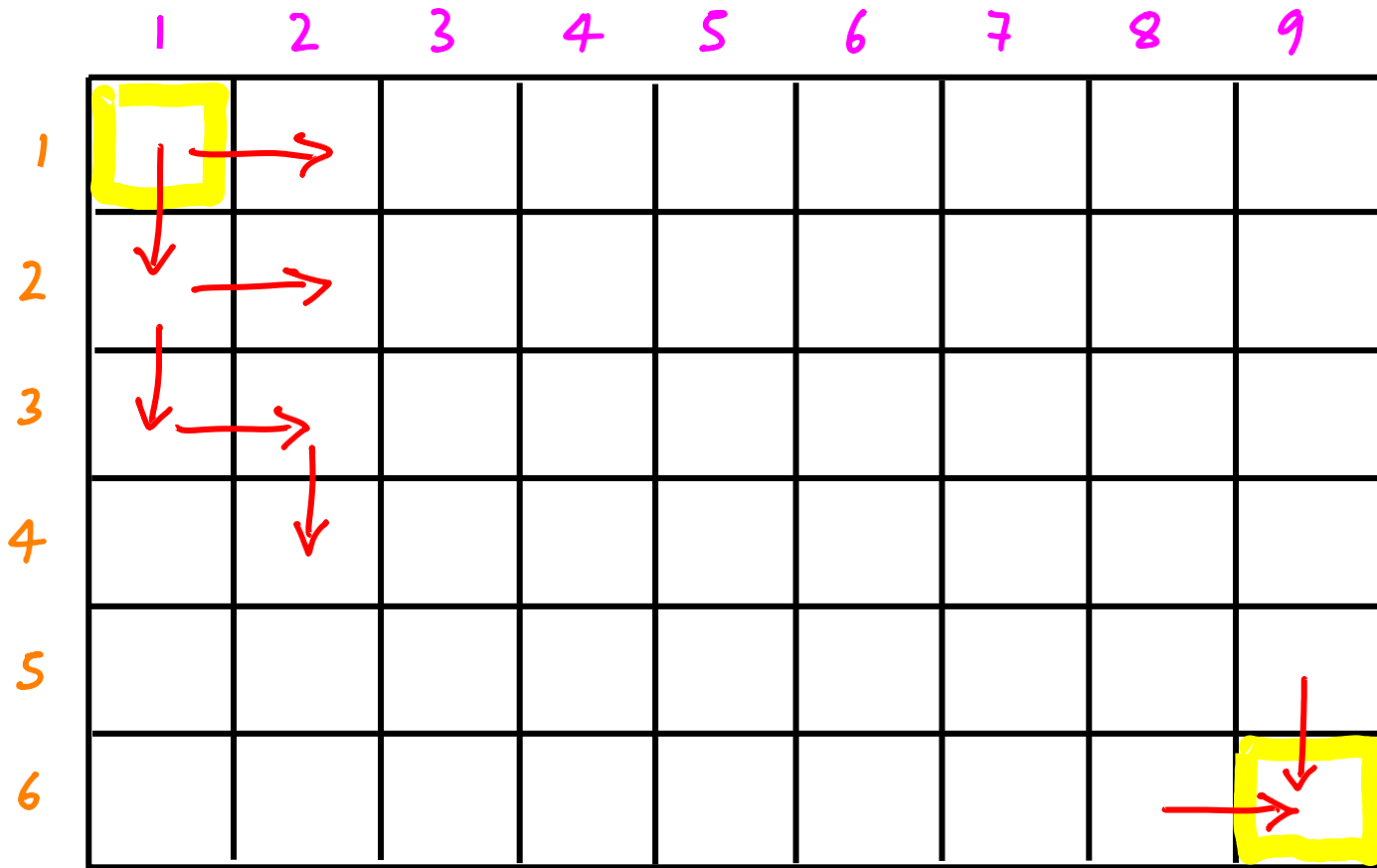


Starting at top-left of $n \times m$ grid, moving only down or right,
how many ways to reach bottom-right?

Recursive form?

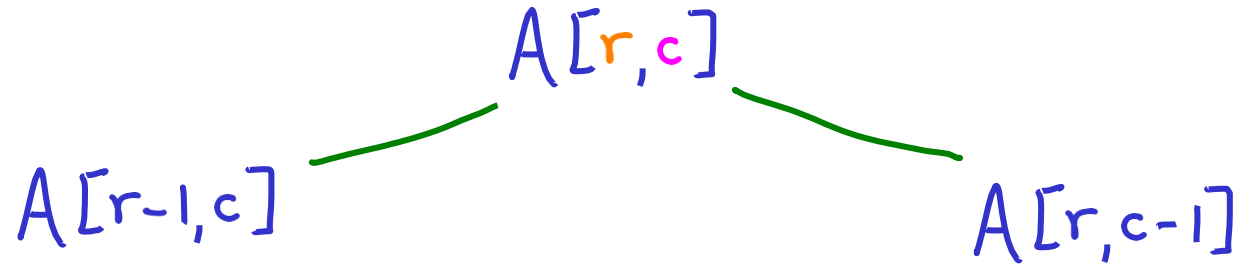


Starting at top-left of $n \times m$ grid, moving only down or right,
how many ways to reach bottom-right?

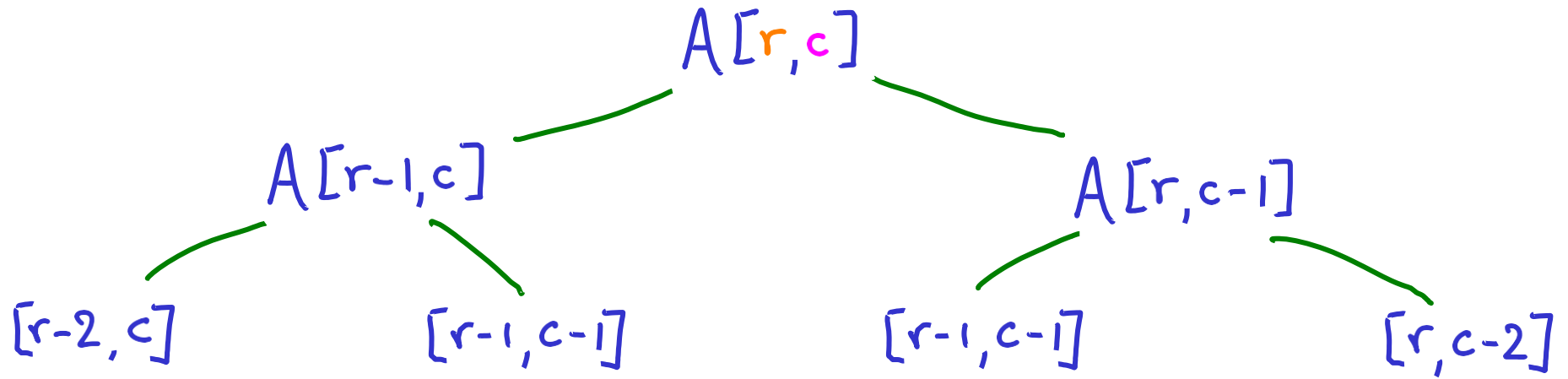
$$A[r, c] = A[r-1, c] + A[r, c-1]$$

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

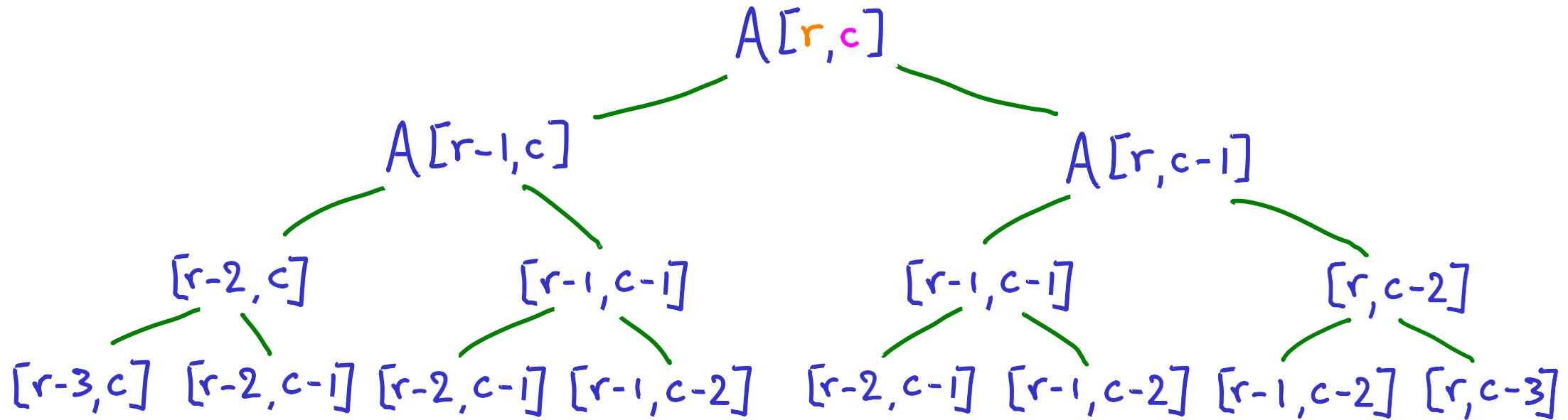
Starting at top-left of $n \times m$ grid, moving only down or right,
how many ways to reach bottom-right?



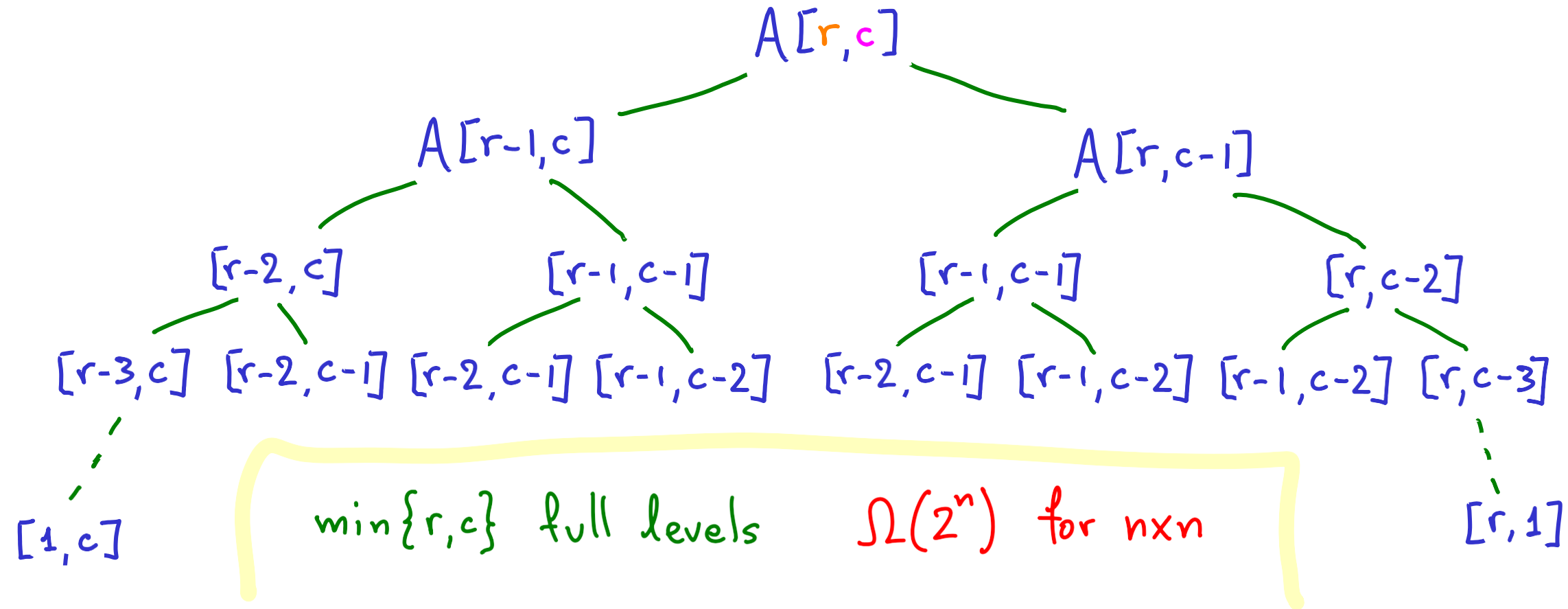
Starting at top-left of $n \times m$ grid, moving only down or right,
how many ways to reach bottom-right?



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



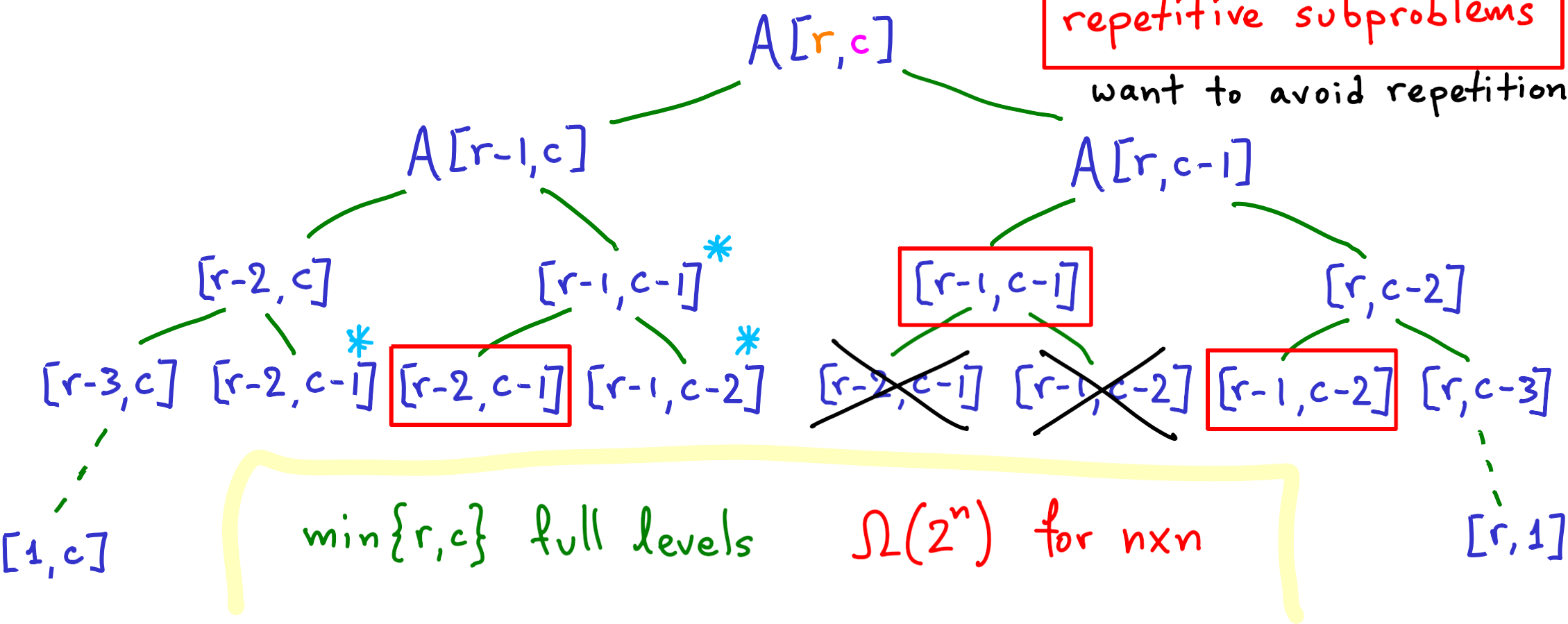
Starting at top-left of $n \times m$ grid, moving only down or right, how many ways to reach bottom-right?



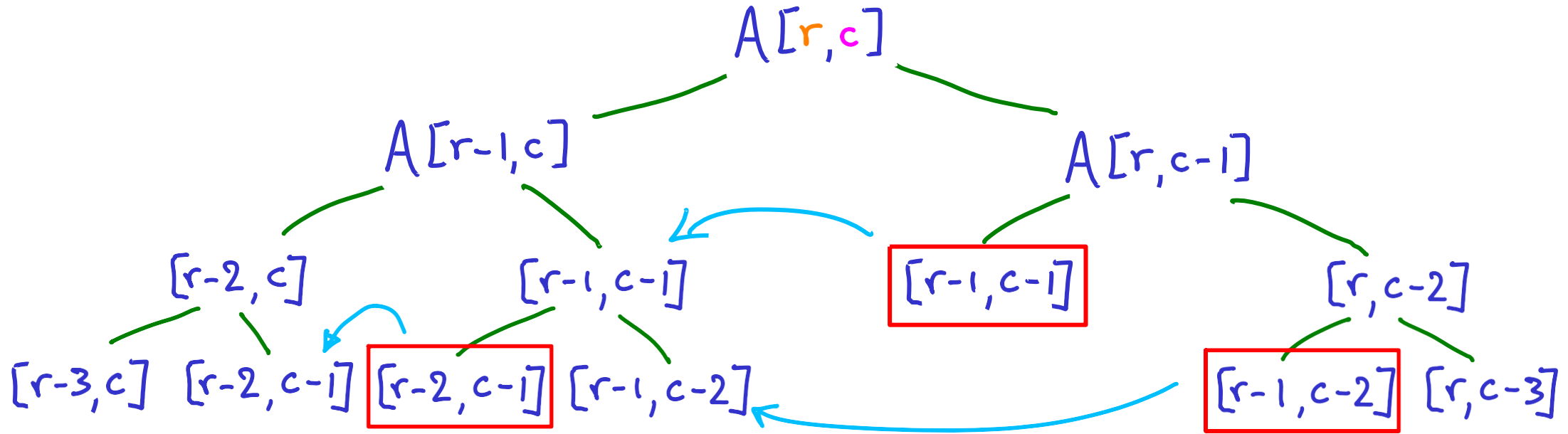
Starting at top-left of $n \times m$ grid, moving only down or right,
 how many ways to reach bottom-right?

repetitive subproblems

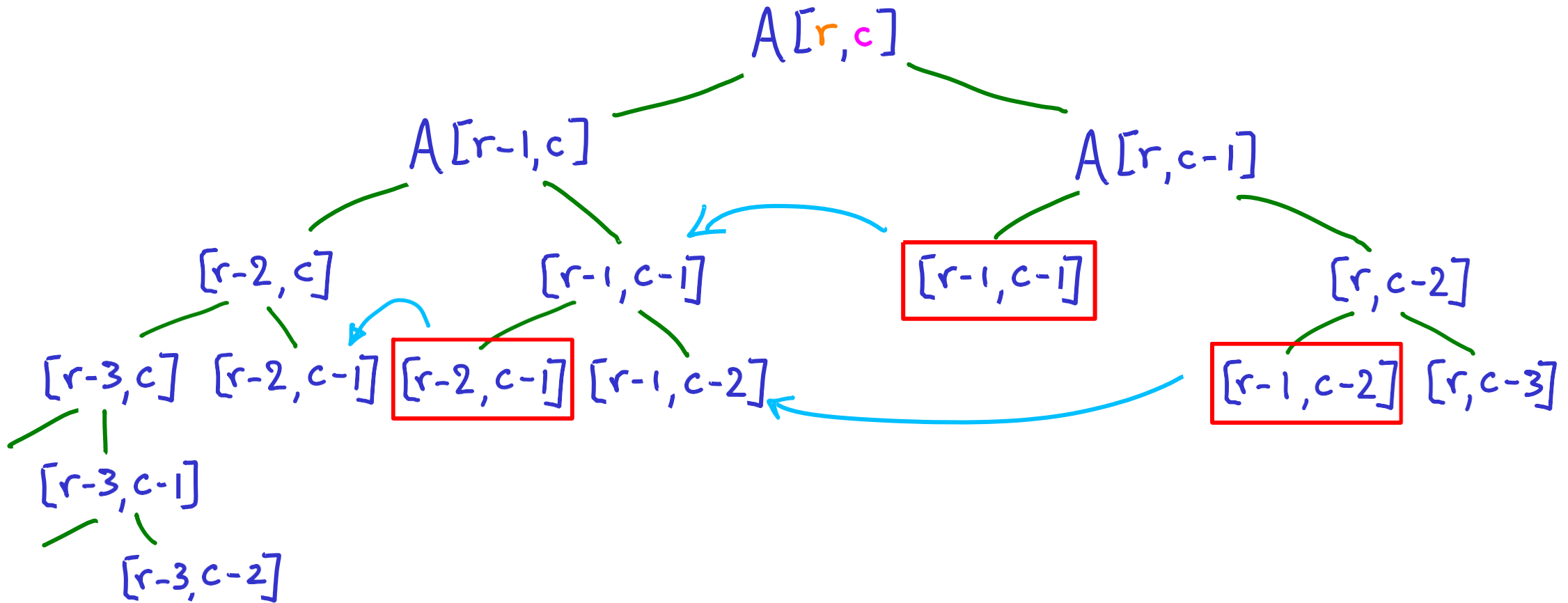
want to avoid repetition



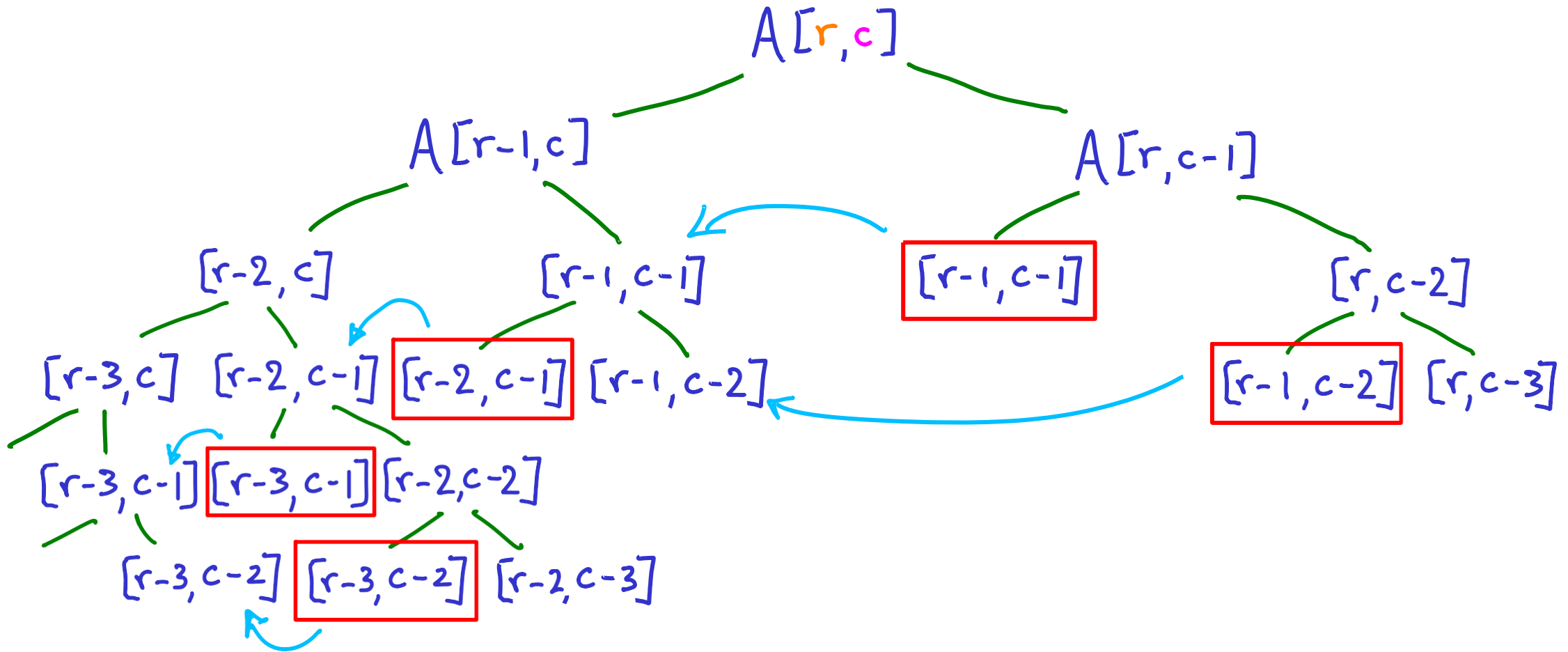
Starting at top-left of $n \times m$ grid, moving only down or right, how many ways to reach bottom-right?



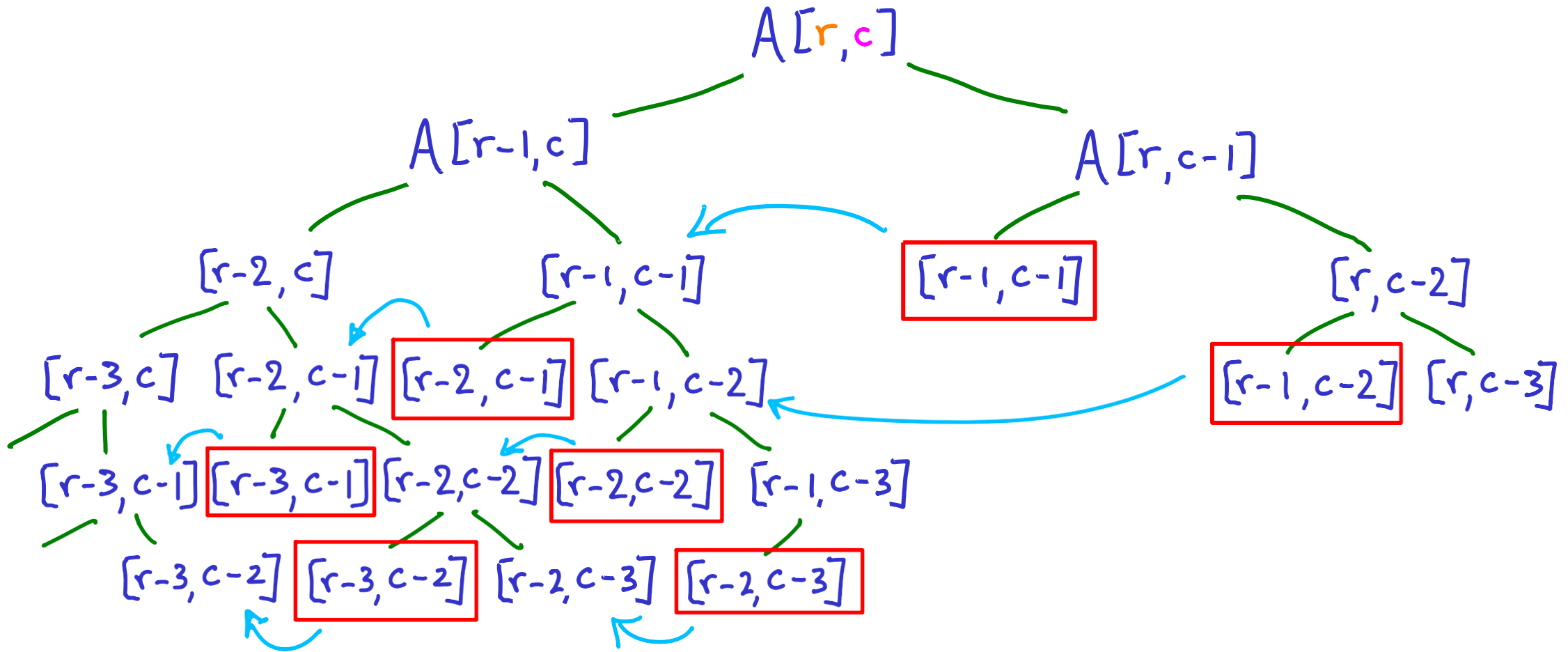
Starting at top-left of $n \times m$ grid, moving only down or right, how many ways to reach bottom-right?



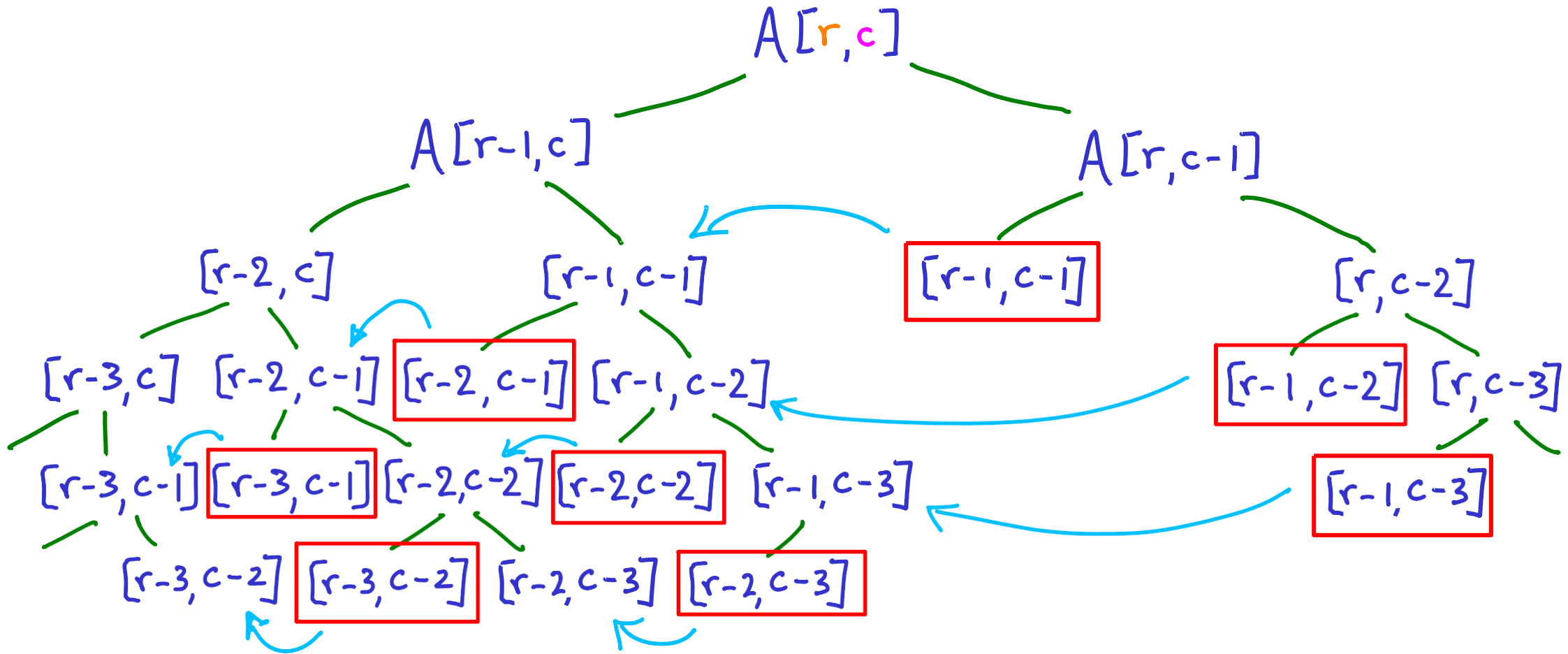
Starting at top-left of $n \times m$ grid, moving only down or right, how many ways to reach bottom-right?



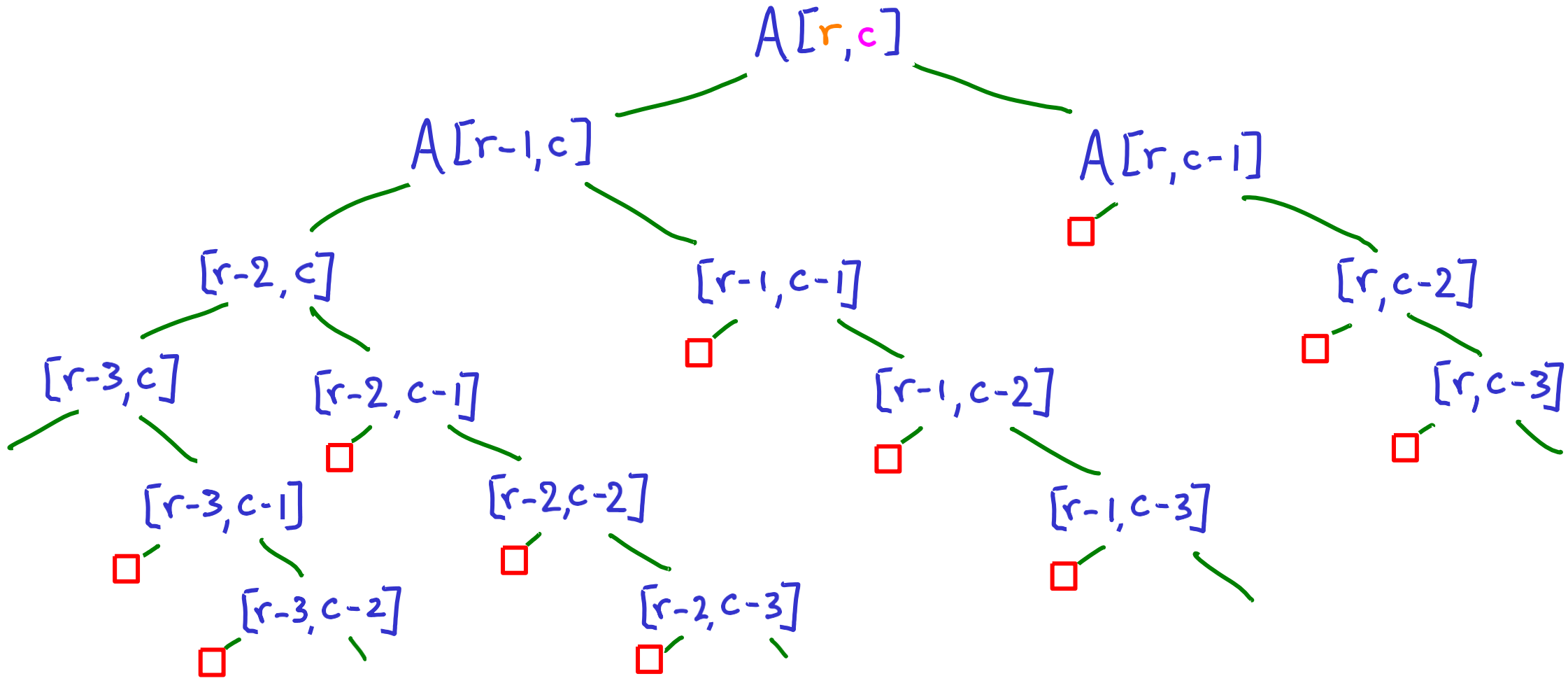
Starting at top-left of $n \times m$ grid, moving only down or right, how many ways to reach bottom-right?



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

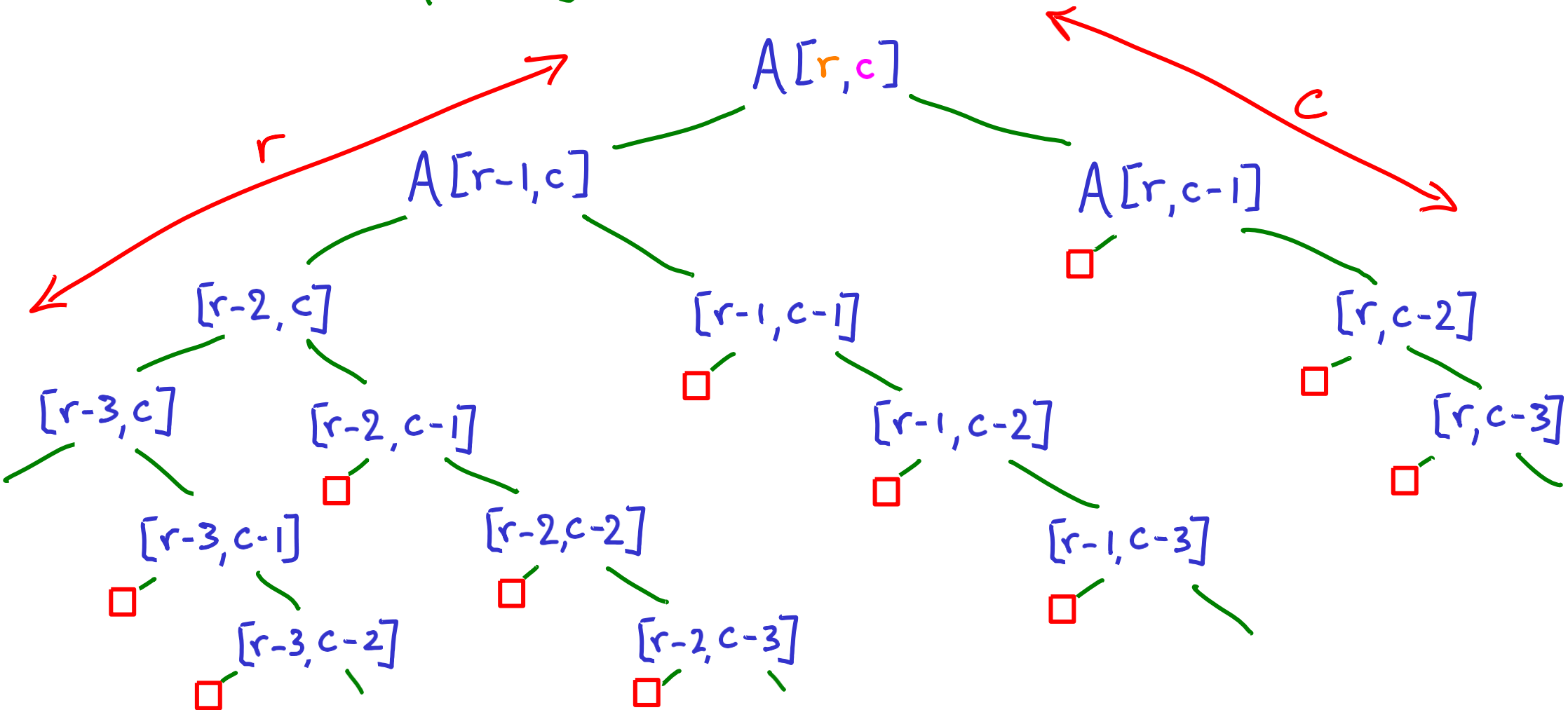


How many times will we recurse in a unique way?



How many times will we recurse in a unique way?

→ r.c distinct subproblems



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									
2									
3									
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

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

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

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

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

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

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

The cell at row 6, column 9 is highlighted in yellow and labeled r, c . Green arrows point upwards from the cell at row 6, column 9 to the cells at rows 1, 2, 3, 4, and 5, column 9.

Recursion:

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

MEMOIZATION (making memos)

For this problem, $m \times n$ table

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

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

The cell at row 6, column 9 is highlighted in yellow and labeled 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

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

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

The cell at row 6, column 9 is highlighted in yellow and labeled 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

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

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

Diagram illustrating a 6x9 grid representing a memoization table. The columns are labeled 1 through 9, and the rows are labeled 1 through 6. The cell at row 6, column 9 is highlighted and labeled r, c . Green arrows indicate the recursive dependencies: vertical arrows point up from row 6 to row 1 in columns 7, 8, and 9; horizontal arrows point left from column 9 to column 7 in rows 1 and 2.

Recursion:

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

MEMOIZATION (making memos)

For this problem, $m \times n$ table

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

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

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

	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									r,c

Diagram illustrating a 6x9 table for memoization. The table is filled with values. The first row (row 1) contains 1s in columns 2 through 9. The second row (row 2) contains 1 in column 1 and 2 in column 2. Green arrows indicate the dependencies: vertical arrows point up from row 2 to row 1 for columns 2-9, and horizontal arrows point left from column 2 to column 1 for row 2. The cell at row 6, column 9 is highlighted and labeled '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

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

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

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

	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 $A[r-1,c]$ ↑
then find $A[r,c-1]$ ←

MEMOIZATION (making memos)

For this problem, $m \times n$ table

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

	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

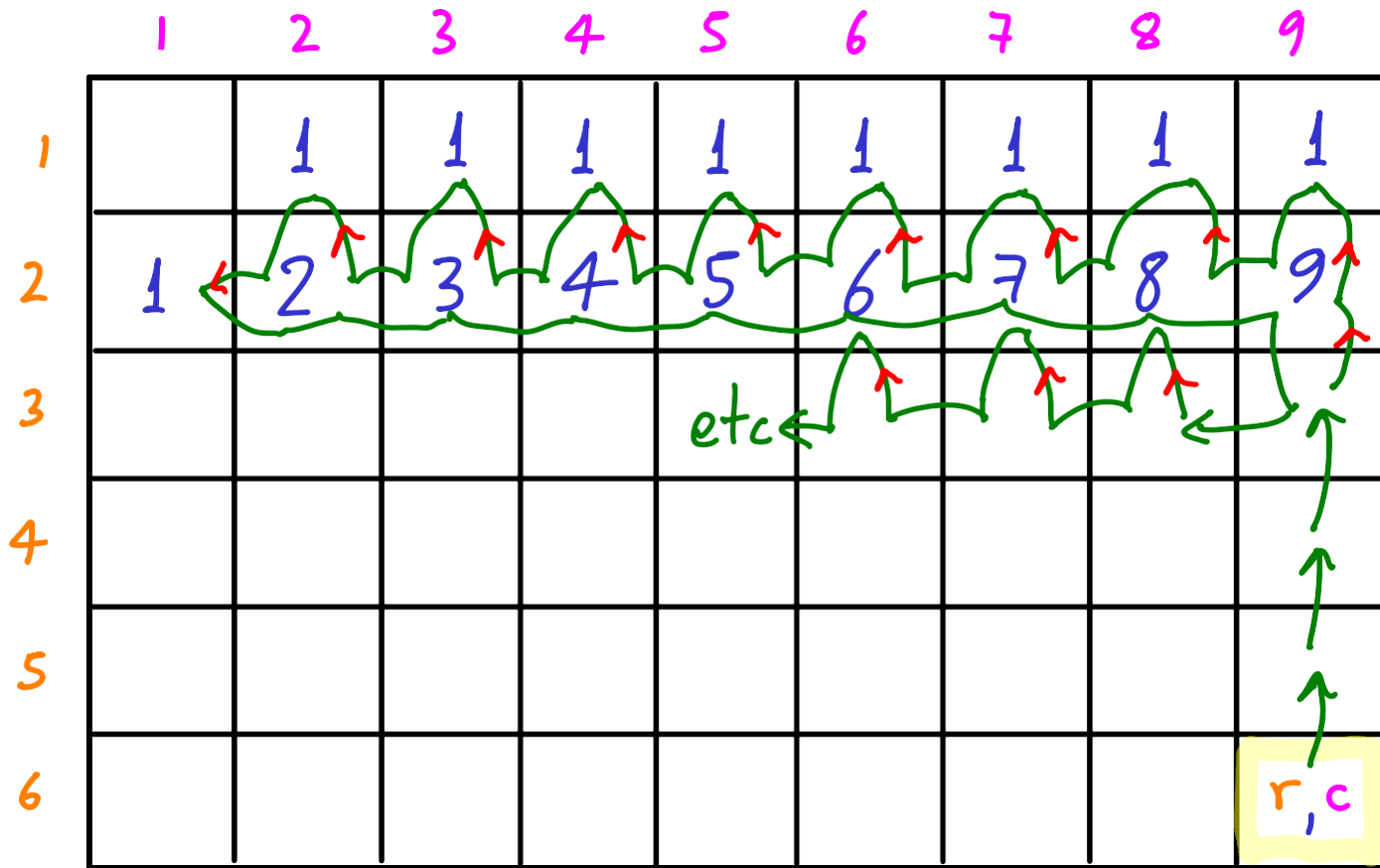
Recursion:

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 $A[r, c-1]$ ←

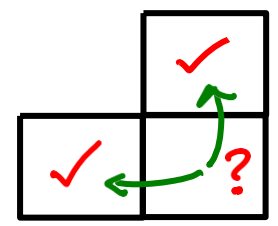
$\Theta(n \cdot m)$
time & space

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								

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



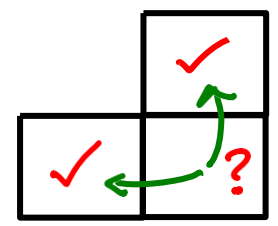
fill any cell as long as what it depends on is full

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	1								
4	1								
5	1								
6	1								

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



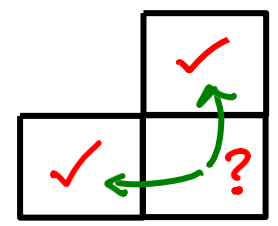
fill any cell as long as what it depends on is full

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								

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



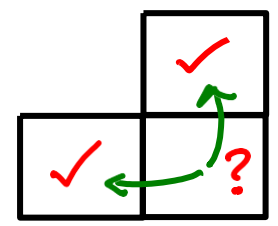
fill any cell as long as what it depends on is full

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								

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



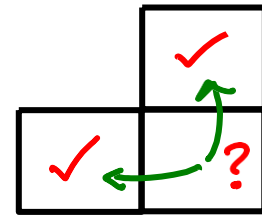
fill any cell as long as what it depends on is full

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								

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



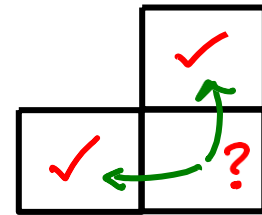
fill any cell as long as
what it depends on is full

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							

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



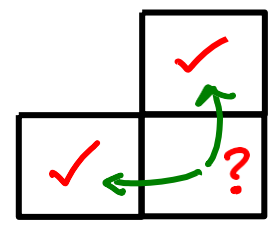
fill any cell as long as
what it depends on is full

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

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




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



The diagram illustrates the sequence 23, 3, 5, 18, 10, 101, 12, 14, 4, 105. Two green arrows are drawn below the numbers. The first arrow starts at 23 and points to 101. The second arrow starts at 101 and points to 105. This indicates that the longest increasing subsequence is 23, 101, 105.

DYNAMIC PROGRAMMING - LONGEST INCREASING SUBSEQUENCE

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



The diagram illustrates the sequence 23, 3, 5, 18, 10, 101, 12, 14, 4, 105. Green arrows connect the elements 3, 5, 18, 101, and 105, showing an increasing subsequence. The arrows are: 3 to 5, 5 to 18, 18 to 101, and 101 to 105.


DYNAMIC PROGRAMMING - LONGEST INCREASING SUBSEQUENCE

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

The diagram illustrates the sequence 23, 3, 5, 18, 10, 101, 12, 14, 4, 105. Green arrows are drawn below the numbers, starting from 3 and pointing to 5, then from 5 to 10, then from 10 to 12, then from 12 to 14, and finally from 14 to 105. This sequence of arrows highlights the longest increasing subsequence: 3, 5, 10, 12, 14, 105.

DYNAMIC PROGRAMMING - LONGEST INCREASING SUBSEQUENCE


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



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

DYNAMIC PROGRAMMING - LONGEST INCREASING SUBSEQUENCE

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




$L(s) = 3, 5, 10, 12, 14, 105$ $|L(s)| = 6$

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

1 2 3 n-1 n
23 , 3 , 5 , 18 , 10 , 101 , 12 , 14 , 4 , 105

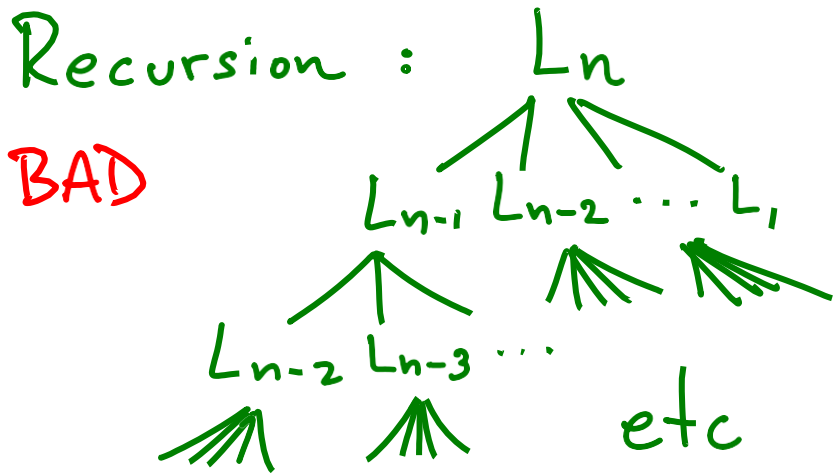


$|L_{n-1}| = 2$

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

look at all L_j ($j < n$)

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



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.

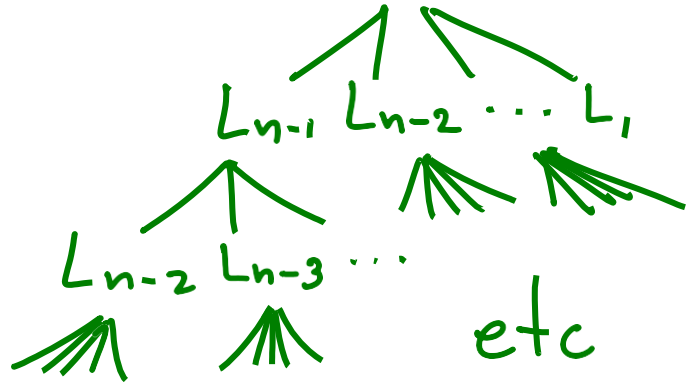
23, 3, 5, 18, 10, 101, 12, 14, 4

1
|L₁|

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

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.

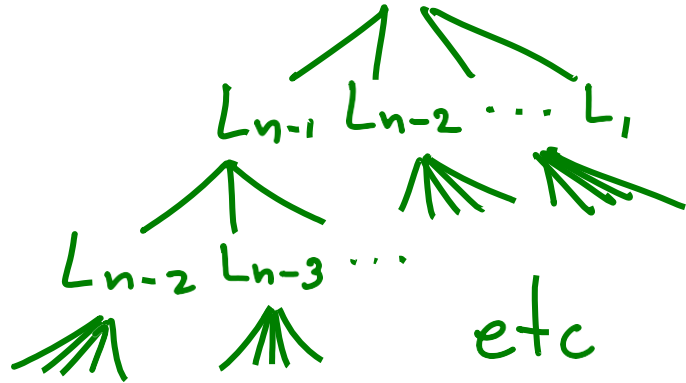
23, 3, 5, 18, 10, 101, 12, 14, 4

1 1
 |L₂|

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

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.

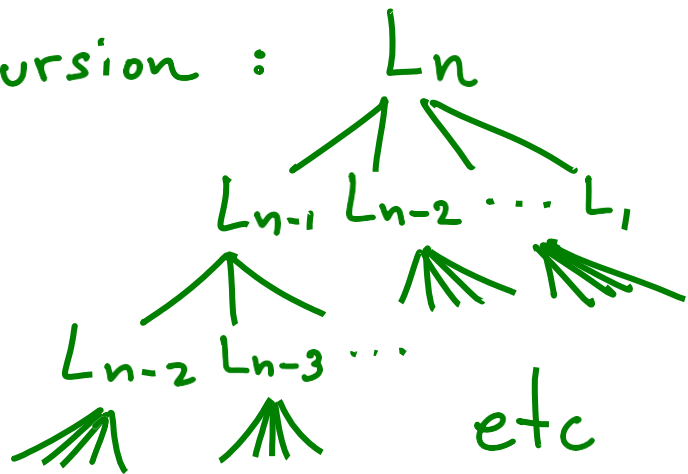
23, 3, 5, 18, 10, 101, 12, 14, 4

1 1 2

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

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.

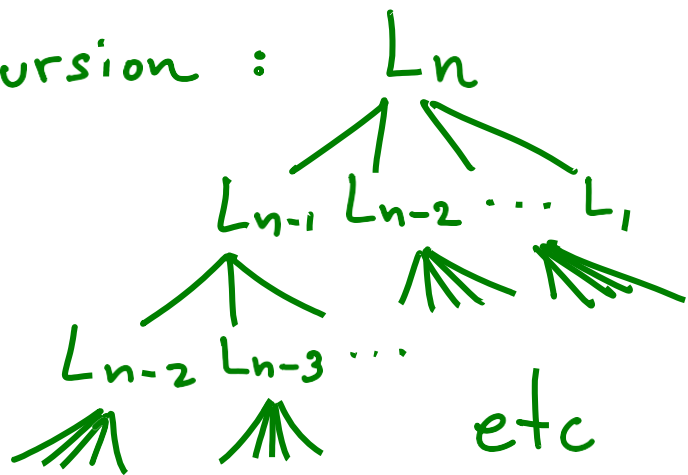
23, 3, 5, 18, 10, 101, 12, 14, 4

1 1 2 3

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

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.

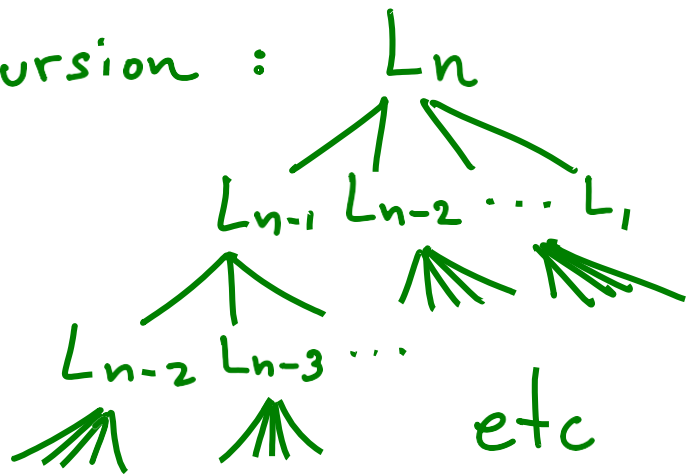
23, 3, 5, 18, 10, 101, 12, 14, 4

1 1 2 3 3

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

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.

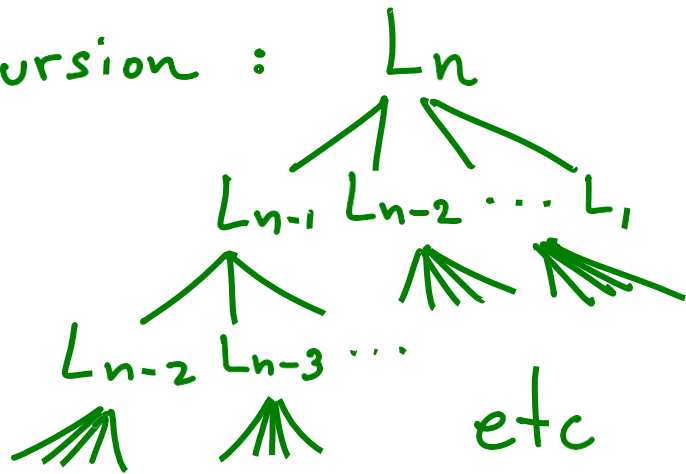
23, 3, 5, 18, 10, 101, 12, 14, 4

1, 1, 2, 3, 3, 4

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

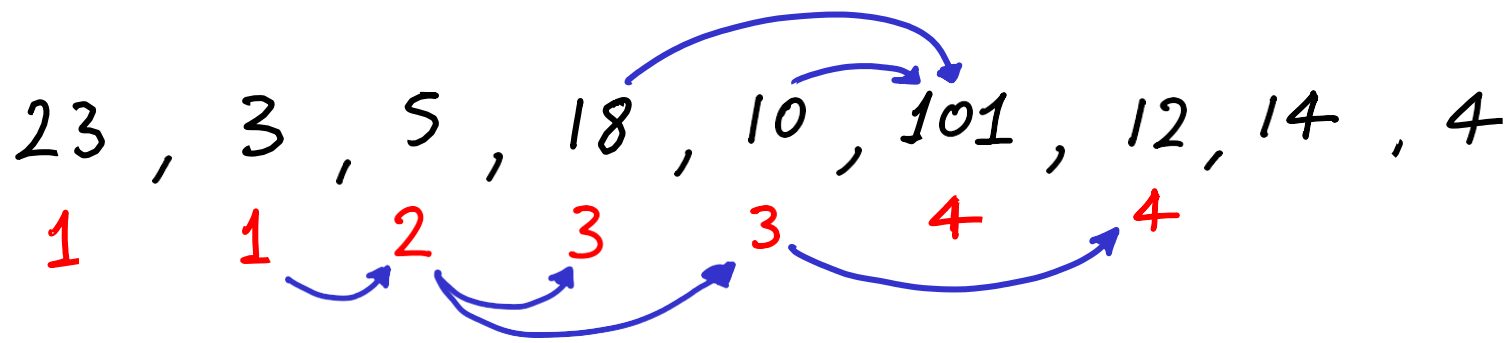
Recursion :

BAD



Dyn. Prog: Build solutions, "bottom up"

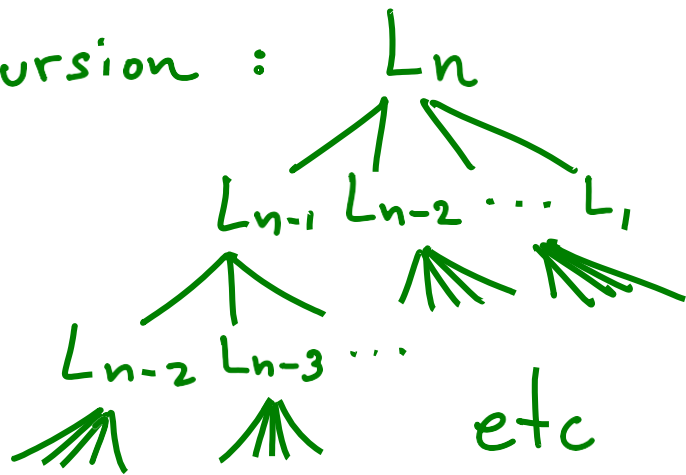
When it's time to solve $|L_k|$ we have stored all $|L_j|$ ($j < k$) in an array.



$$|L_n| = 1 + \text{MAX}_{\{j \text{ st. } S[j] < S[n]\}} |L_j|$$

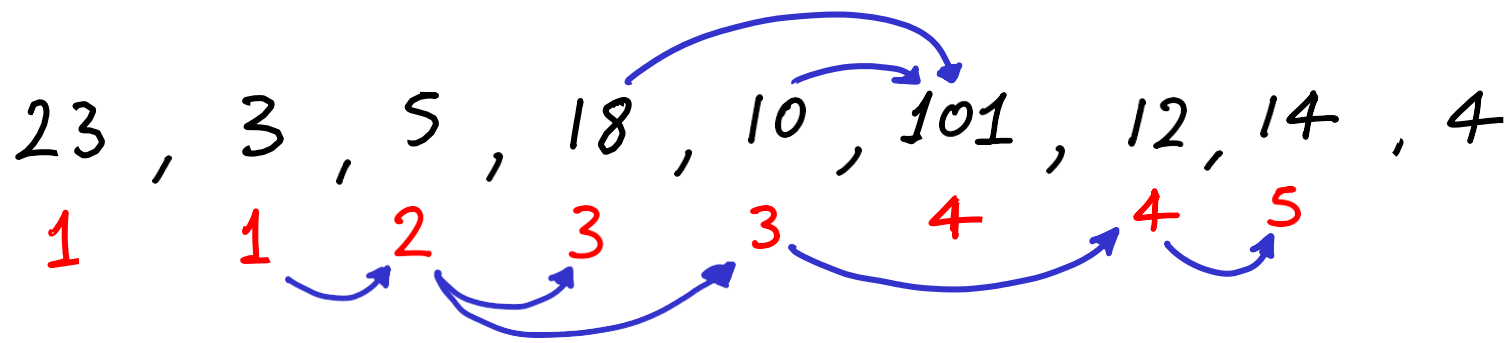
Recursion :

BAD



Dyn. Prog: Build solutions, "bottom up"

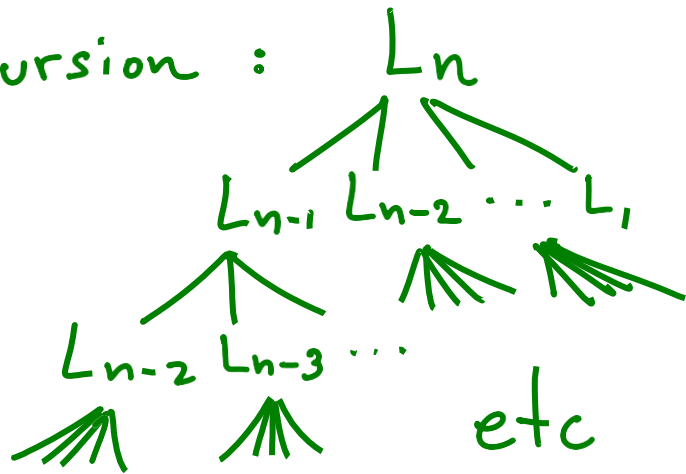
When it's time to solve $|L_k|$ we have stored all $|L_j|$ ($j < k$) in an array.



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

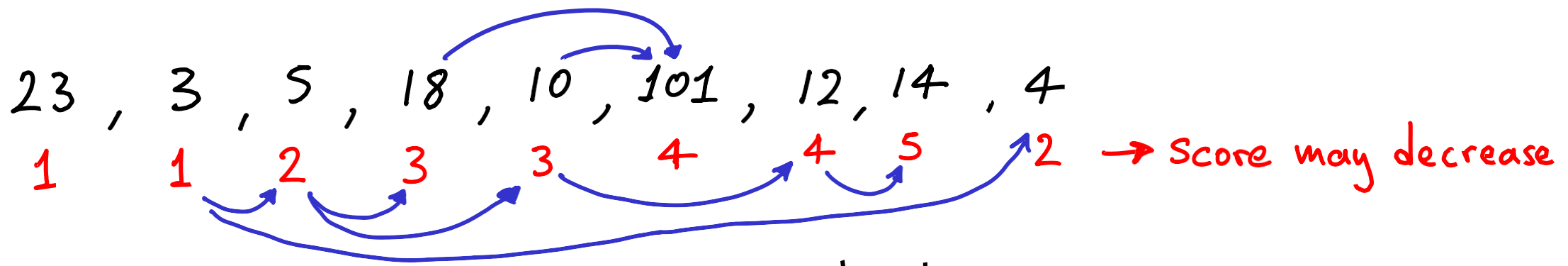
Recursion :

BAD



Dyn. Prog:

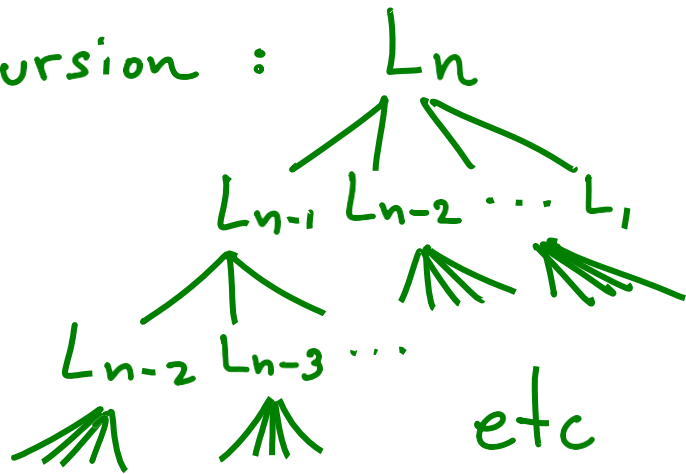
Build solutions, "bottom up"
 When it's time to solve $|L_k|$ we have stored all $|L_j|$ ($j < k$) in an array.



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

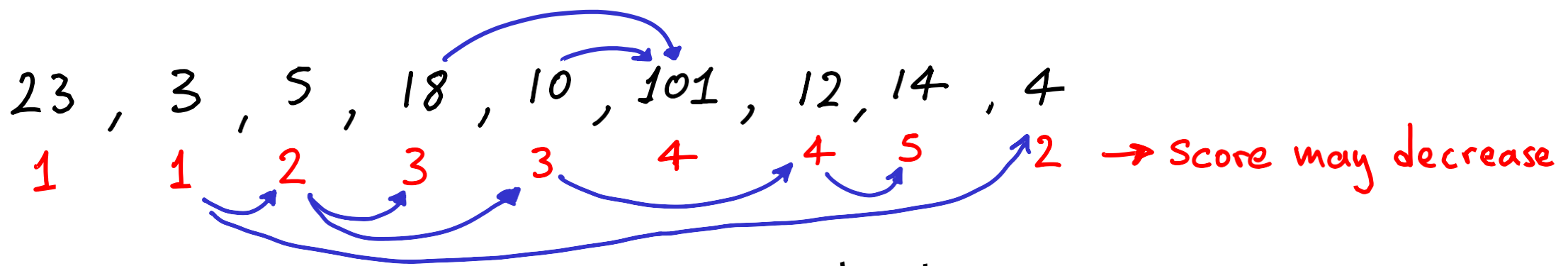
Recursion :

BAD

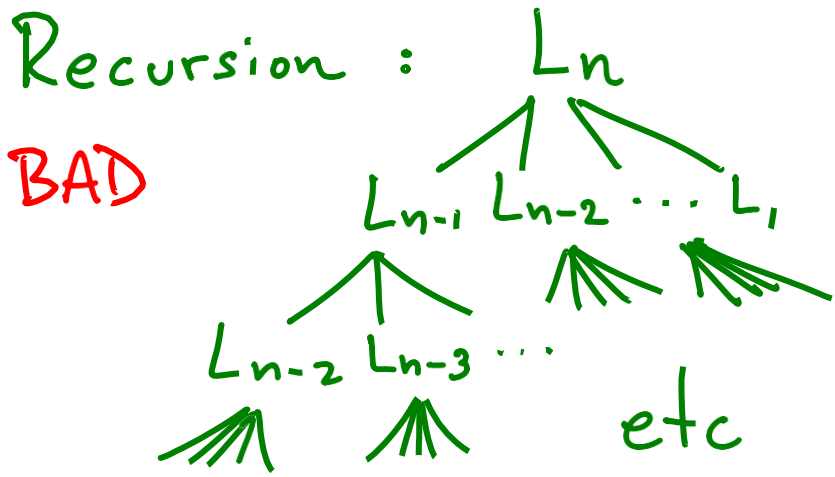


Dyn. Prog: Build solutions, "bottom up"

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



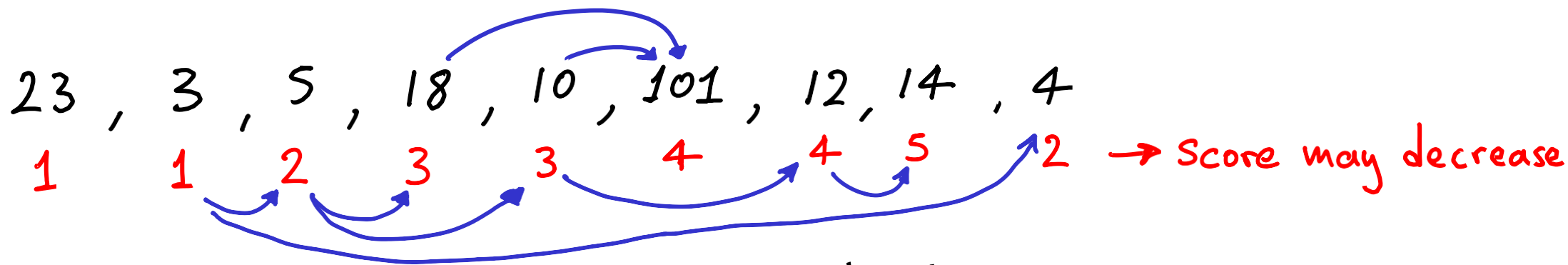
$$|L_n| = 1 + \text{MAX}_{\{ \text{all } j \text{ s.t. } S[j] < S[n] \}} |L_j|$$



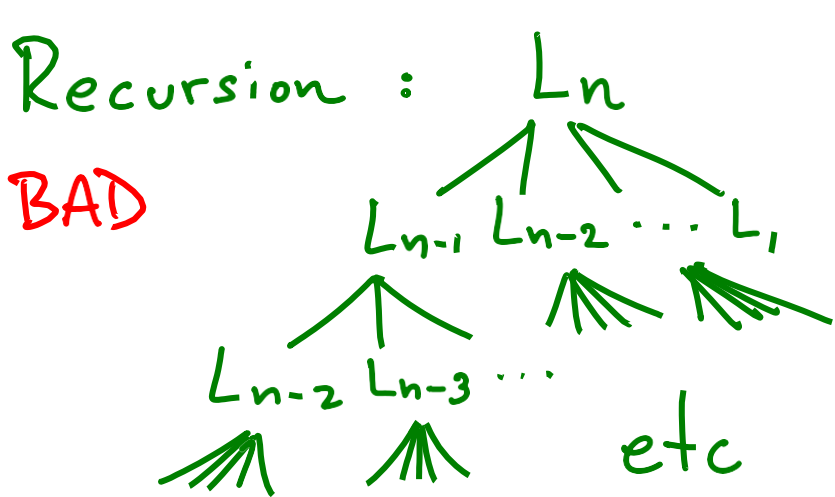
Dyn. Prog: Build solutions, "bottom up"

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

time? space?



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



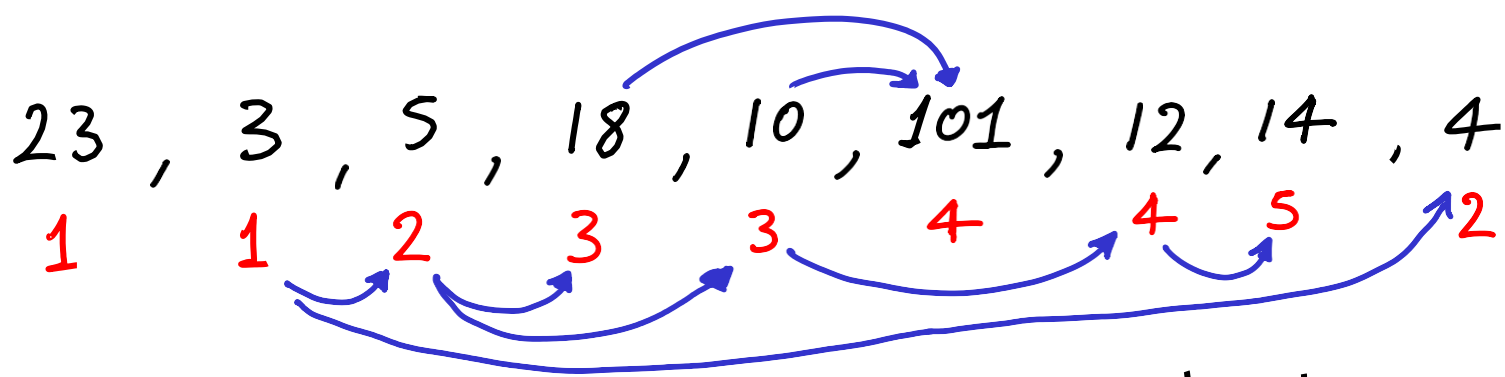
Dyn. Prog: Build solutions, "bottom up"

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

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

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

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



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

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

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

LONGEST COMMON SUBSEQUENCE

&

DYNAMIC PROGRAMMING

LONGEST COMMON SUBSEQUENCE

&

DYNAMIC PROGRAMMING

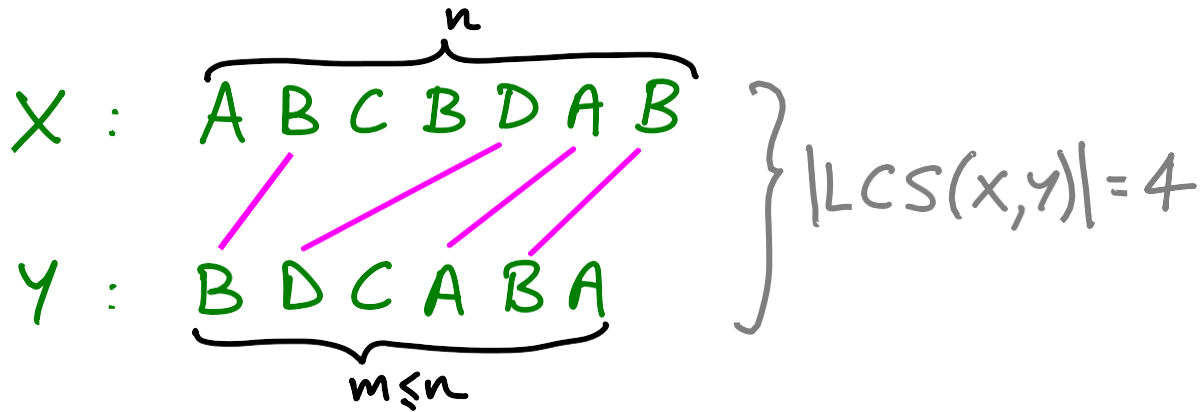
X : $\overbrace{A B C B D A B}^n$

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

LONGEST COMMON SUBSEQUENCE

&

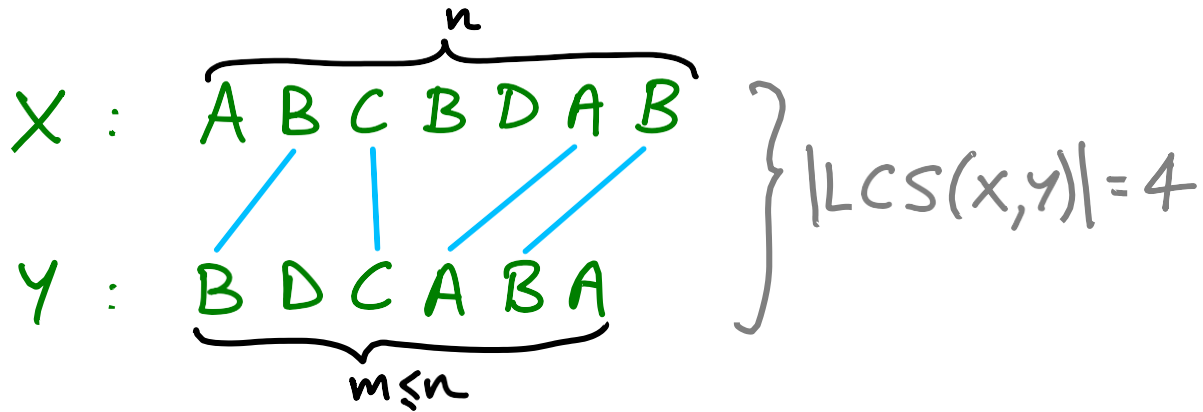
DYNAMIC PROGRAMMING



LONGEST COMMON SUBSEQUENCE

&

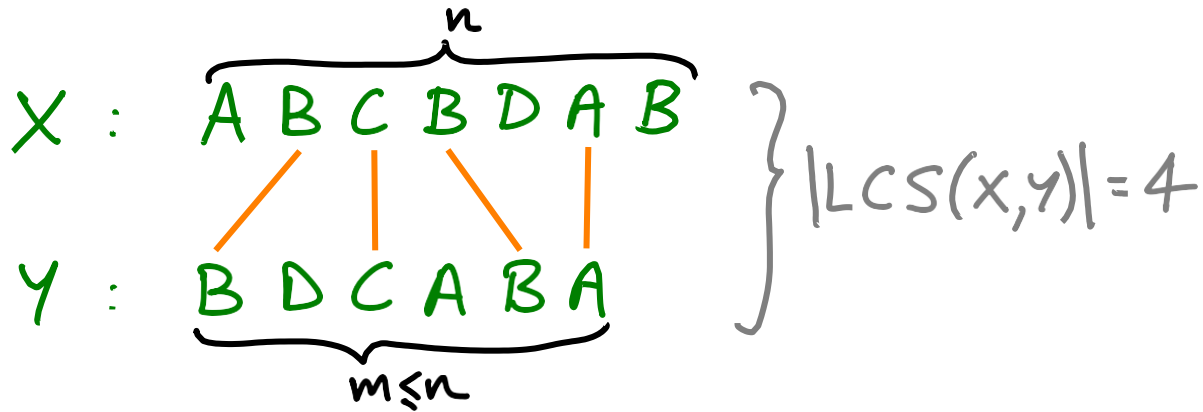
DYNAMIC PROGRAMMING



LONGEST COMMON SUBSEQUENCE

&

DYNAMIC PROGRAMMING



LONGEST COMMON SUBSEQUENCE

&

DYNAMIC PROGRAMMING

X : A B C B D A B

Y : B D C A B A

n

$m \leq n$

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

Brute force to find LCS:
for every subsequence of Y

$\theta(2^m)$

LONGEST COMMON SUBSEQUENCE

&

DYNAMIC PROGRAMMING

X : A B C B D A B
Y : B D C A B A

n
 $m \leq n$

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

$\theta(2^m)$

Brute force to find LCS:

for every subsequence of Y
check if it exists in X
 $\hookrightarrow O(n)$: easy

LONGEST COMMON SUBSEQUENCE

&

DYNAMIC PROGRAMMING

X : A B C B D A B
Y : B D C A B A

n
 $m \leq n$

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

Brute force to find LCS:

for every subsequence of Y
check if it exists in X

$\hookrightarrow O(n)$: easy

$\theta(2^m)$

$\rightarrow O(n \cdot 2^m)$

Finding |LCS|

$$c(i,j) = |LCS(x[1...i], y[1...j])|$$

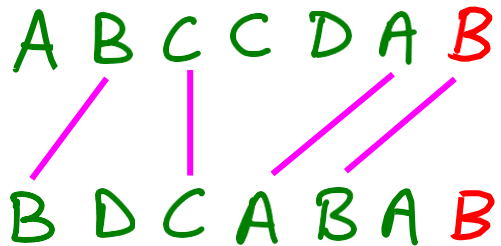
Finding |LCS|

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

Finding |LCS|

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

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



Finding |LCS|

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

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



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

Finding |LCS|

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

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

Slide last match over:
just as good

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

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

Finding |LCS|

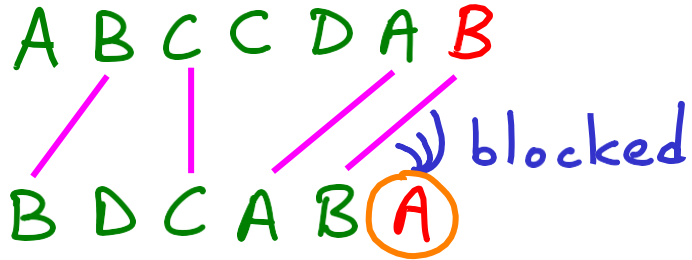
$$c(i, j) = |\text{LCS}(x[1 \dots i], y[1 \dots j])| = \begin{cases} c(i-1, j-1) + 1 & \text{if } x[i] = y[j] \\ \max\{c(i, j-1), c(i-1, j)\} & \text{otherwise} \end{cases}$$

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 $Y[j]$

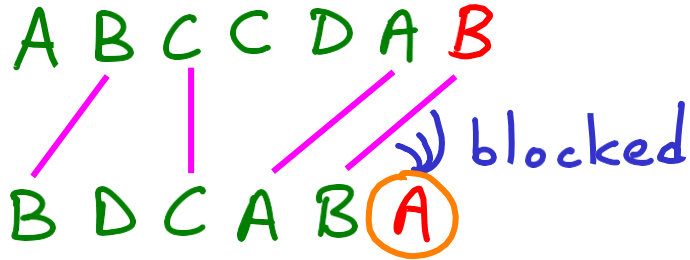


Finding |LCS|

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

$LCS(x[1...i], y[1...j])$

cannot use both $x[i]$ and $y[j]$



Hide each

A B C C D A B

B D C A B ~~A~~

A B C C D A ~~B~~

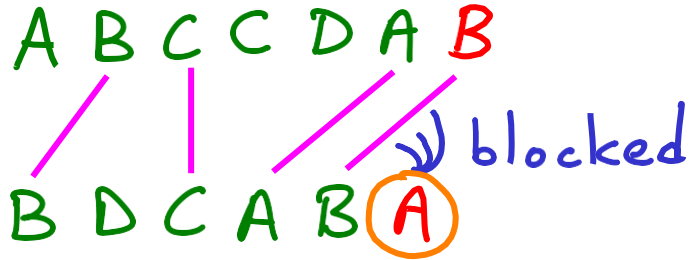
B D C A B A

Finding |LCS|

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

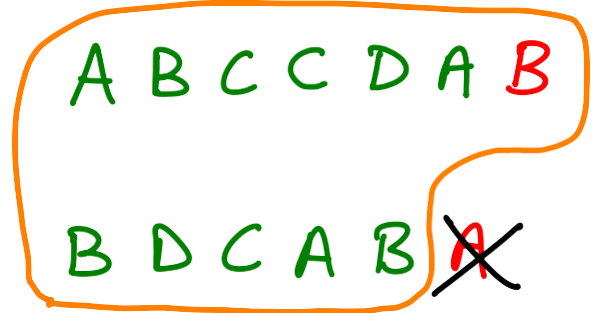
$LCS(x[1...i], y[1...j])$

cannot use both $x[i]$ and $y[j]$

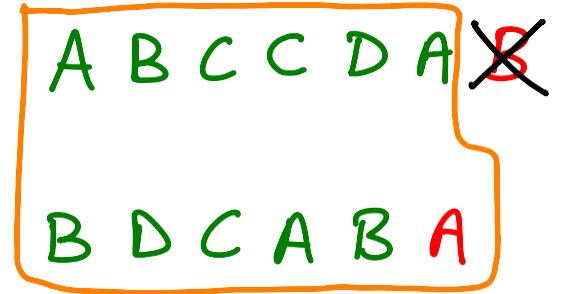


Hide each
and take
best result

$c(i, j-1)$



$c(i-1, j)$



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

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

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

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

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

LCS(x, y, i, j)

return c_{ij}

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

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

LCS(x, y, i, j)

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

return c_{ij}

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

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

$LCS(x, y, i, j)$ \ \ 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 $c_{ij} \leftarrow \text{LCS}(X, Y, i-1, j-1) + 1$

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

return c_{ij}

worst case : ?

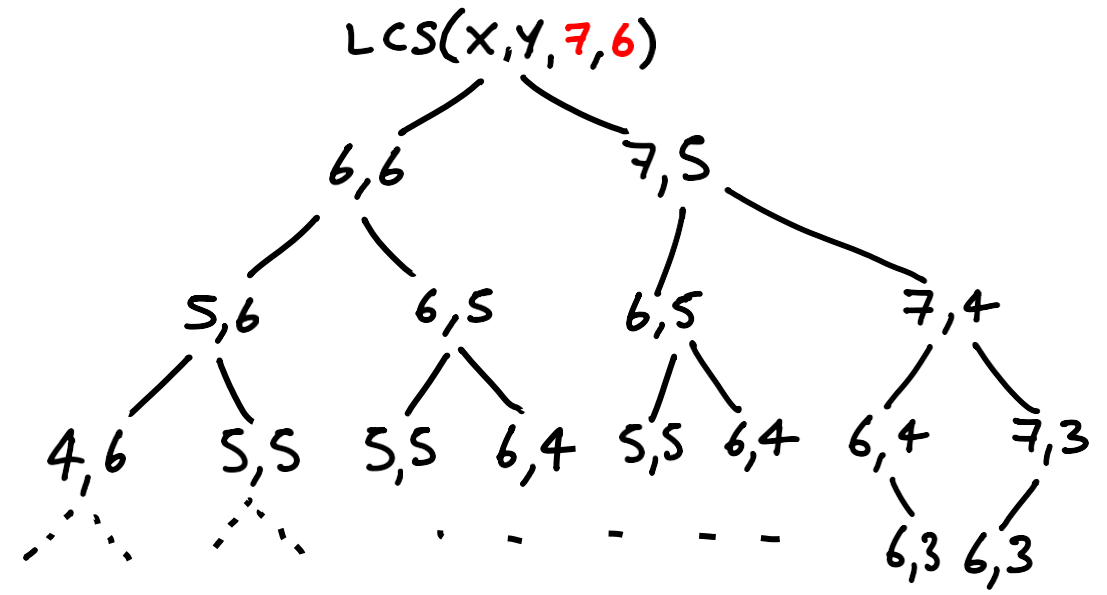
LCS(X,Y,i,j)

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: $n=7, m=6$



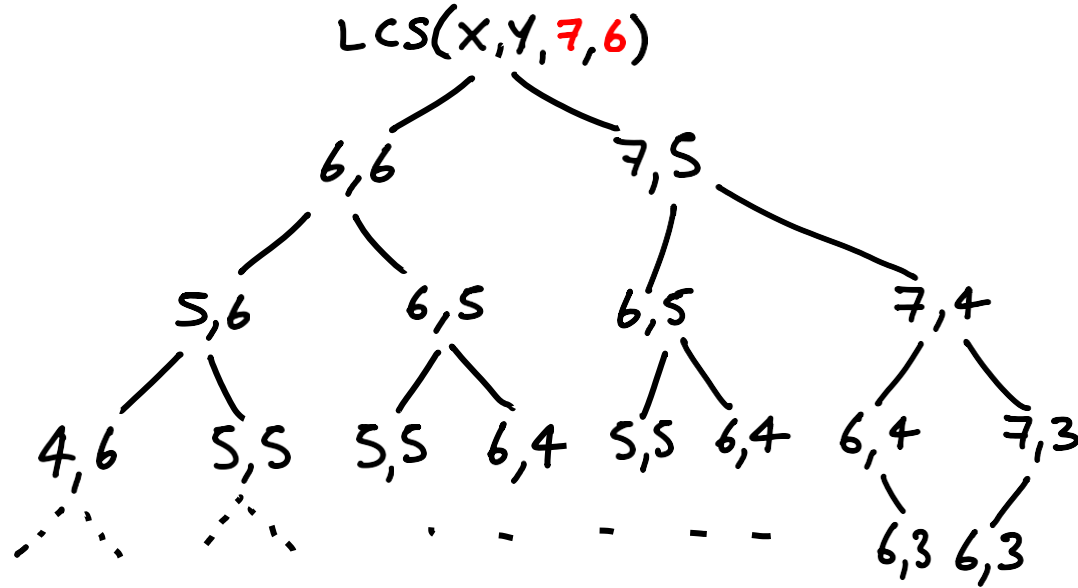
$LCS(X, Y, i, j)$

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 $m \neq n$

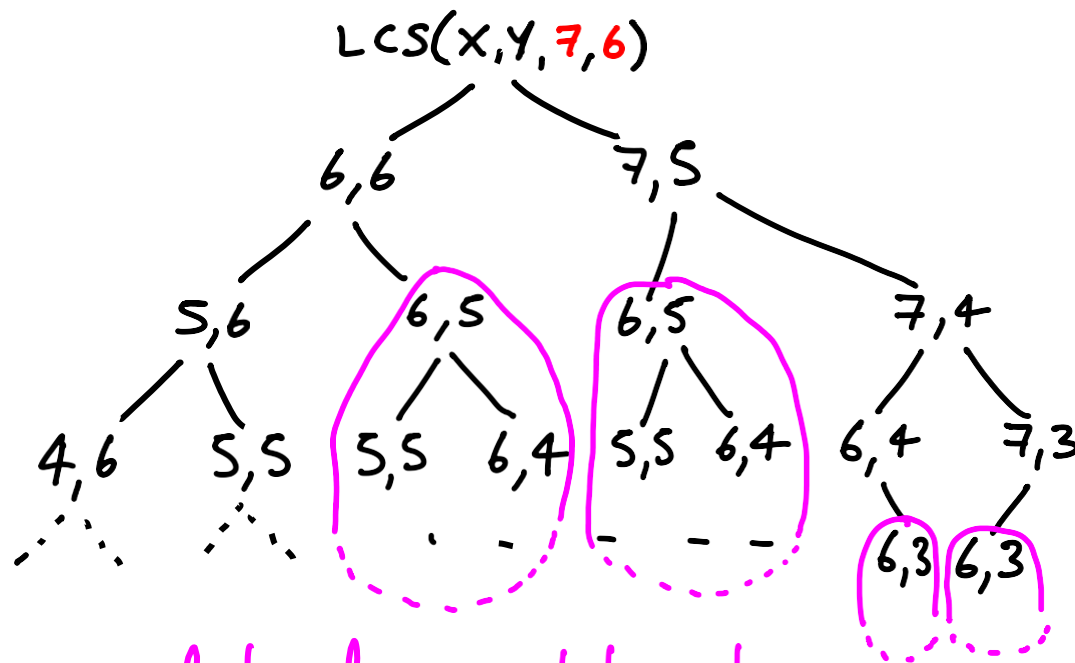
$LCS(X, Y, i, j)$

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 $m < n$

lots of repeated work

\hookrightarrow #distinct subproblems = ?

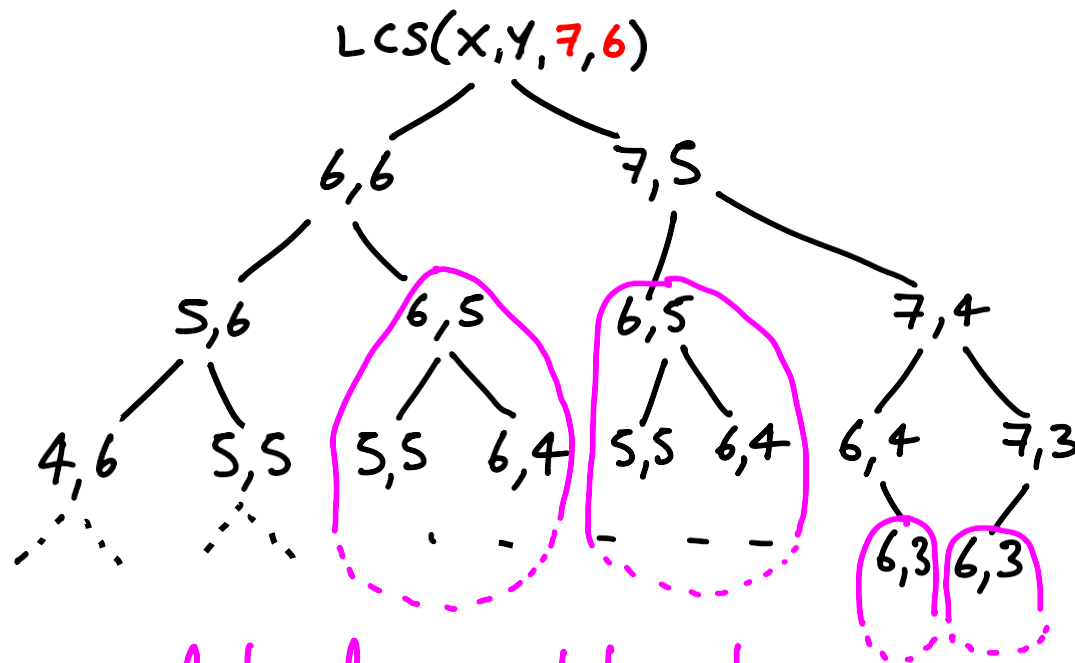
$LCS(X, Y, i, j)$

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 $m \neq n$

lots of repeated work

\hookrightarrow #distinct subproblems = $m \cdot n$

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

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}

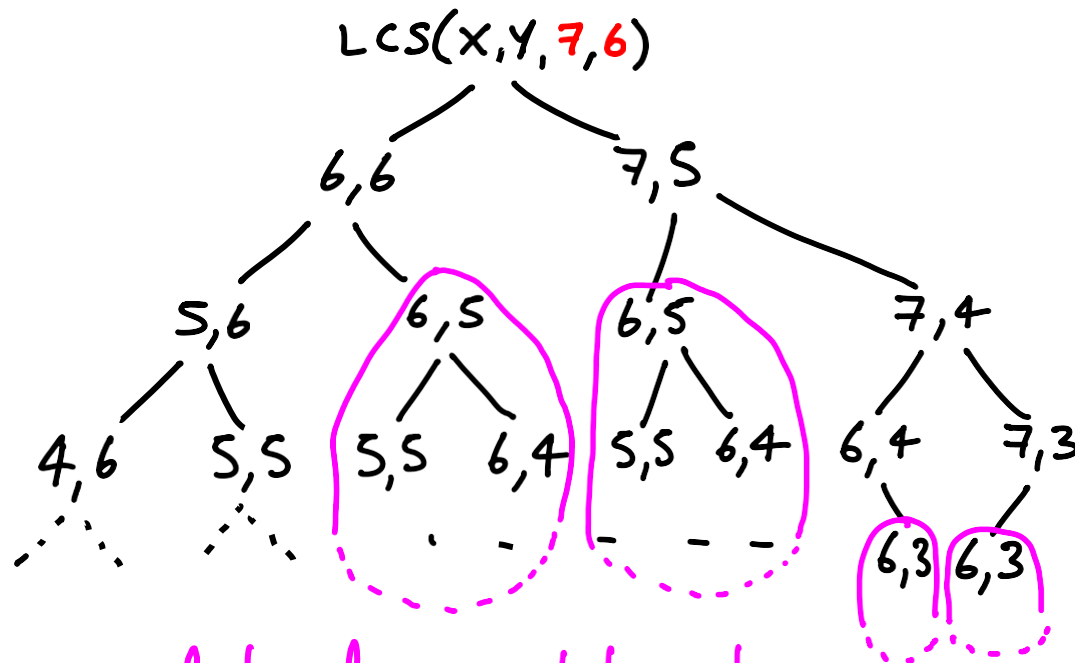
worst case : always get $X_i \neq Y_j$

ex: $n=7, m=6$

Repeated subproblems
+
optimal substructure



try dynamic programming



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

work = $\Omega(2^n)$

if $m \neq n$

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}

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)

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

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 $c[1\dots m, 1\dots n]$ 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 $c_{ij} \leftarrow \text{LCS}(X, Y, i-1, j-1) + 1$

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 $c[1\dots m, 1\dots n]$ 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 $c_{ij} \leftarrow \text{LCS}(X, Y, i-1, j-1) + 1$

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 $c[1\dots m, 1\dots n]$ 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 $c_{ij} \leftarrow \text{LCS}(X, Y, i-1, j-1) + 1$

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 $c[1\dots m, 1\dots n]$ 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 $c_{ij} \leftarrow \text{LCS}(X, Y, i-1, j-1) + 1$

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 $c[1\dots m, 1\dots n]$ 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 $c[i, j] \leftarrow \text{LCS}(X, Y, i-1, j-1) + 1$

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

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 $c[1\dots m, 1\dots n]$ 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 $c[i, j] \leftarrow \text{LCS}(X, Y, i-1, j-1) + 1$

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

return $c[i, j]$ // 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)

time?

Let $C[1 \dots m, 1 \dots n]$ 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 $C[i, j] \leftarrow \text{LCS}(X, Y, i-1, j-1) + 1$

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

return $C[i, j]$ // 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)

Let $C[1\dots m, 1\dots n]$ 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 $C[i, j] \leftarrow \text{LCS}(X, Y, i-1, j-1) + 1$

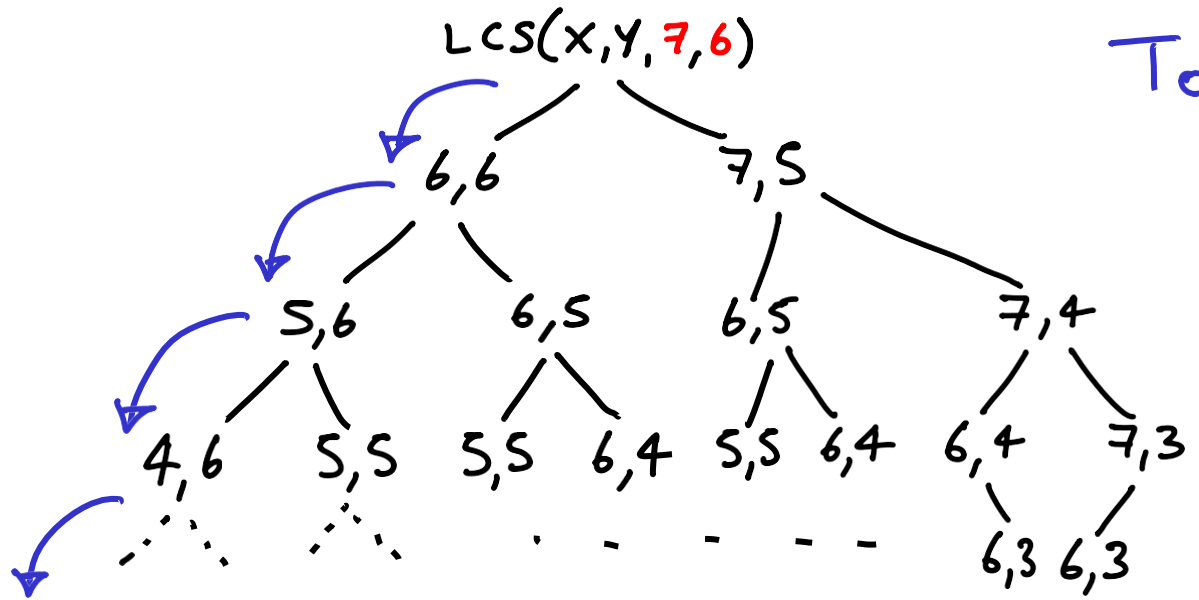
else $C[i, j] \leftarrow \max\{\text{LCS}(X, Y, i, j-1), \text{LCS}(X, Y, i-1, j)\}$

return $C[i, j]$ // look up

Memoization

Make "memos" of solutions
(to subproblems)

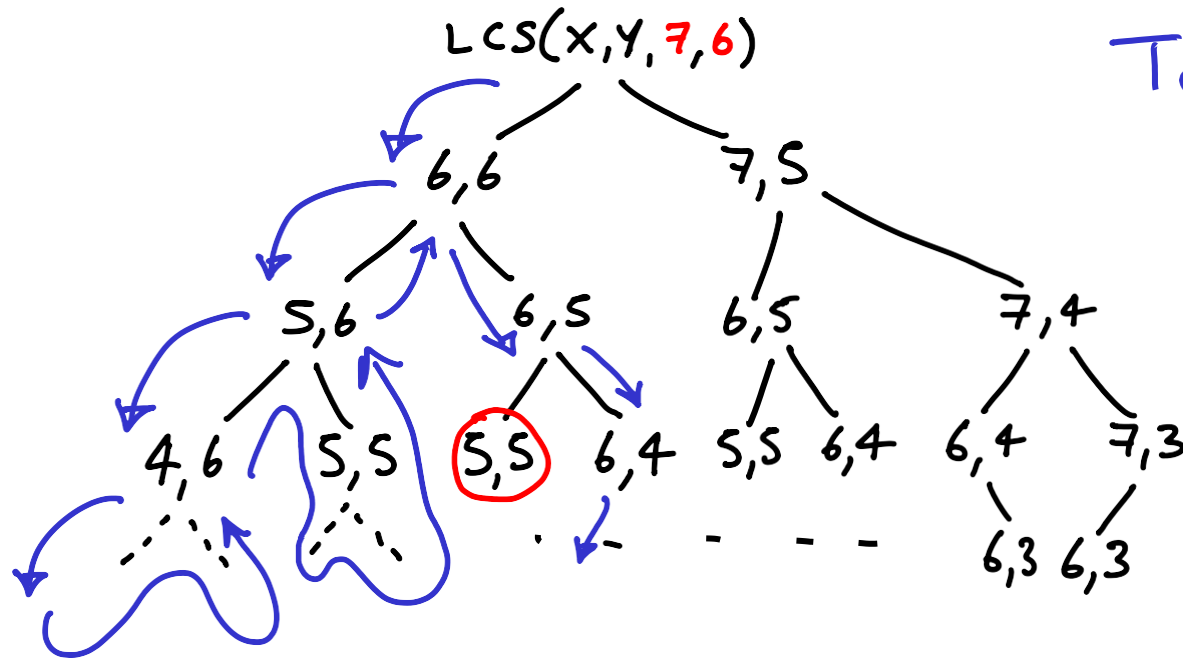
Top-down



Memoization

Make "memos" of solutions
(to subproblems)

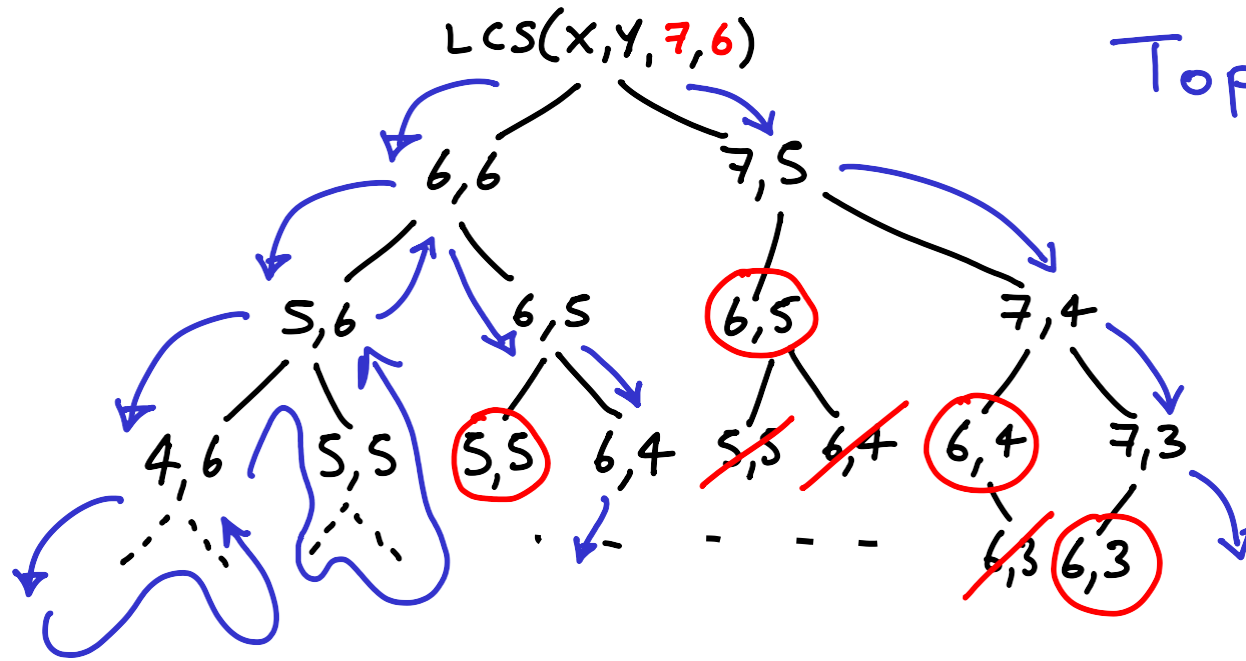
Top-down



Memoization

Make "memos" of solutions
(to subproblems)

Top-down



DYNAMIC PROGRAMMING

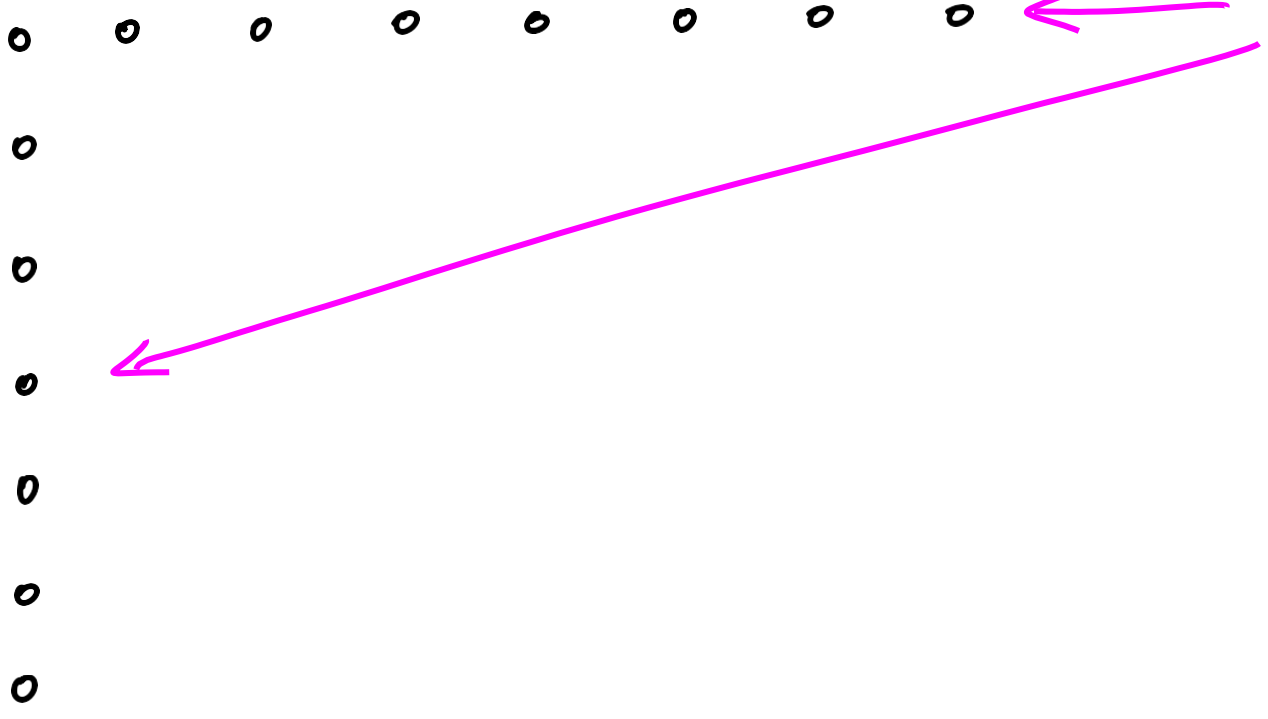
DYNAMIC PROGRAMMING

bottom-up

A B C B D A B

← base cases

B
D
C
A
B
A

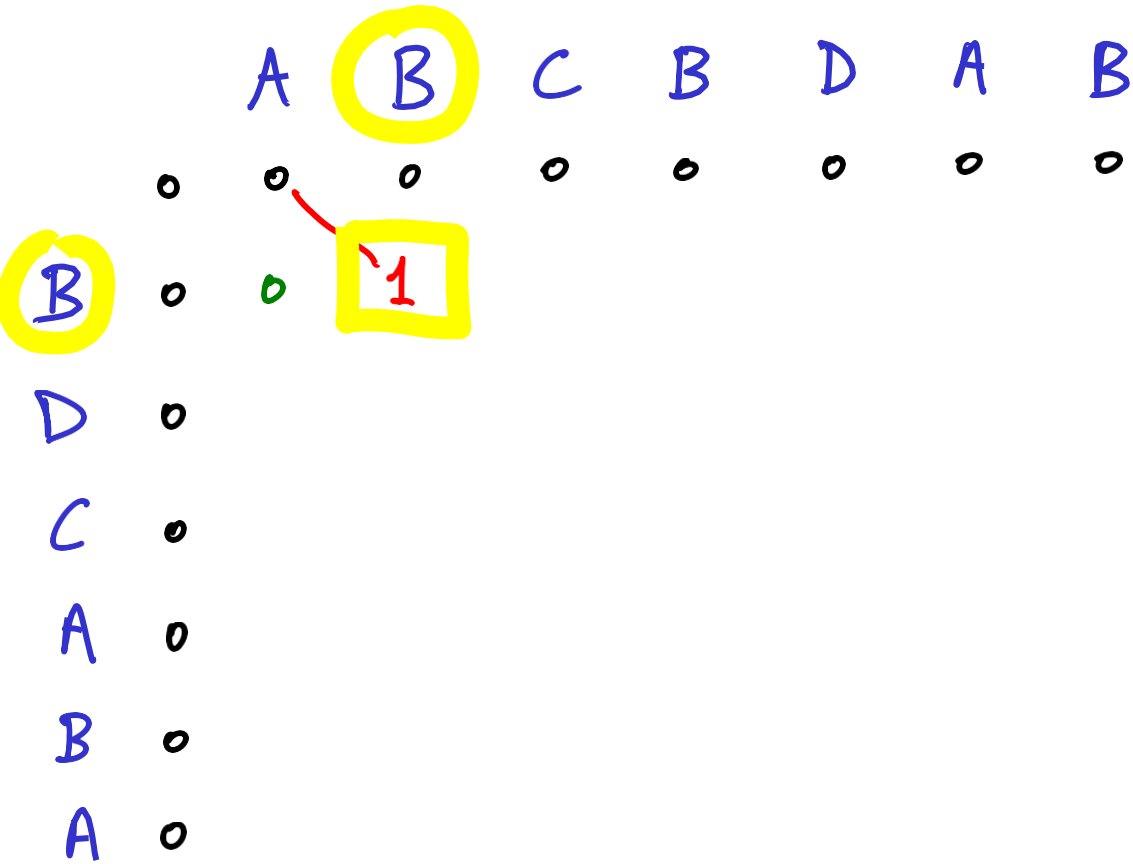


DYNAMIC PROGRAMMING

	A	B	C	B	D	A	B
	o	o	o	o	o	o	o
B	o	o					
D	o						
C	o						
A	o						
B	o						
A	o						

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

DYNAMIC PROGRAMMING



red # : $1 + \text{diag}^{\uparrow}\#$

when letters in column & row of #
match

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

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

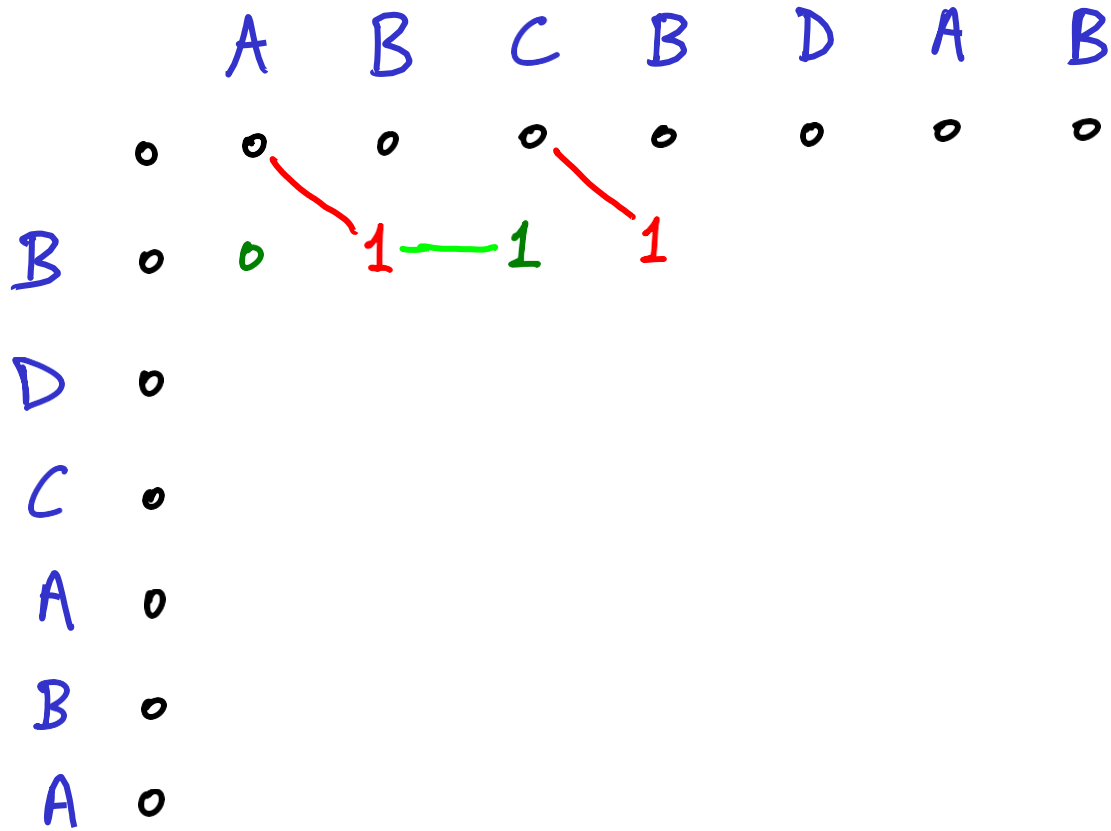
red # : $1 + \text{diag}^{\uparrow}\#$

when letters in column & row of #
match

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

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

DYNAMIC PROGRAMMING



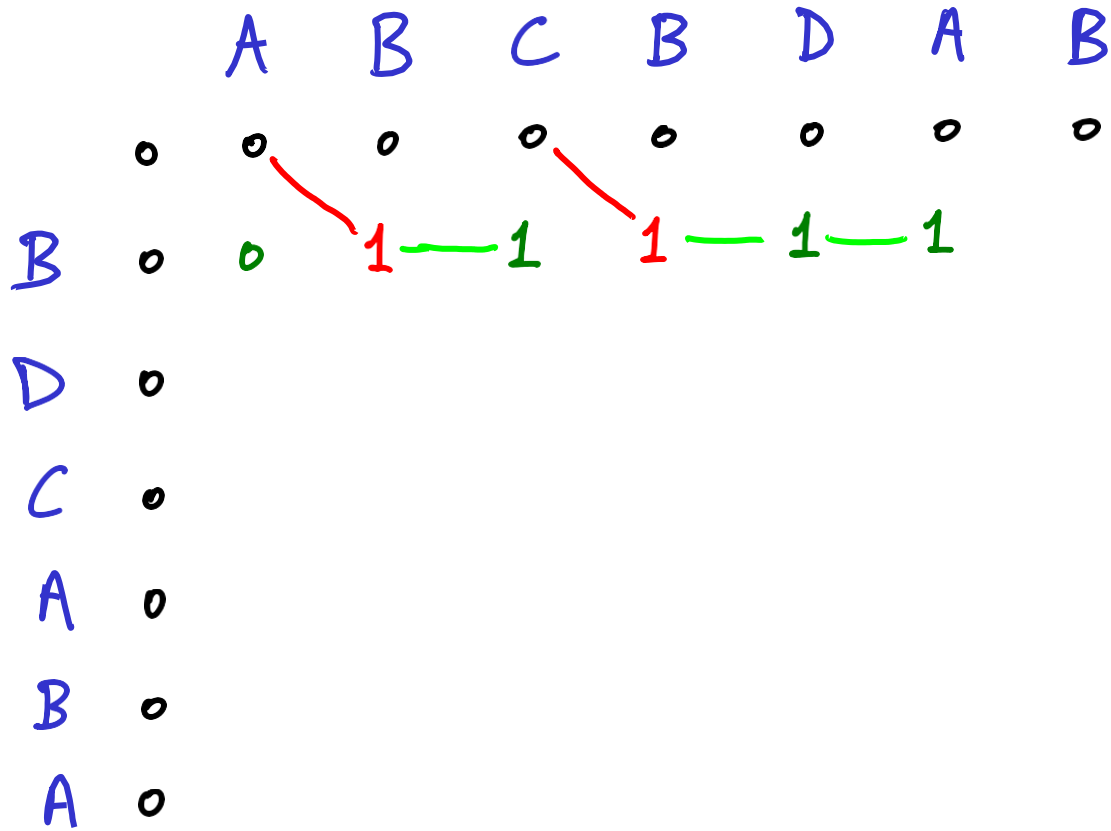
red # : $1 + \text{diag}^{\uparrow}\#$

when letters in column & row of # match

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

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

DYNAMIC PROGRAMMING



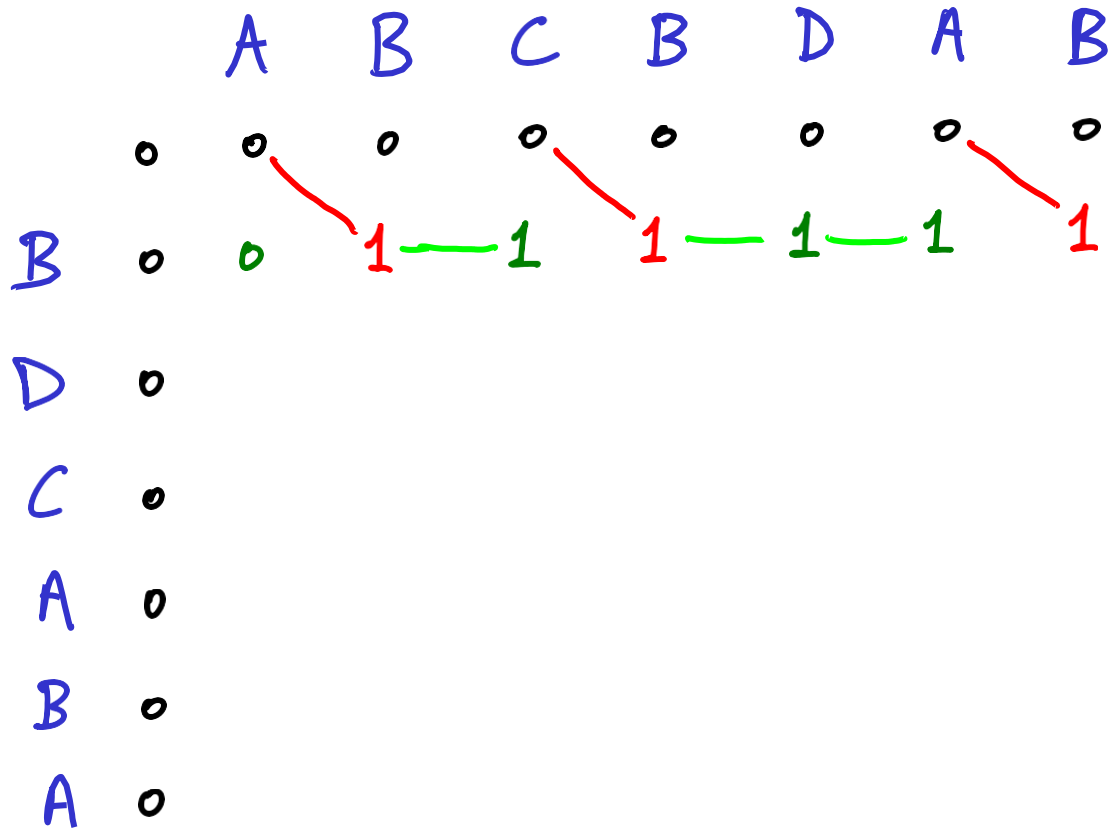
red # : $1 + \text{diag}^{\uparrow}\#$

when letters in column & row of # match

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

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

DYNAMIC PROGRAMMING



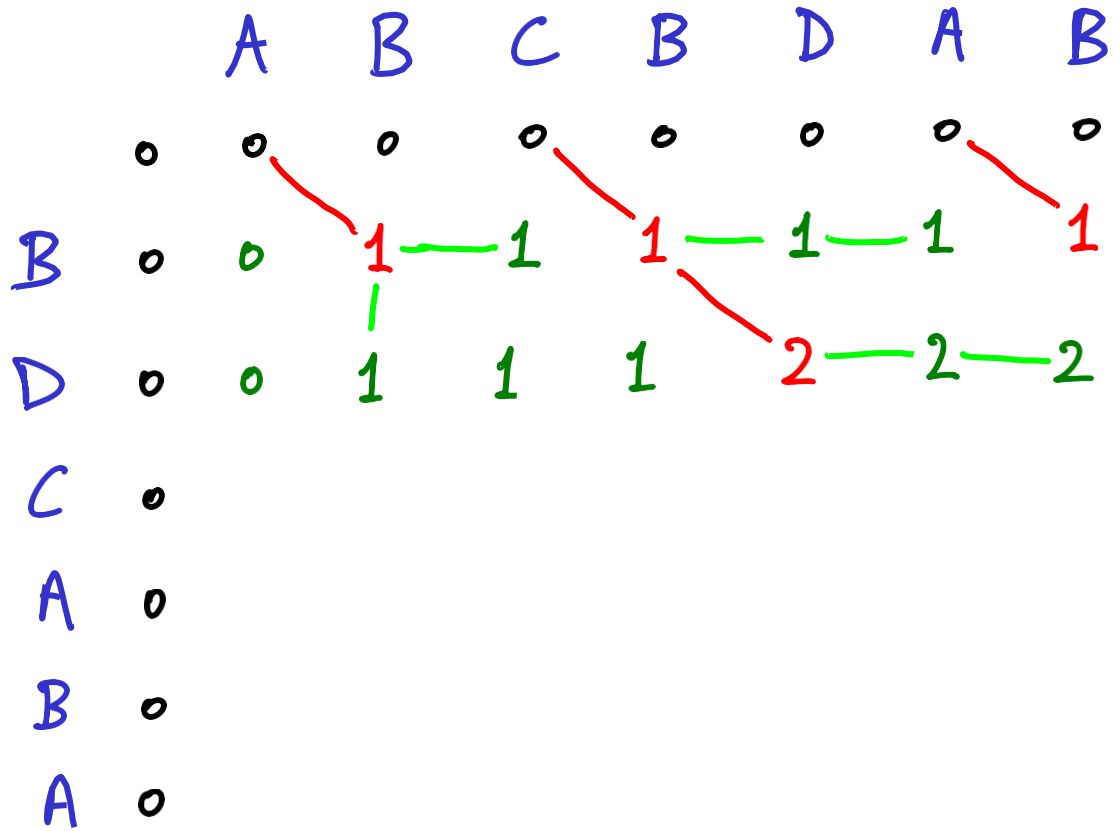
red # : $1 + \text{diag} \uparrow \#$

when letters in column & row of # match

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

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

DYNAMIC PROGRAMMING



red # : $1 + \text{diag}^{\uparrow}\#$

when letters in column & row of # match

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

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

DYNAMIC PROGRAMMING

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

red # : $1 + \text{diag}^{\uparrow}\#$

when letters in column & row of # match

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

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

DYNAMIC PROGRAMMING

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

The table shows the dynamic programming table for the sequence "ABCBA" (rows) and "ABCBA" (columns). Red lines indicate the path for the longest common subsequence (LCS) "ABAB". Green lines indicate the values for the longest common subsequence ending at each cell.

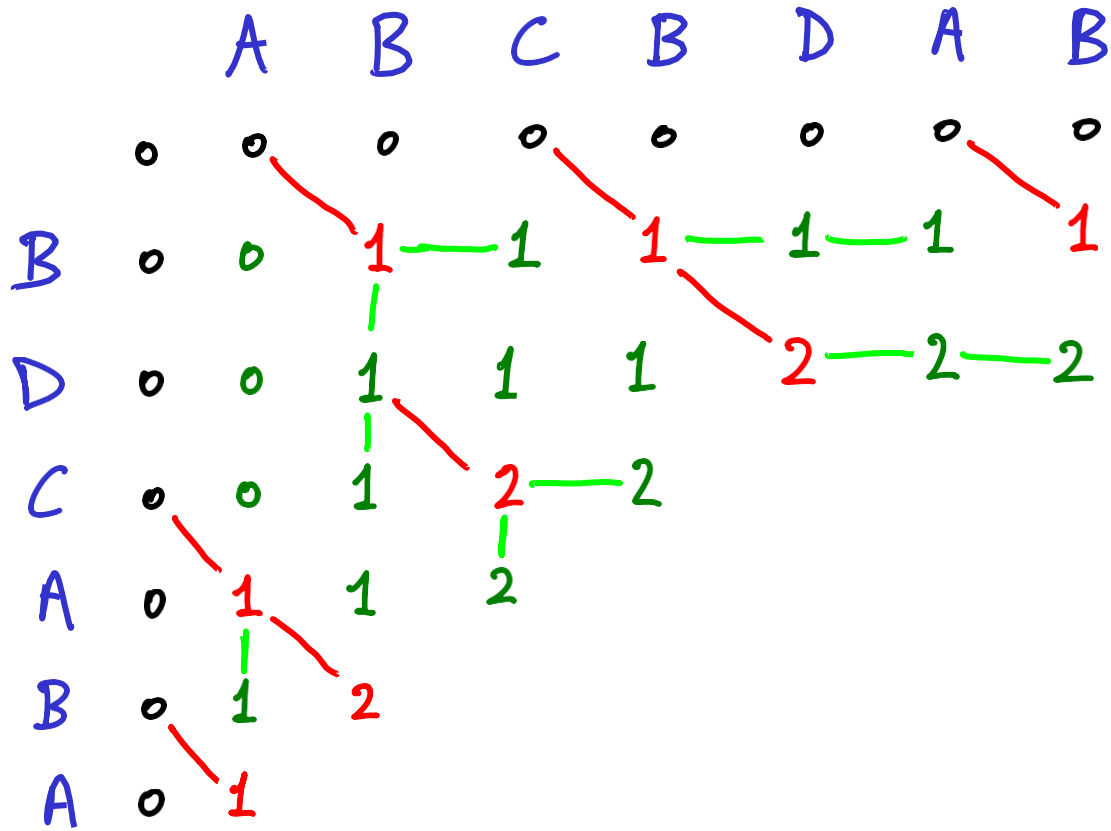
red # : $1 + \text{diag}^{\uparrow} \#$

when letters in column & row of # match

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

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

DYNAMIC PROGRAMMING



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

when letters in column & row of # match

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

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

DYNAMIC PROGRAMMING

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

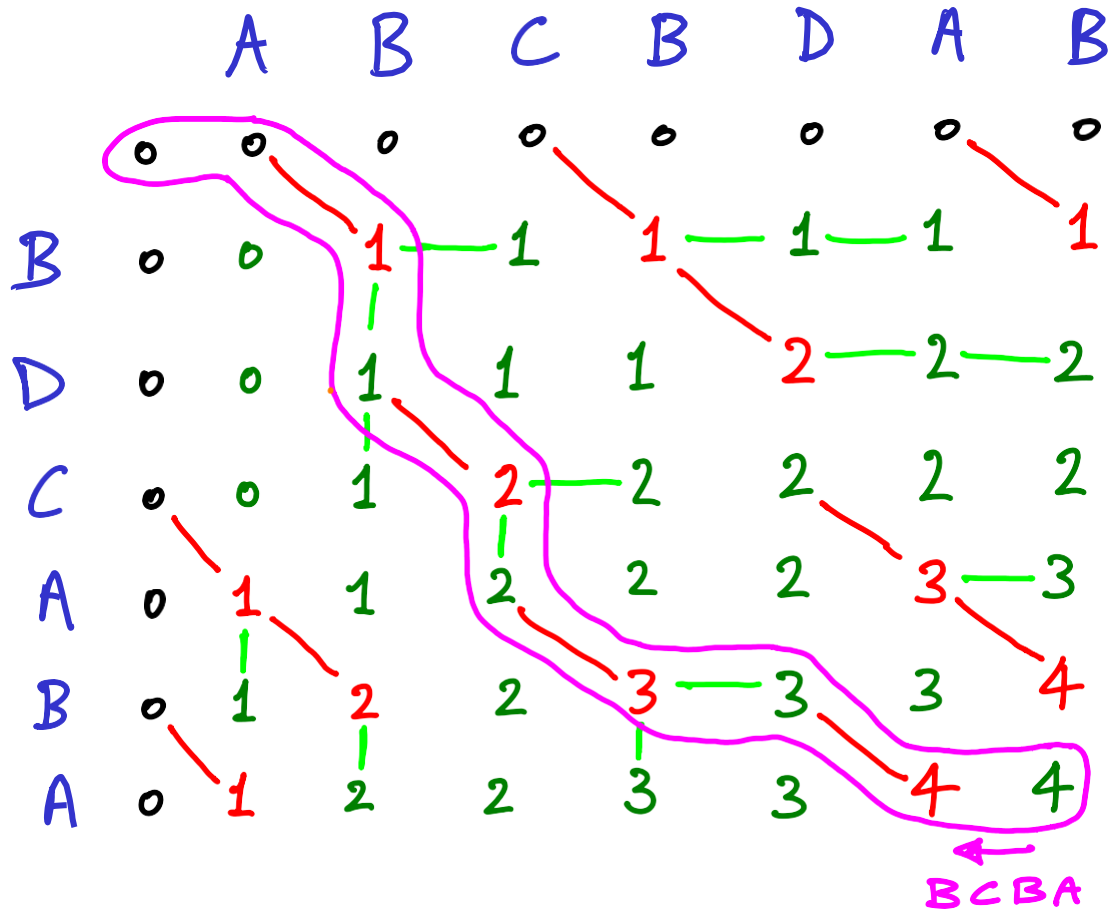
red # : $1 + \text{diag} \uparrow \#$

when letters in column & row of # match

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

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

DYNAMIC PROGRAMMING



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

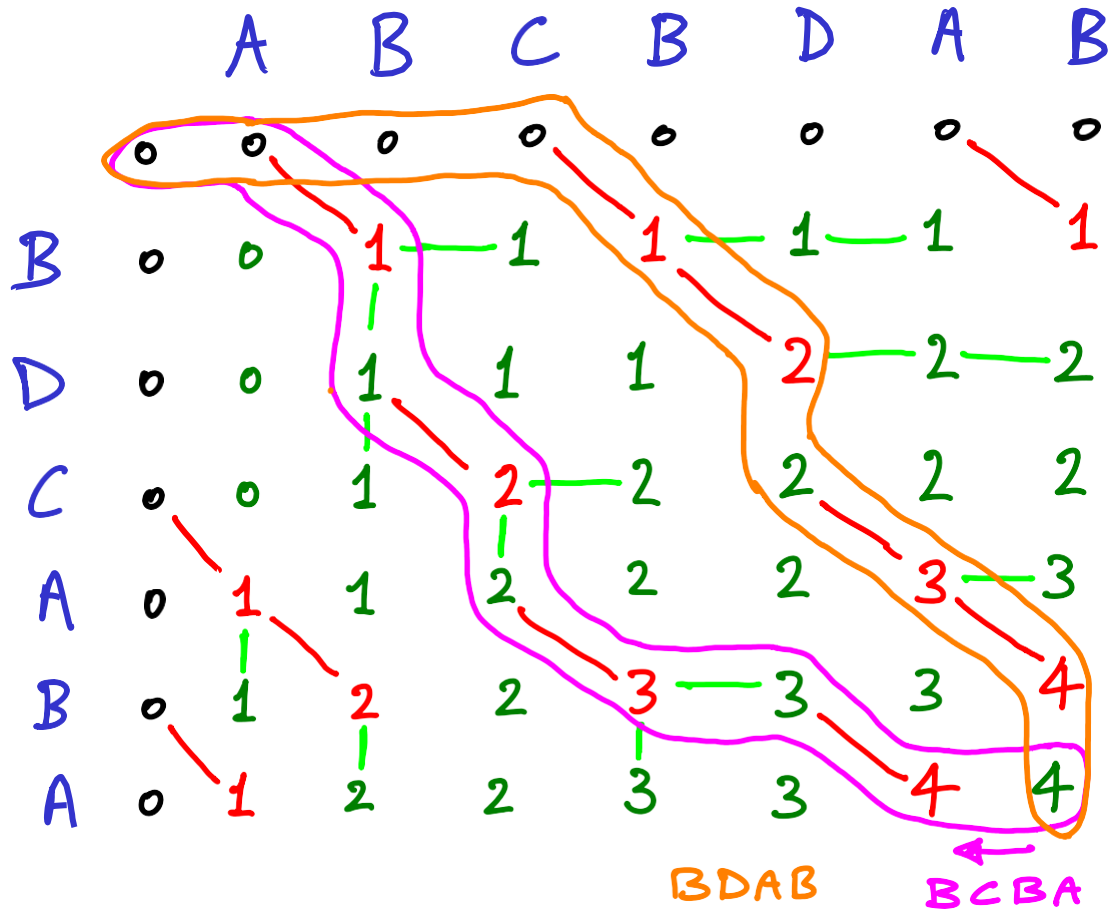
when letters in column & row of # match

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

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

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

DYNAMIC PROGRAMMING



red # : 1 + diag↑#

when letters in column & row of # match

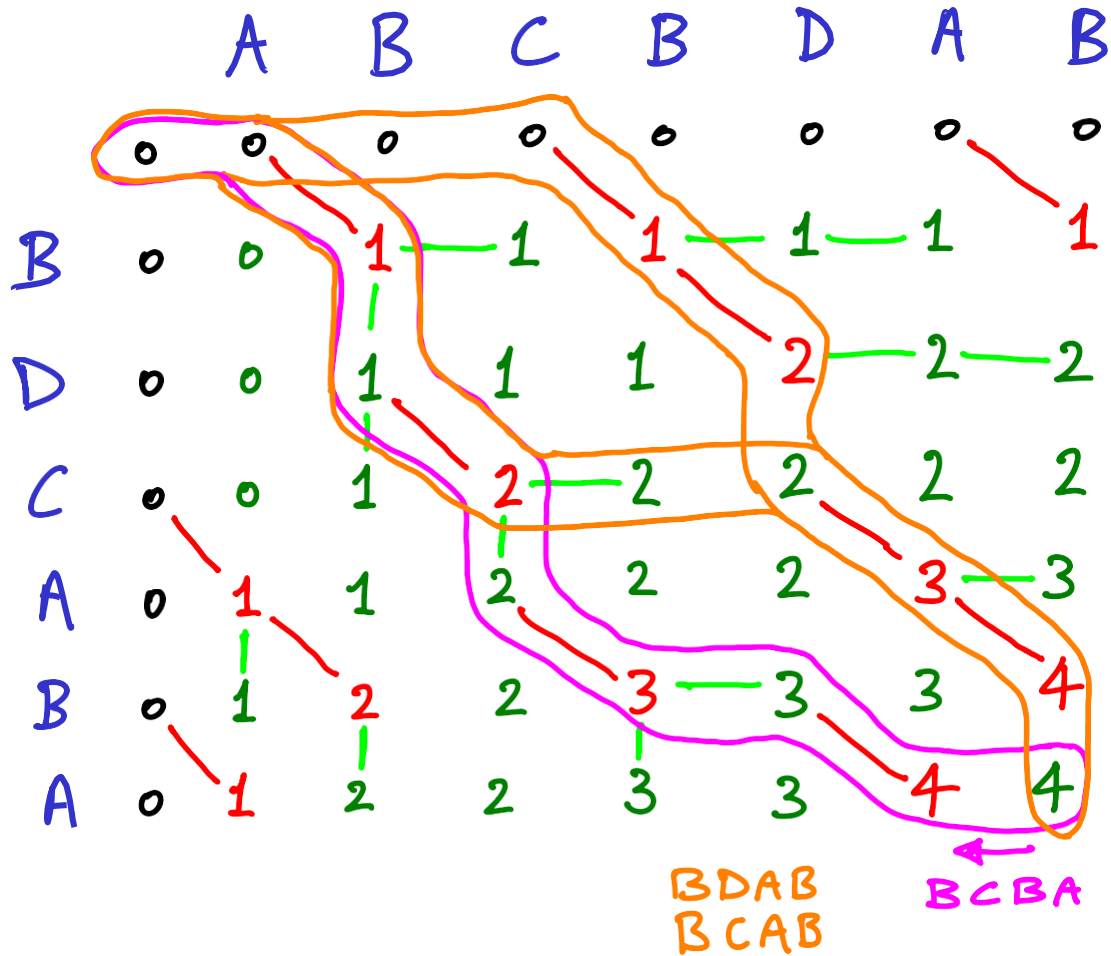
green # : max of {above, left}

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

DYNAMIC PROGRAMMING



red # : $1 + \text{diag}^{\uparrow} \#$

when letters in column & row of # match

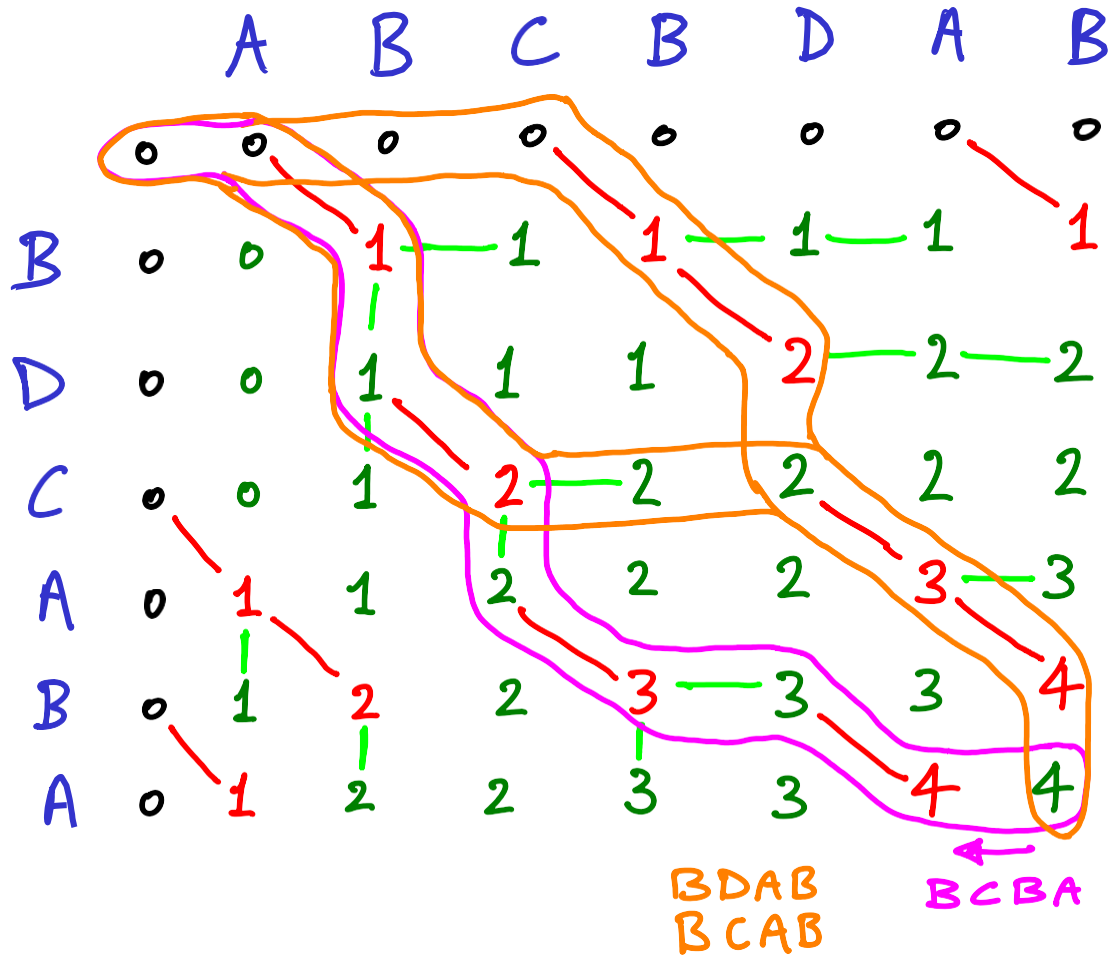
green # : $\max\{\text{above, left}\}$

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

DYNAMIC PROGRAMMING



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

when letters in column & row of # match

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

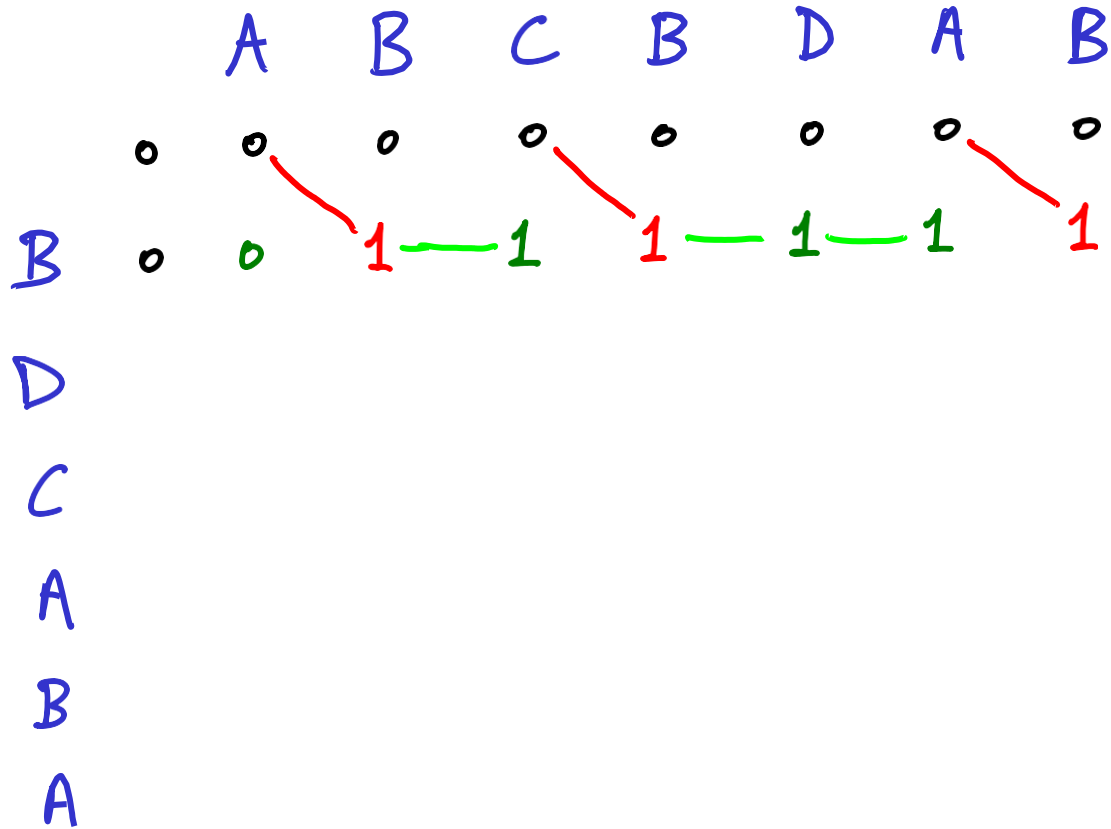
when letters in column & row of # don't match

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

when letters in column & row of # match

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

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								

Diagram description: The table shows the DP values for the LCS of "BDBACBA" and "ABCBA". Red numbers (1, 2) indicate the value at the current cell. Green lines show the path taken to reach the current cell: horizontal lines from the left and vertical lines from above. A red diagonal line from (row 2, col 5) to (row 1, col 4) indicates a match between 'D' and 'B'.

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

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								
B								
A								

etc

red # : $1 + \text{diag}^{\uparrow} \#$

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 : $\min\{m, n\}$

