAMMING

$$X : \overbrace{A\ B\ C\ B\ D\ A\ B}^{n}$$

$$Y : \underbrace{B\ D\ C\ A\ B\ A}_{m \leq n}$$

$$|LCS(x,y)| = 4$$

LONGEST COMMON SUBSEQUENCE

&

DYNAMIC PROGRAMMING

$$X : \overbrace{A\ B\ C\ B\ D\ A\ B}^{n}$$

$$Y : \underbrace{B\ D\ C\ A\ B\ A}_{m \leq n}$$

$$|LCS(x,y)| = 4$$

LONGEST COMMON SUBSEQUENCE

&

DYNAMIC PROGRAMMING

$$X : \overbrace{A\ B\ C\ B\ D\ A\ B}^{n}$$

$$Y : \underbrace{B\ D\ C\ A\ B\ A}_{m \le n}$$

$$|LCS(X,Y)| = 4$$

Brute force to find LCS:

for every subsequence of Y

$\Theta(2^m)$

LONGEST COMMON SUBSEQUENCE

&

DYNAMIC PROGRAMMING

$$X : \overbrace{A\ B\ C\ B\ D\ A\ B}^{n}$$

$$Y : \underbrace{B\ D\ C\ A\ B\ A}_{m \leq n}$$

$$|LCS(X,Y)| = 4$$

Brute force to find LCS:

for every subsequence of Y → $\Theta(2^m)$

check if it exists in X

↳ $O(n)$: easy

LONGEST COMMON SUBSEQUENCE
&
DYNAMIC PROGRAMMING

$X : \overbrace{A\ B\ C\ B\ D\ A\ B}^{n}$

$Y : \underbrace{B\ D\ C\ A\ B\ A}_{m \leq n}$

$|LCS(X,Y)| = 4$

Brute force to find LCS:

for every subsequence of Y $\rightarrow \Theta(2^m)$

check if it exists in X $\rightarrow O(n)$: easy

$O(n \cdot 2^m)$

Finding |LCS|

$$c(i,j) = |LCS(X[1 \ldots i], Y[1 \ldots j])|$$

Finding $|LCS|$

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

Finding |LCS|

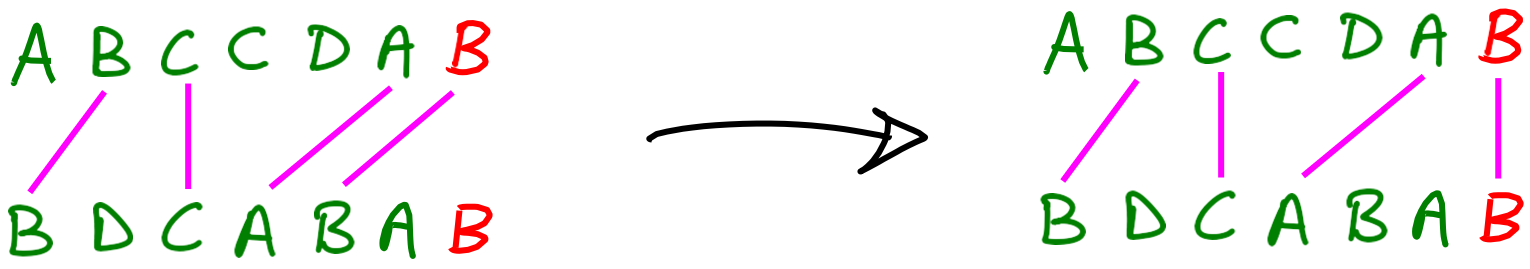
$$c(i,j) = |LCS(X[1\ldots i], Y[1\ldots j])| = \begin{cases} c(i-1,j-1)+1 & \text{if } X[i]=Y[j] \\ \end{cases}$$

A B C C D A B

B D C A B A B

Finding |LCS|

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



Finding |LCS|

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

A B C C D A B

B D C A B A B

Slide last match over:
just as good

$$\boxed{\begin{matrix} A\ B\ C\ C\ D\ A \\ \\ B\ D\ C\ A\ B\ A \end{matrix}}\ \begin{matrix} B \\ | \\ B \end{matrix}$$

$c(i-1, j-1) + 1$

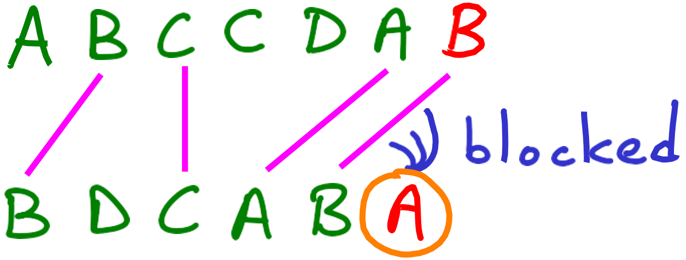
Finding |LCS|

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

Finding |LCS|

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

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



Finding |LCS|

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

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

A B C C D A B

B D C A B A

blocked

A B C C D A B

B D C A B A

blocked

Hide each

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

Finding |LCS|

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

$LCS(X[1\ldots i], Y[1\ldots j])$

cannot use both $X[i]$ and $Y[j]$

A B C C D A B

B D C A B A

blocked

A B C C D A B

B D C A B A

blocked

Hide each and take best result

$c(i,j-1)$

A B C C D A B

B D C A B ~~A~~

$c(i-1,j)$

A B C C D A ~~B~~

B D C A B A

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

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

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

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

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

$LCS(X, Y, i, j)$

return c_{ij}

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

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

$LCS(X, Y, i, j)$

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

return c_{ij}

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

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

$LCS(X, Y, i, j)$ \\ ignoring base case : if i or $j = 0$ then $c_{ij} = 0$

    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}

worst case : ?

$LCS(X,Y,i,j)$

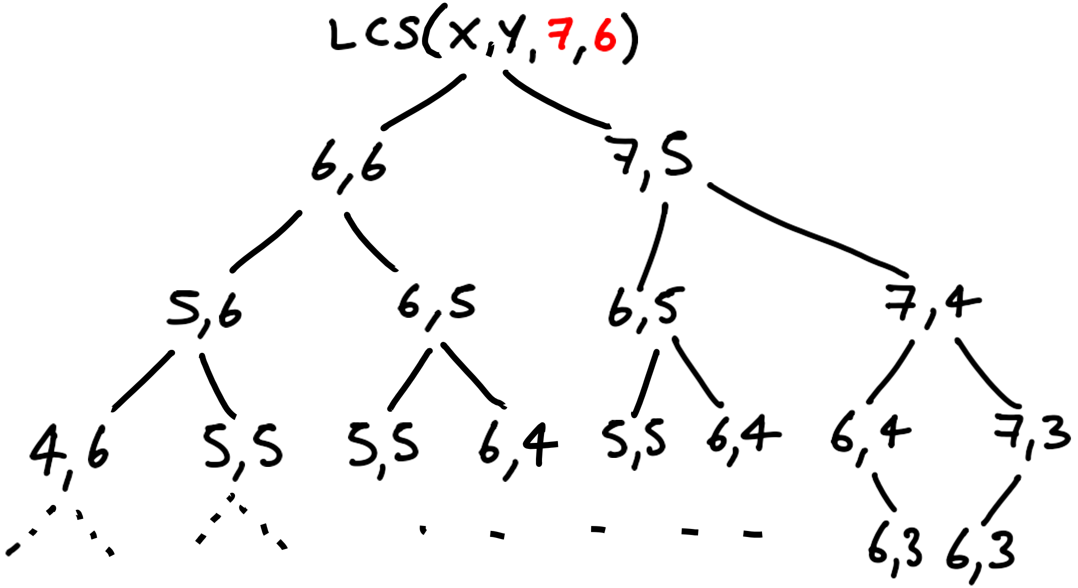
if $X_i = Y_j$ then $c_{ij} \leftarrow LCS(X,Y,i-1,j-1)+1$

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: $n=7$, $m=6$



LCS(X,Y,i,j)

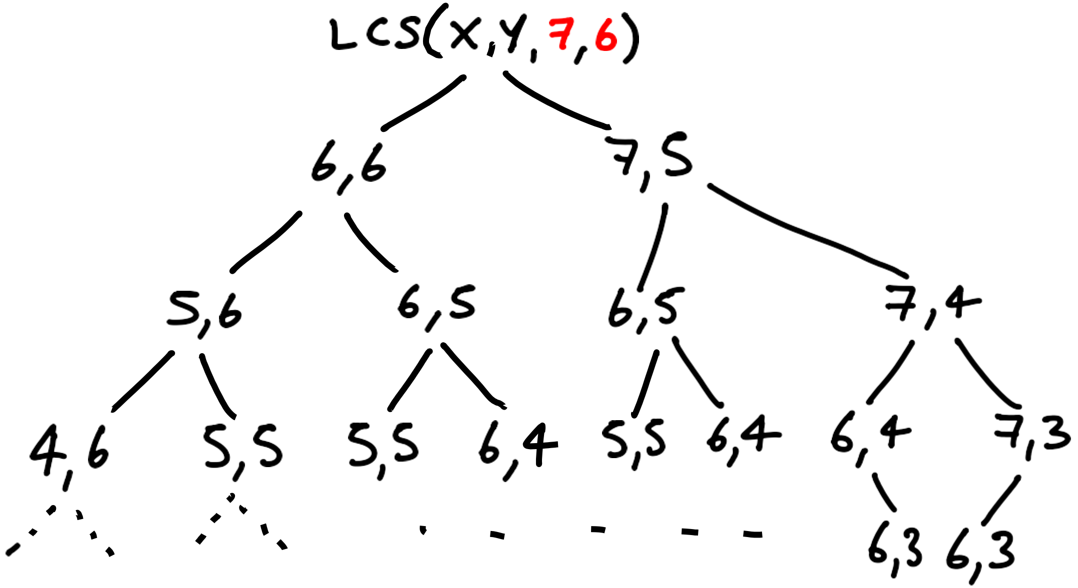
if $X_i = Y_j$ then $c_{ij} \leftarrow$ LCS(X,Y,i-1,j-1) + 1

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



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

work $= \Omega(2^n)$

if m~n

LCS(X, Y, i, j)

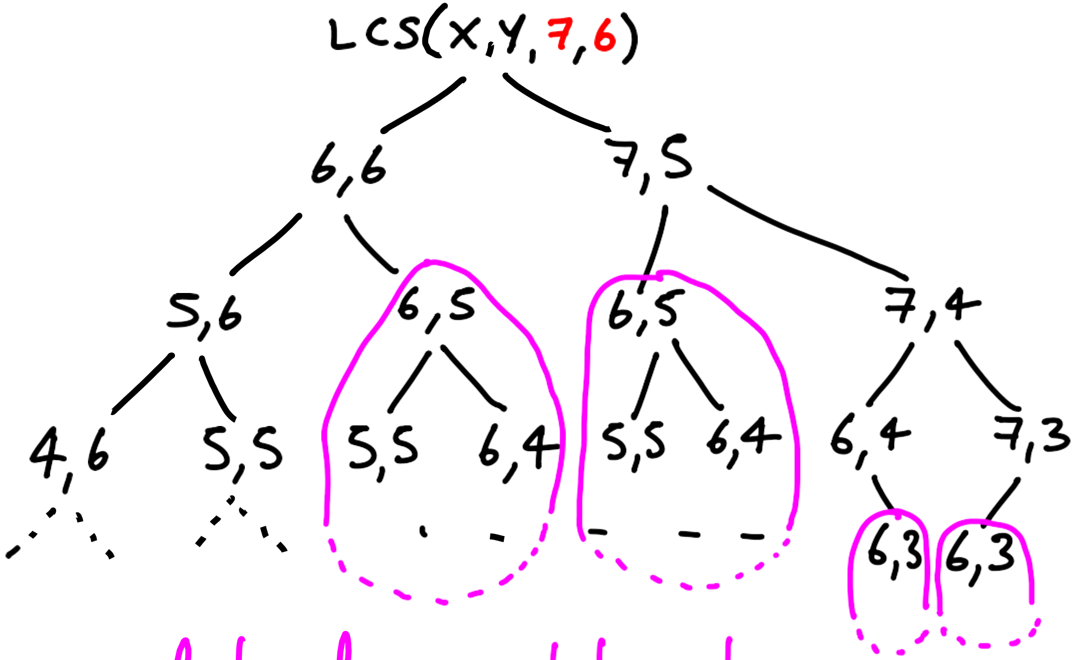
  if $x_i = y_j$ then $c_{ij} \leftarrow$ LCS(X, Y, i-1, j-1) + 1

  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: $n = 7$, $m = 6$



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

work $= \Omega(2^n)$ if $m \sim n$

lots of repeated work

↳ # distinct subproblems = ?

LCS(X, Y, i, j)

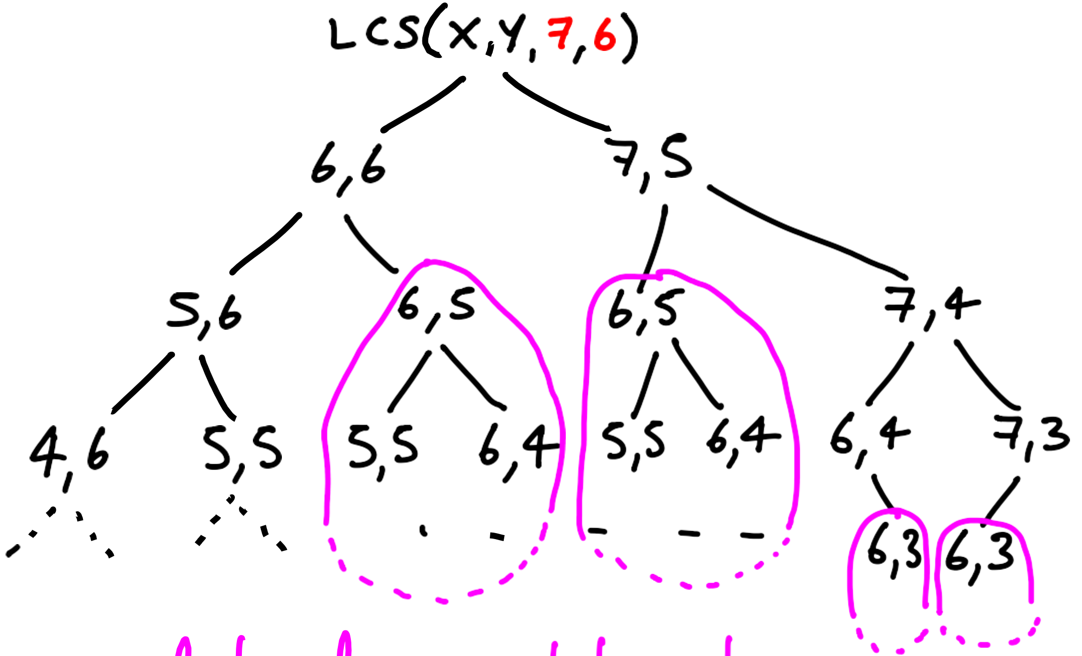
if $X_i = Y_j$ then $c_{ij} \leftarrow$ LCS(X, Y, i-1, j-1) + 1

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



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

work $= \Omega(2^n)$

if $m \sim n$

lots of repeated work

↳ #distinct subproblems $= m \cdot n$

LCS(X,Y,i,j)

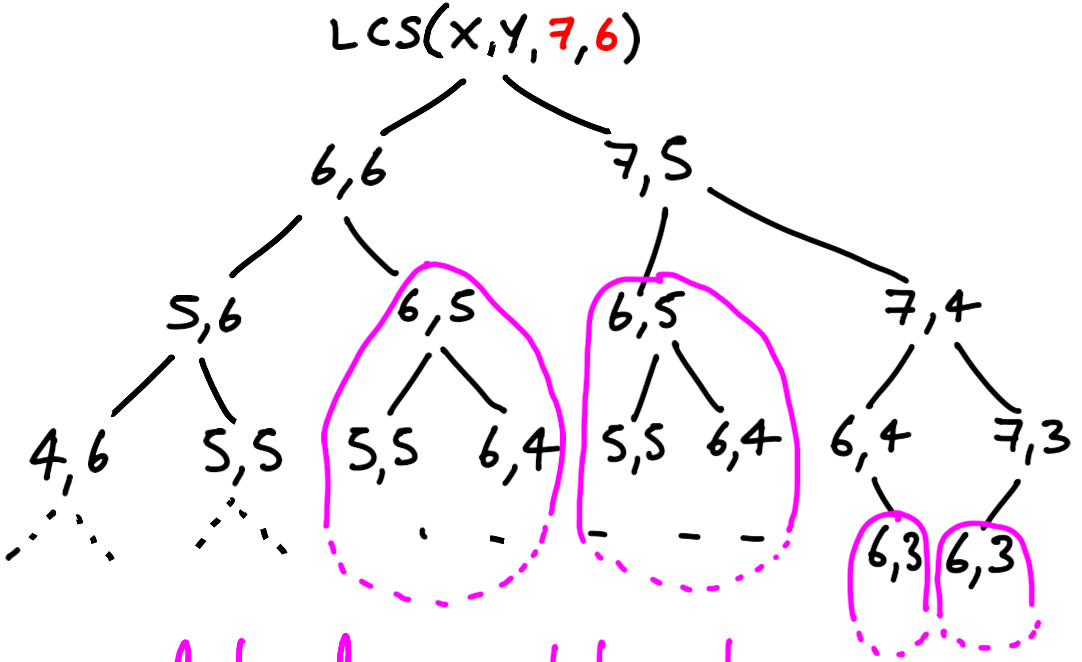
  if $x_i = y_j$ then $c_{ij} \leftarrow \text{LCS}(X,Y,i-1,j-1)+1$

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

  return c_{ij}

worst case : always get $x_i \neq y_j$

ex: $n=7$, $m=6$



lots of repeated work

↳ #distinct subproblems $= m \cdot n$

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

work $= \Omega(2^n)$

if $m \sim n$

Repeated subproblems

+

optimal substructure

↓

try dynamic programming

```
LCS(X,Y,i,j)
    if X_i = Y_j then c_ij ← LCS(X,Y,i-1,j-1) + 1
    else c_ij ← max{LCS(X,Y,i,j-1), LCS(X,Y,i-1,j)}
    return c_ij
```

$\mathrm{LCS}(X, Y, i, j)$

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

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

  return c_{ij}

Memoization

Make "memos" of solutions
(to subproblems)

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}

Memoization

Make "memos" of solutions (to subproblems)

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

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}

Memoization

Make "memos" of solutions (to subproblems)

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

whenever we need to know c_{ij}

 if it's the first time . . . . then calculate it

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}

Memoization

Make "memos" of solutions (to subproblems)

Let $c[1 \ldots m, 1 \ldots n]$ 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 $X_i = Y_j$ 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}

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

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}

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 \\ first time

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}

Memoization

Make "memos" of solutions (to subproblems)

Let c[1...m, 1...n] be a m × 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 \\ 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)}

$LCS(X,Y,i,j)$

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}

Memoization

Make "memos" of solutions (to subproblems)

Let $c[1\ldots m, 1\ldots n]$ 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 \\ first time

if $X_i = Y_j$ then $c[i,j] \leftarrow LCS(X,Y,i-1,j-1)+1$

else $c[i,j] \leftarrow \max\{LCS(X,Y,i,j-1), LCS(X,Y,i-1,j)\}$

return $c[i,j]$ \\ look up

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}

Memoization

Make "memos" of solutions (to subproblems)

time ?

Let c[1...m, 1...n] be a m × 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 \\ 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] \\ look up

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}

Memoization

Make "memos" of solutions (to subproblems)

Let $c[1\ldots m, 1\ldots n]$ 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

$\Theta(mn)$ time & space

LCS(X,Y,i,j)

if $\min\{i,j\} = 0$ then return 0

if $c[i,j] = -1$ then \\ first time

if $X_i = Y_j$ then $c[i,j] \leftarrow LCS(X,Y,i-1,j-1)+1$

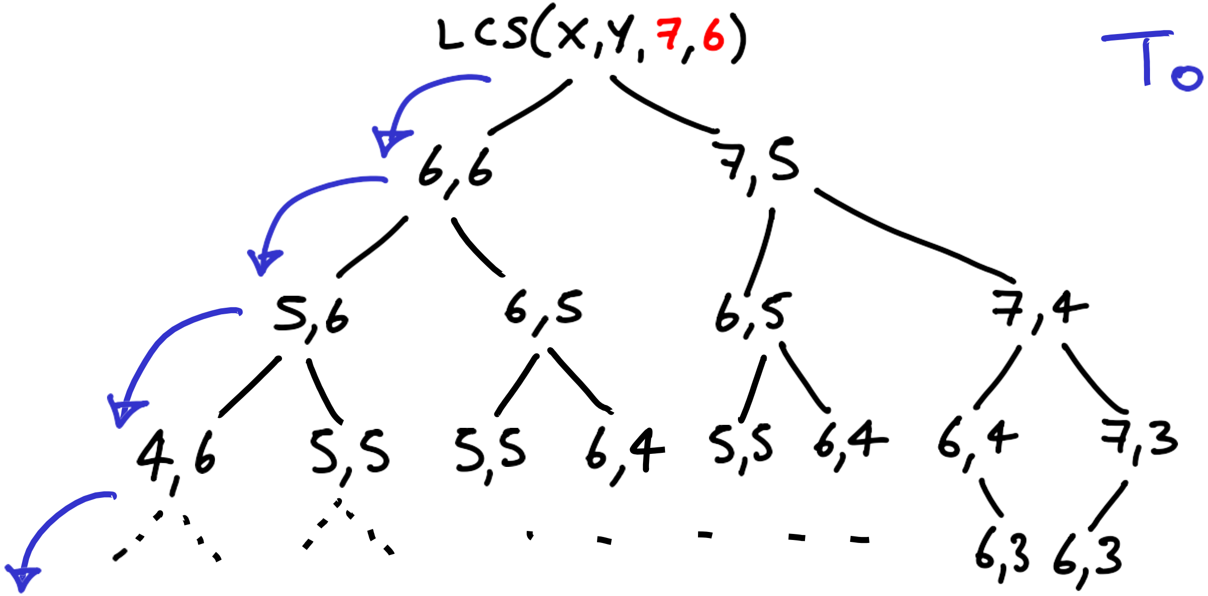
else $c[i,j] \leftarrow \max\{LCS(X,Y,i,j-1), LCS(X,Y,i-1,j)\}$

return $c[i,j]$ \\ look up

Memoization

Make "memos" of solutions
(to subproblems)

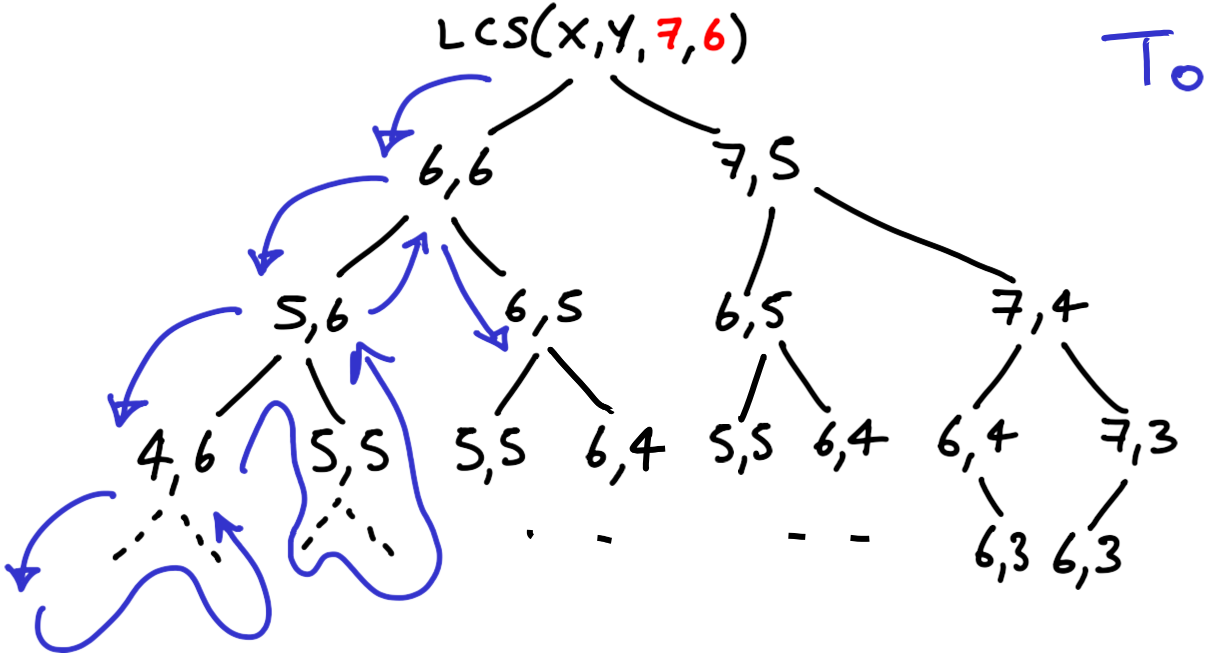
Top-down



Memoization

Make "memos" of solutions (to subproblems)

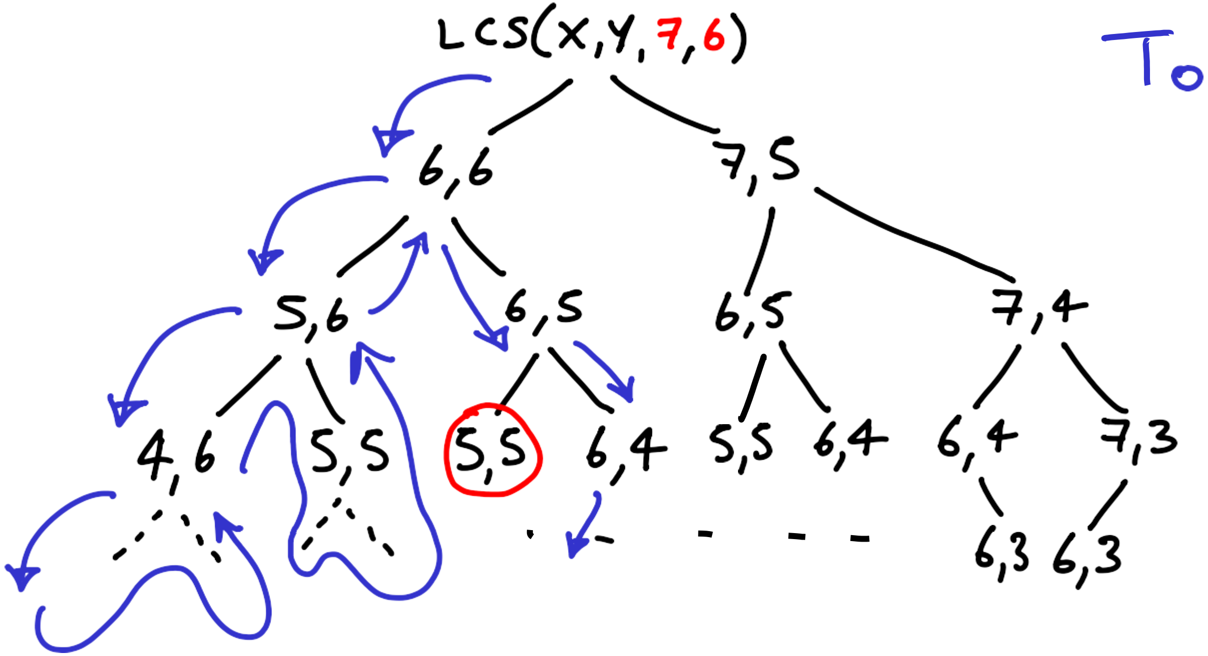
Top-down



Memoization

Make "memos" of solutions (to subproblems)

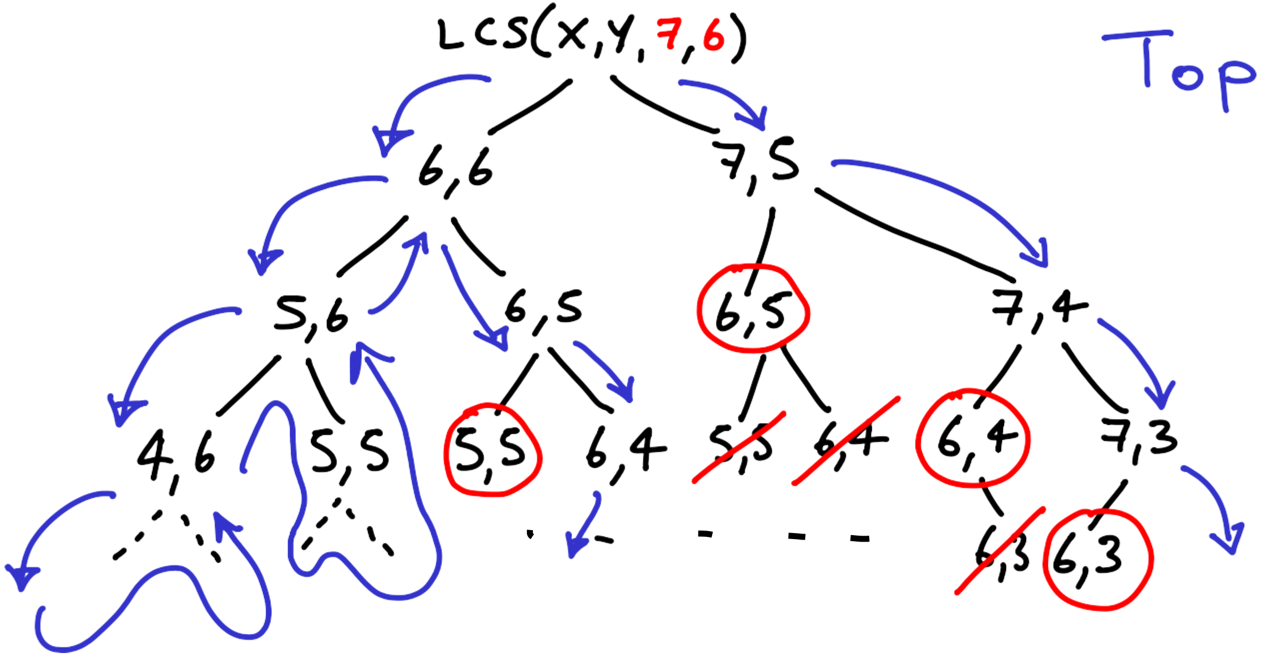
Top-down



Memoization

Make "memos" of solutions (to subproblems)

Top-down



Dynamic programming

DYNAMIC PROGRAMMING

bottom-up

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

base cases

DYNAMIC PROGRAMMING

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

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					
D	0							
C	0							
A	0							
B	0							
A	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

DYNAMIC PROGRAMMING

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

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 + 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	
D	0							
C	0							
A	0							
B	0							
A	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

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							
C	0							
A	0							
B	0							
A	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

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							
A	0							
B	0							
A	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

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						
A	0	1						
B	0	1						
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

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				
A	0	1	1					
B	0	1						
A	0	1						

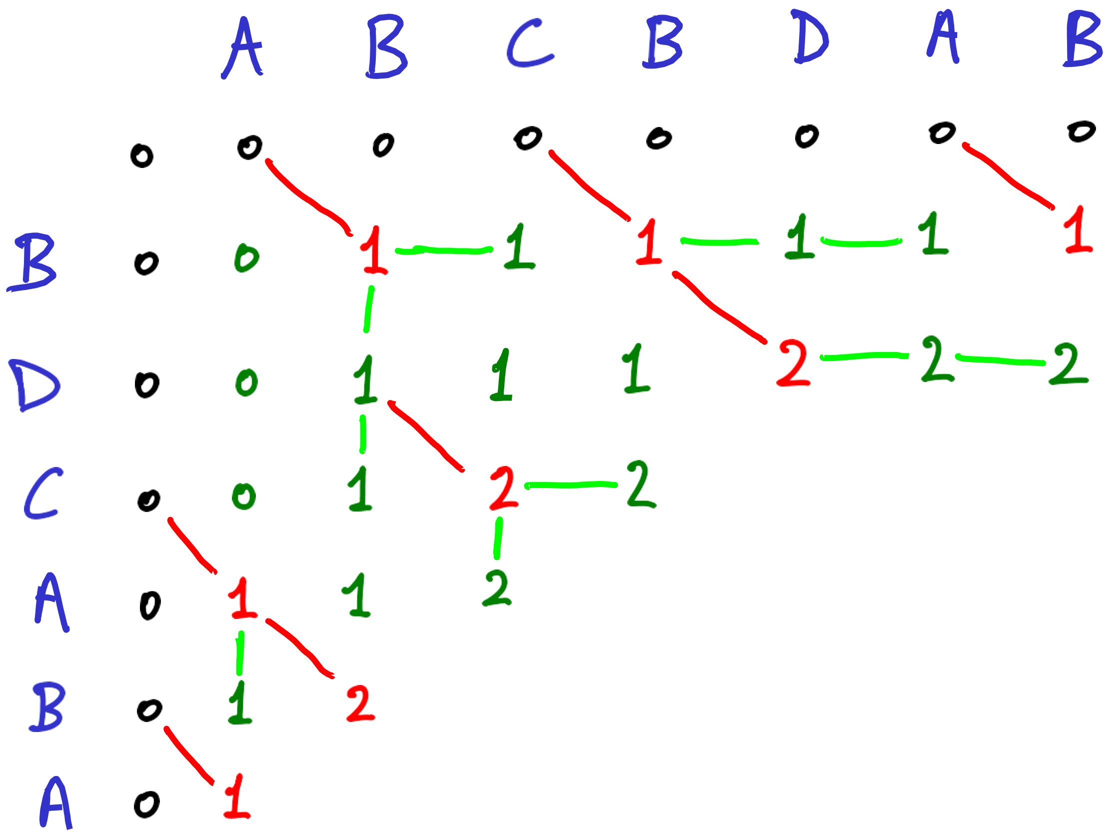
red # : 1 + diag↖#

when letters in column & row of # match

green # : max of {above, left}

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

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
	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	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

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	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

← BCBA

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

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	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

BDAB

BCBA

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

		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	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

BDAB
BCAB

BCBA

red # : 1 + diag↖ #

when letters in column & row of # match

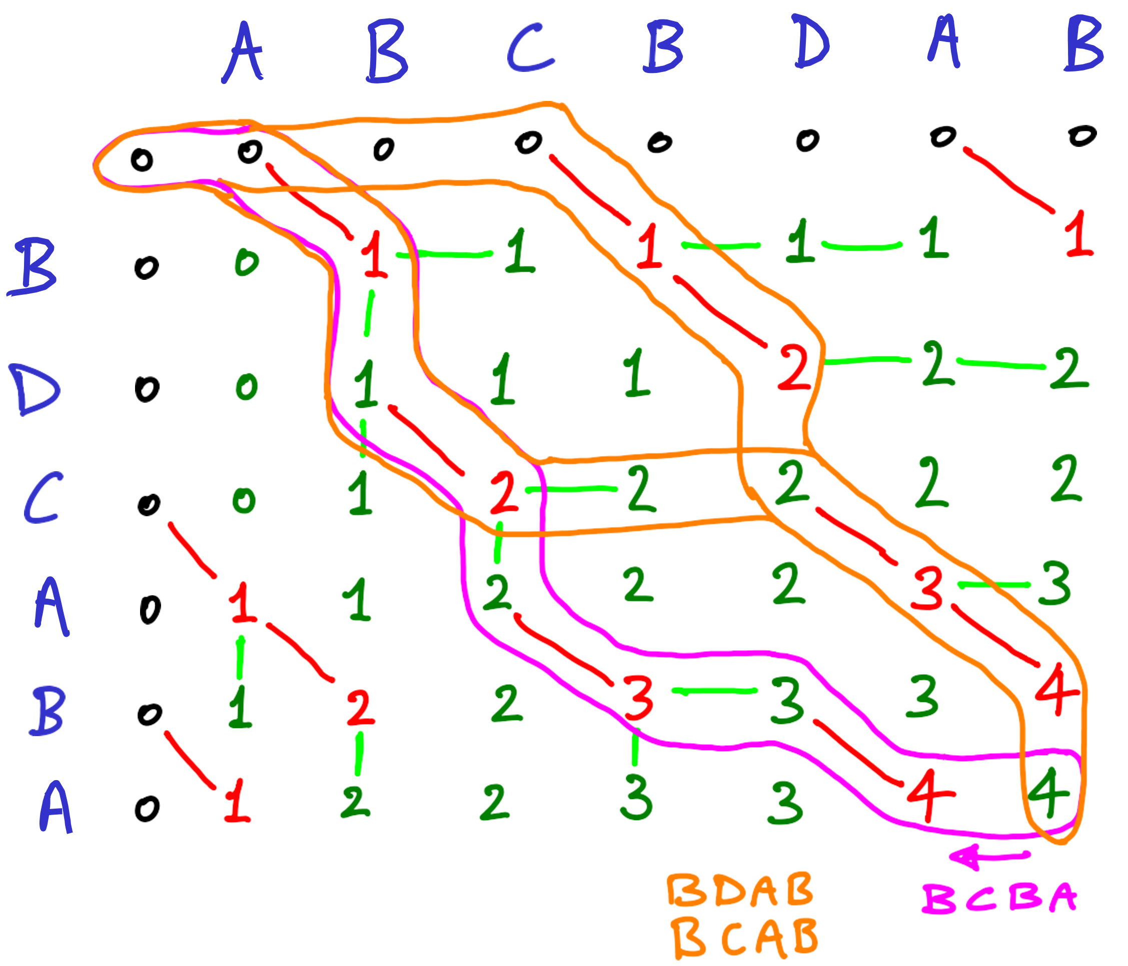
green # : max of {above, left}

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 + diag↖ #

when letters in column & row of # match

green # : max of {above, left}

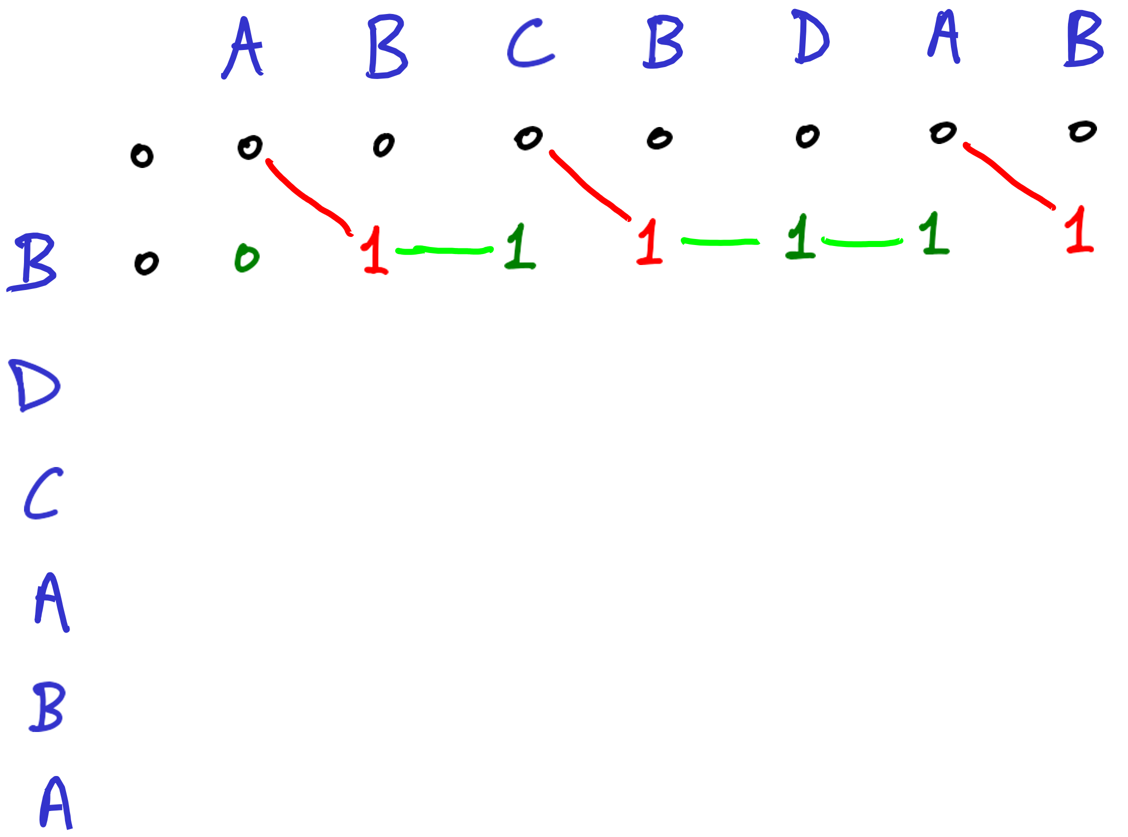
when letters in column & row of # don't match

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

↳ follow mandatory paths ; optional branches : multiple solutions

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

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

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	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
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

$\Theta(mn)$ 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								
B								
A								

⋮
etc

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)

Save space: min{m,n}

DYNAMIC PROGRAMMING

		A	B	C	B	D	A	B
B								
D								
C					etc			
A					⋮			
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

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)

Save space: min{m,n}

get |LCS| but not LCS