

Dynamic Programming

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

20 December, 2021

Memoizing recursive implementations

```
def fib(n):  
    if n in fibtable.keys():  
        return(fibtable[n])  
  
    if n <= 1:  
        value = n  
    else:  
        value = fib(n-1) + fib(n-2)  
  
    fibtable[n] = value  
  
    return(value)
```

$\text{fib}(10)$

0	1	2	3	4	5	6	7	8	9	10
0	1	1	2	3	5	8	13	21	34	55

In general

```
def f(x,y,z):  
    if (x,y,z) in ftable.keys():  
        return(ftable[(x,y,z)])  
  
    recursively compute value  
    from subproblems  
  
    ftable[(x,y,z)] = value  
  
    return(value)
```

Dynamic programming

- Anticipate the structure of subproblems
 - Derive from inductive definition
 - Dependencies are acyclic

Dynamic programming

- Anticipate the structure of subproblems

- Derive from inductive definition
- Dependencies are acyclic

Evaluating `fib(5)`

`fib(5)`

`fib(4)`

`fib(3)`

`fib(2)`

`fib(1)`

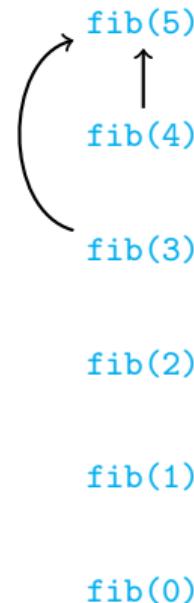
`fib(0)`

Dynamic programming

- Anticipate the structure of subproblems

- Derive from inductive definition
- Dependencies are acyclic

Evaluating $\text{fib}(5)$

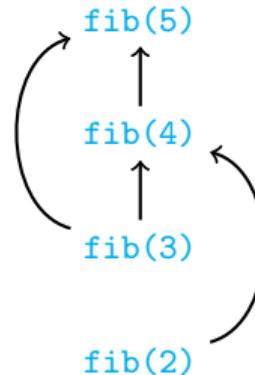


Dynamic programming

- Anticipate the structure of subproblems

- Derive from inductive definition
- Dependencies are acyclic

Evaluating $\text{fib}(5)$



$\text{fib}(1)$

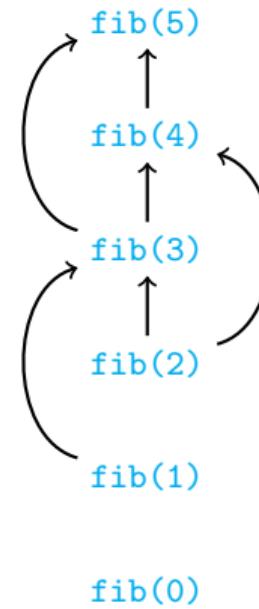
$\text{fib}(0)$

Dynamic programming

- Anticipate the structure of subproblems

- Derive from inductive definition
- Dependencies are acyclic

Evaluating $\text{fib}(5)$

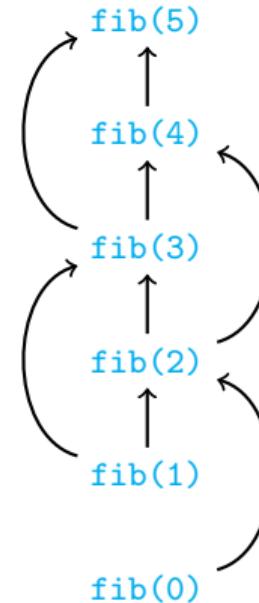


Dynamic programming

- Anticipate the structure of subproblems

- Derive from inductive definition
- Dependencies are acyclic

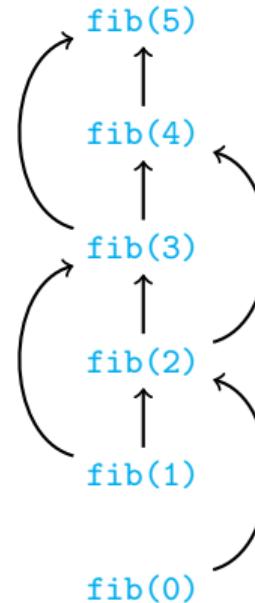
Evaluating $\text{fib}(5)$



Dynamic programming

- Anticipate the structure of subproblems
 - Derive from inductive definition
 - Dependencies are acyclic
- Solve subproblems in appropriate order

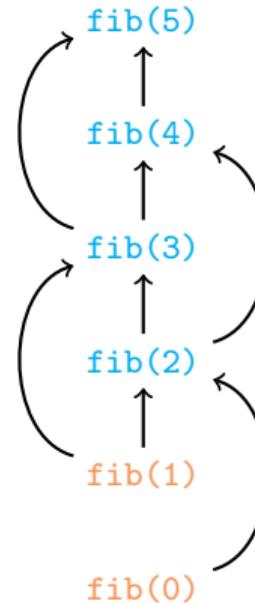
Evaluating $\text{fib}(5)$



Dynamic programming

- Anticipate the structure of subproblems
 - Derive from inductive definition
 - Dependencies are acyclic
- Solve subproblems in appropriate order
 - Start with base cases — no dependencies

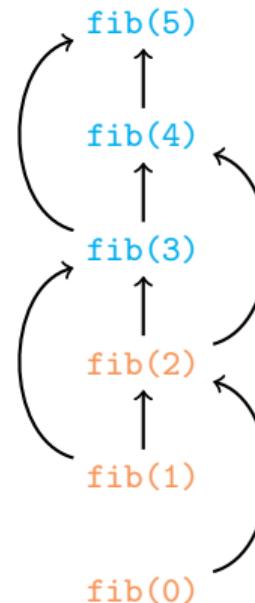
Evaluating $\text{fib}(5)$



Dynamic programming

- Anticipate the structure of subproblems
 - Derive from inductive definition
 - Dependencies are acyclic
- Solve subproblems in appropriate order
 - Start with base cases — no dependencies
 - Evaluate a value after all its dependencies are available

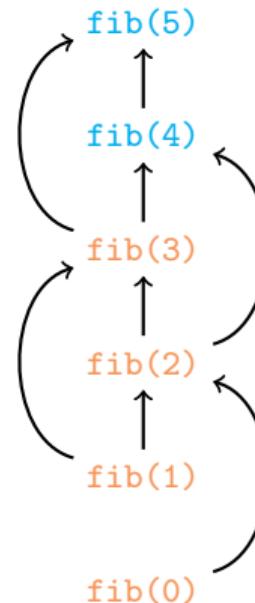
Evaluating $\text{fib}(5)$



Dynamic programming

- Anticipate the structure of subproblems
 - Derive from inductive definition
 - Dependencies are acyclic
- Solve subproblems in appropriate order
 - Start with base cases — no dependencies
 - Evaluate a value after all its dependencies are available

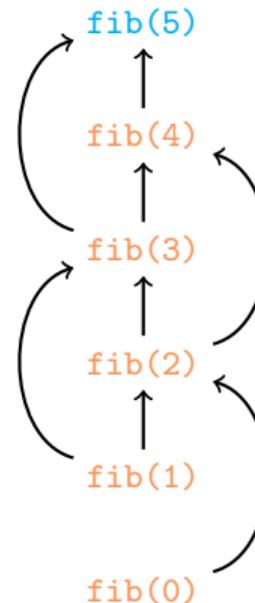
Evaluating $\text{fib}(5)$



Dynamic programming

- Anticipate the structure of subproblems
 - Derive from inductive definition
 - Dependencies are acyclic
- Solve subproblems in appropriate order
 - Start with base cases — no dependencies
 - Evaluate a value after all its dependencies are available

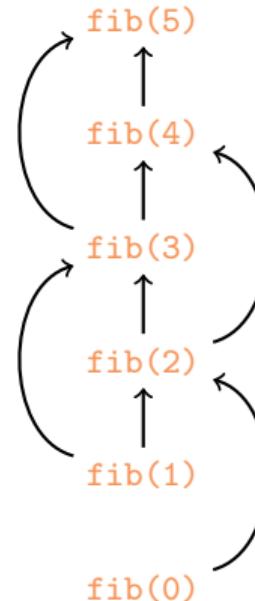
Evaluating $\text{fib}(5)$



Dynamic programming

- Anticipate the structure of subproblems
 - Derive from inductive definition
 - Dependencies are acyclic
- Solve subproblems in appropriate order
 - Start with base cases — no dependencies
 - Evaluate a value after all its dependencies are available

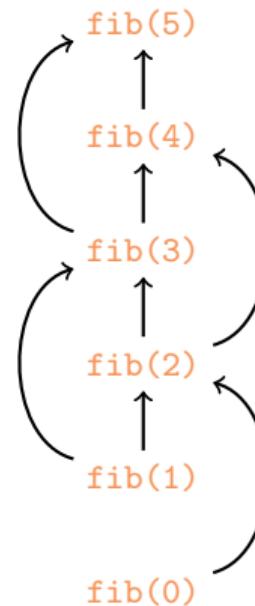
Evaluating $\text{fib}(5)$



Dynamic programming

- Anticipate the structure of subproblems
 - Derive from inductive definition
 - Dependencies are acyclic
- Solve subproblems in appropriate order
 - Start with base cases — no dependencies
 - Evaluate a value after all its dependencies are available
 - Fill table iteratively
 - Never need to make a recursive call

Evaluating $\text{fib}(5)$

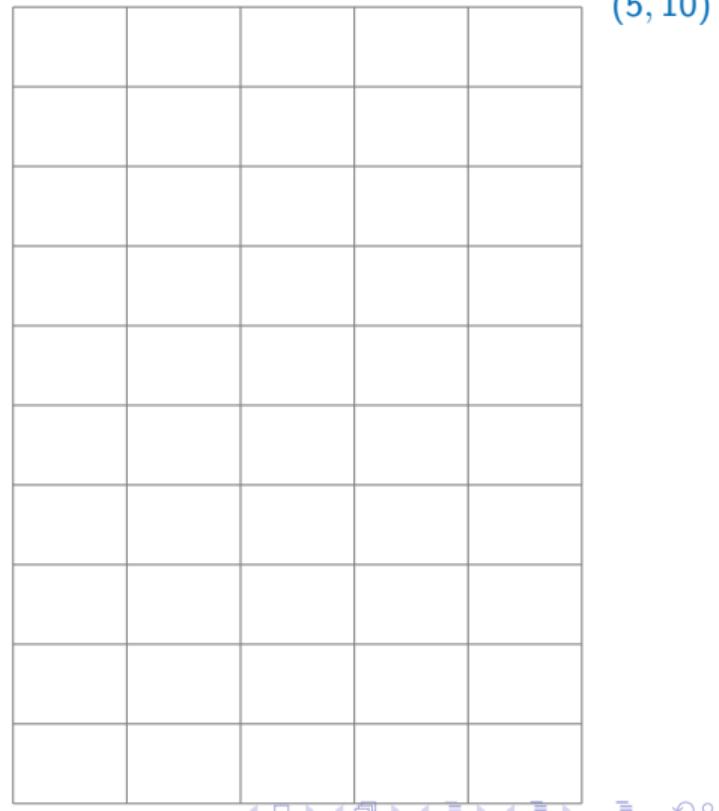


k	$f(k)$
5	
4	
3	
2	
1	
0	

A green arrow points vertically upwards through the table, starting from the bottom cell (0) and ending at the top cell (5), representing the iterative filling of the dynamic programming table.

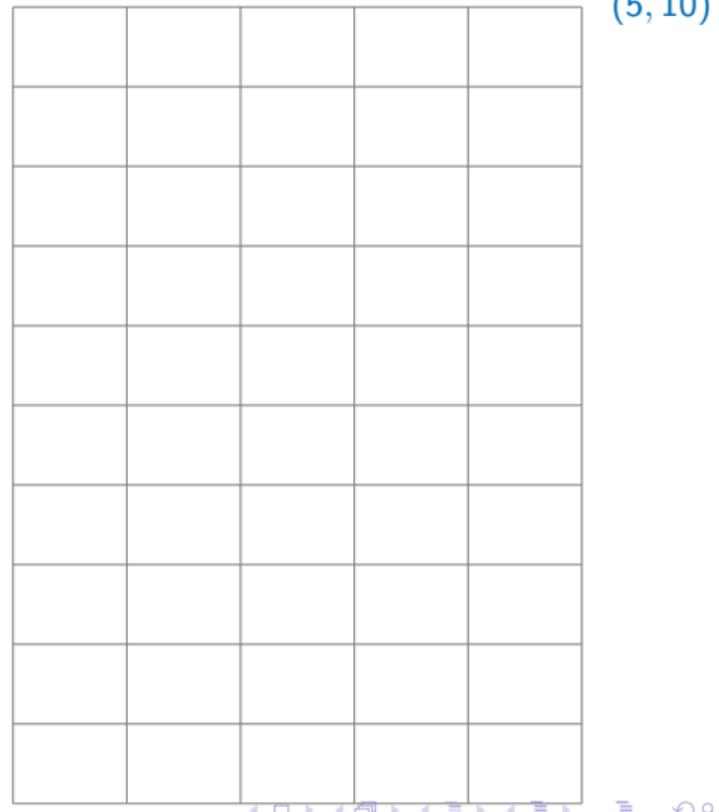
Grid paths

- Rectangular grid of one-way roads



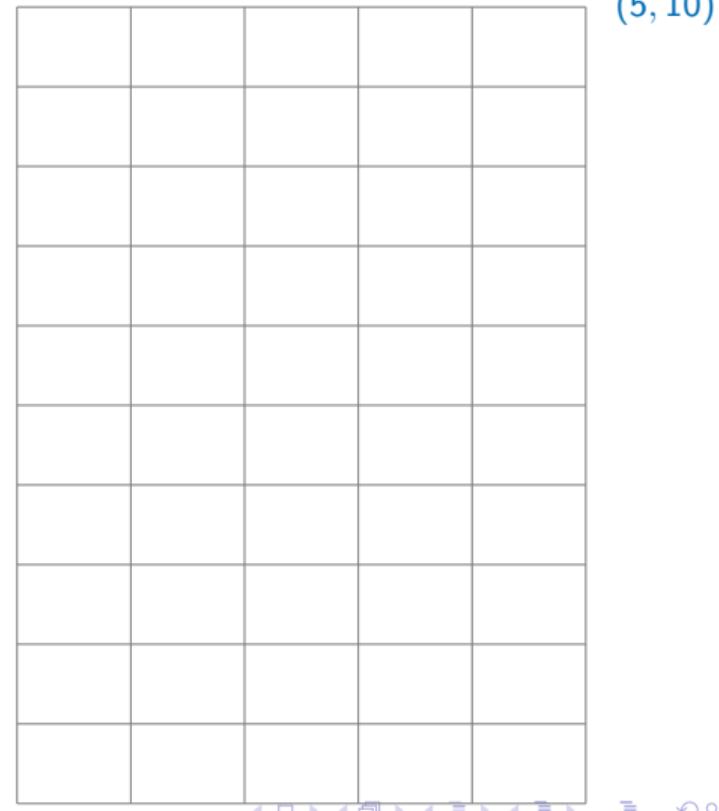
Grid paths

- Rectangular grid of one-way roads
- Can only go up and right



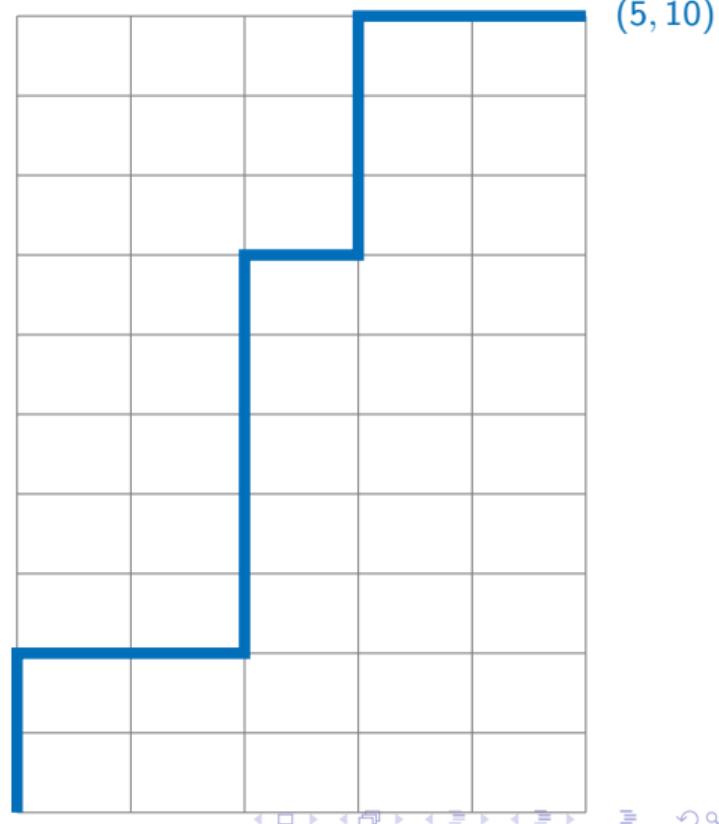
Grid paths

- Rectangular grid of one-way roads
- Can only go up and right
- How many paths from $(0, 0)$ to (m, n) ?



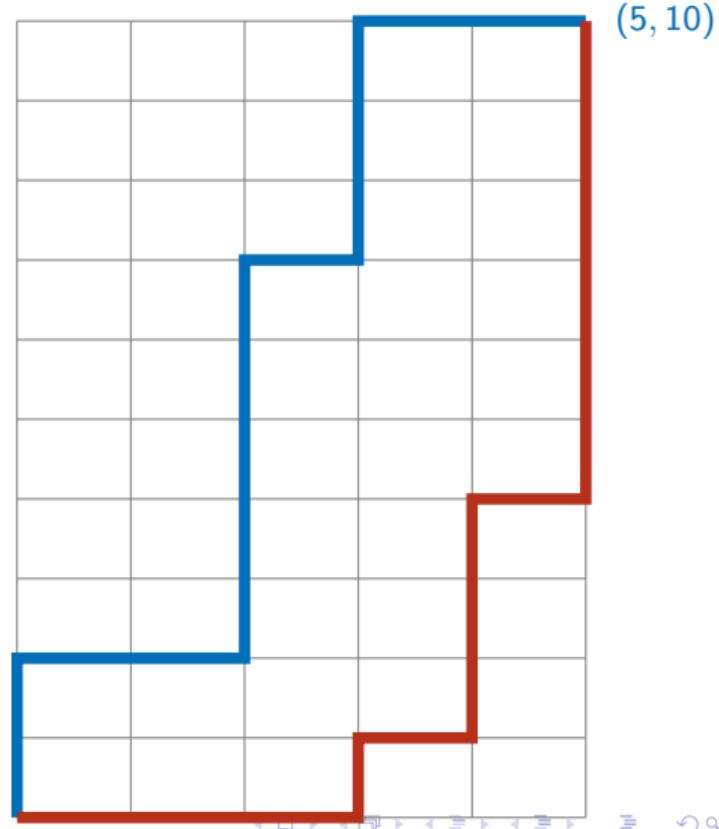
Grid paths

- Rectangular grid of one-way roads
- Can only go up and right
- How many paths from $(0, 0)$ to (m, n) ?



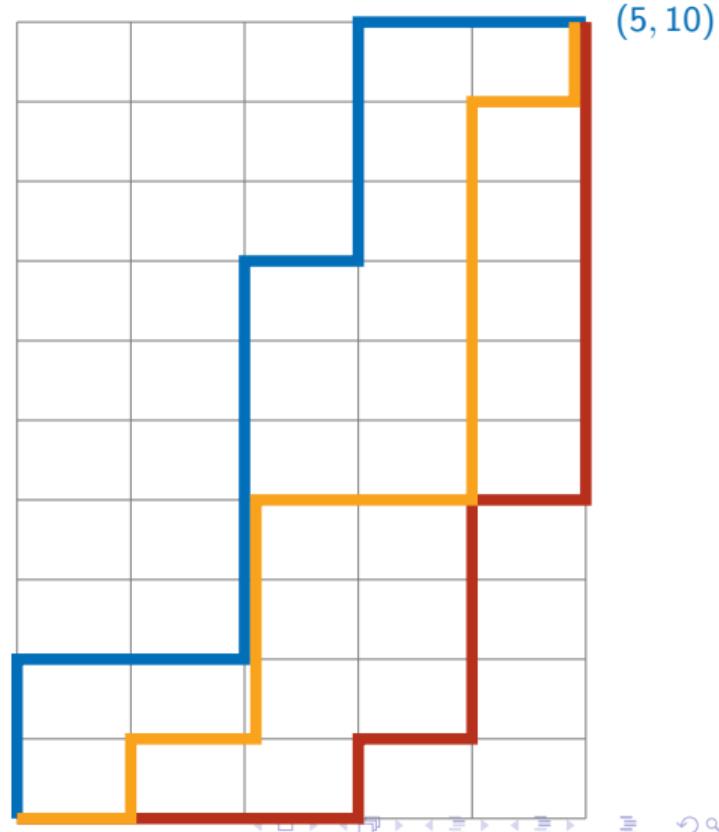
Grid paths

- Rectangular grid of one-way roads
- Can only go up and right
- How many paths from $(0, 0)$ to (m, n) ?



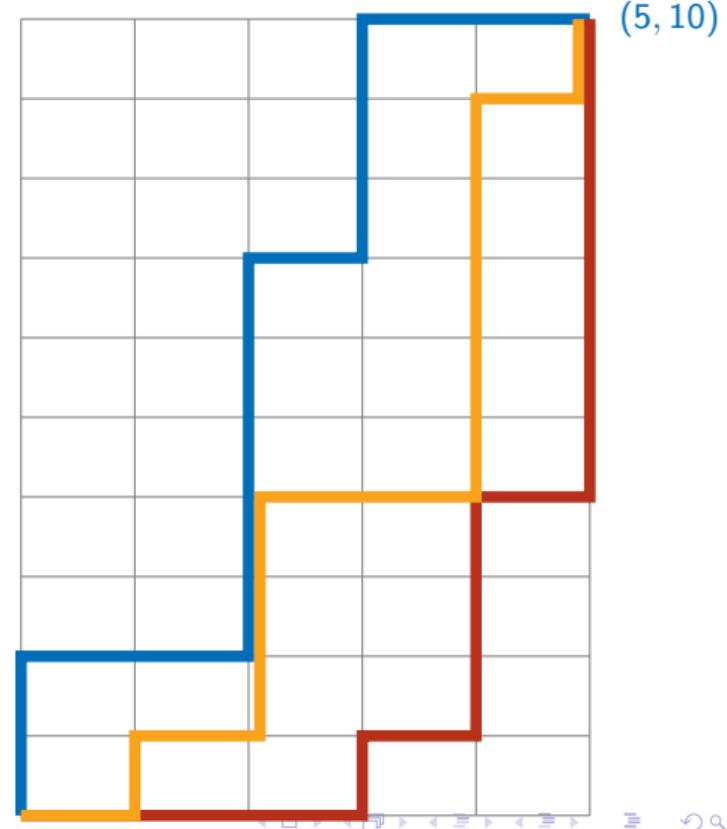
Grid paths

- Rectangular grid of one-way roads
- Can only go up and right
- How many paths from $(0, 0)$ to (m, n) ?



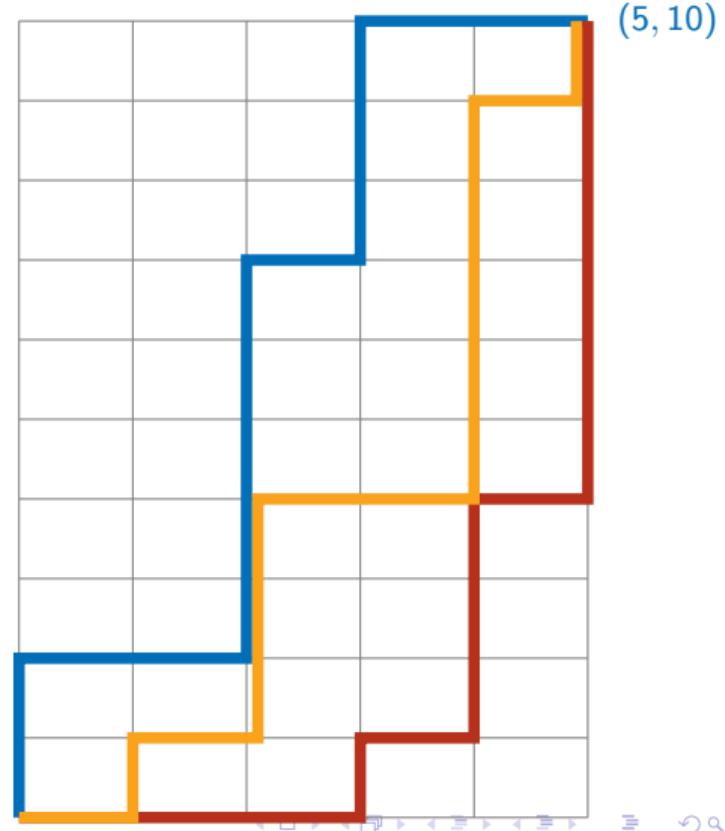
Combinatorial solution

- Every path from $(0, 0)$ to $(5, 10)$ has 15 segments
 - In general $m+n$ segments from $(0, 0)$ to (m, n)



Combinatorial solution

- Every path from $(0, 0)$ to $(5, 10)$ has 15 segments
 - In general $m+n$ segments from $(0, 0)$ to (m, n)
- Out of 15, exactly 5 are right moves, 10 are up moves



Combinatorial solution



- Every path from $(0, 0)$ to $(5, 10)$ has 15 segments

 - In general $m+n$ segments from $(0, 0)$ to (m, n)

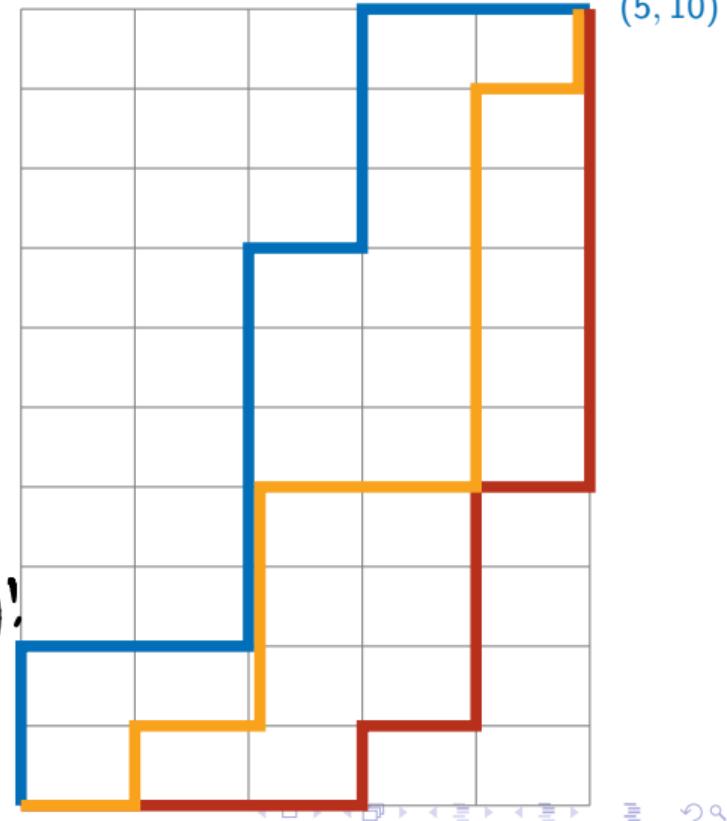
- Out of 15, exactly 5 are right moves, 10 are up moves

- Fix the positions of the 5 right moves among the 15 positions overall

- $\binom{15}{5} = \frac{15!}{10! \cdot 5!} = 3003$

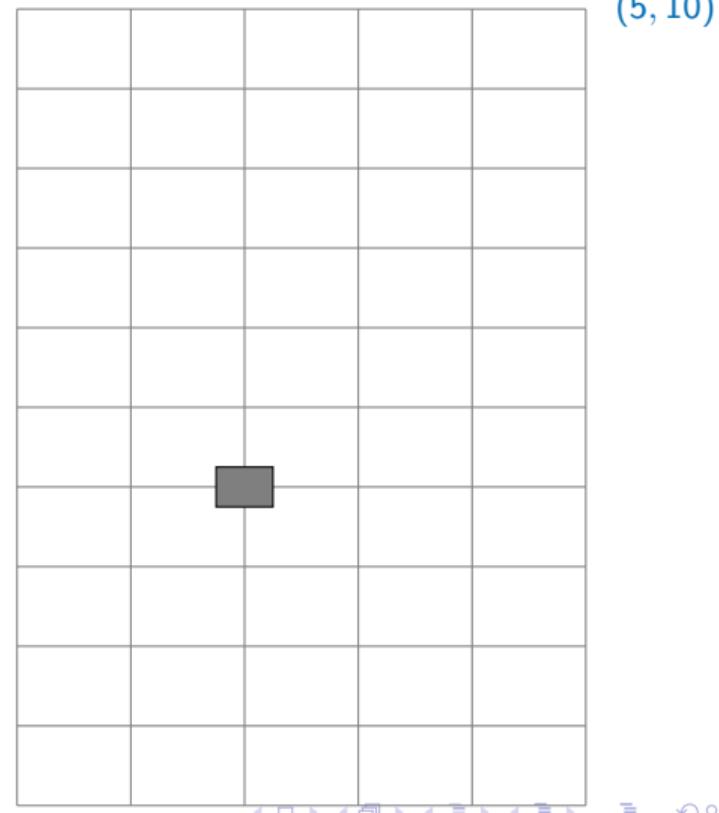
 - Same as $\binom{15}{10}$ — fix the 10 up moves

$$\frac{n!}{r! (n-r)!}$$



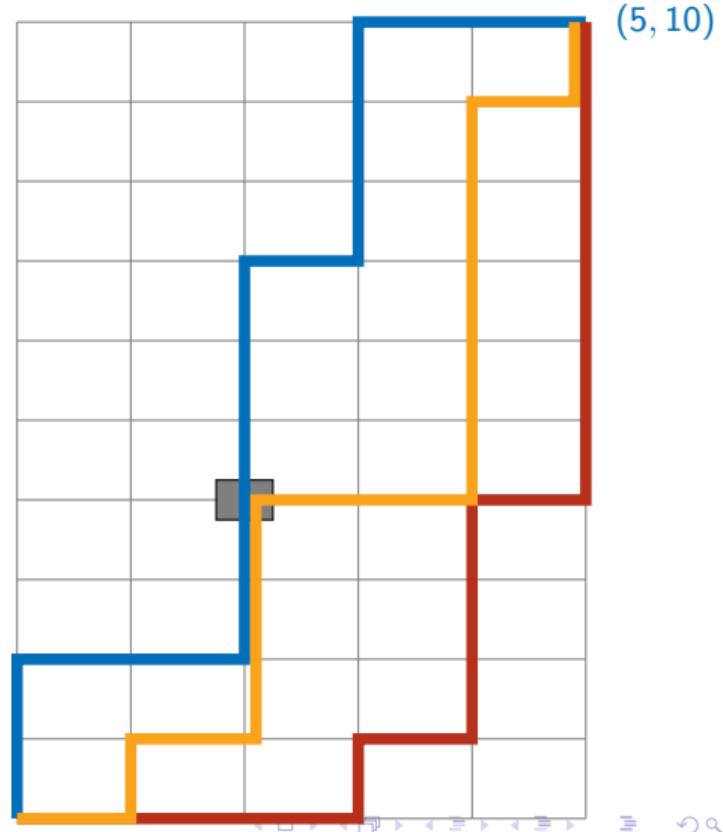
Holes

- What if an intersection is blocked?
 - For instance, $(2, 4)$



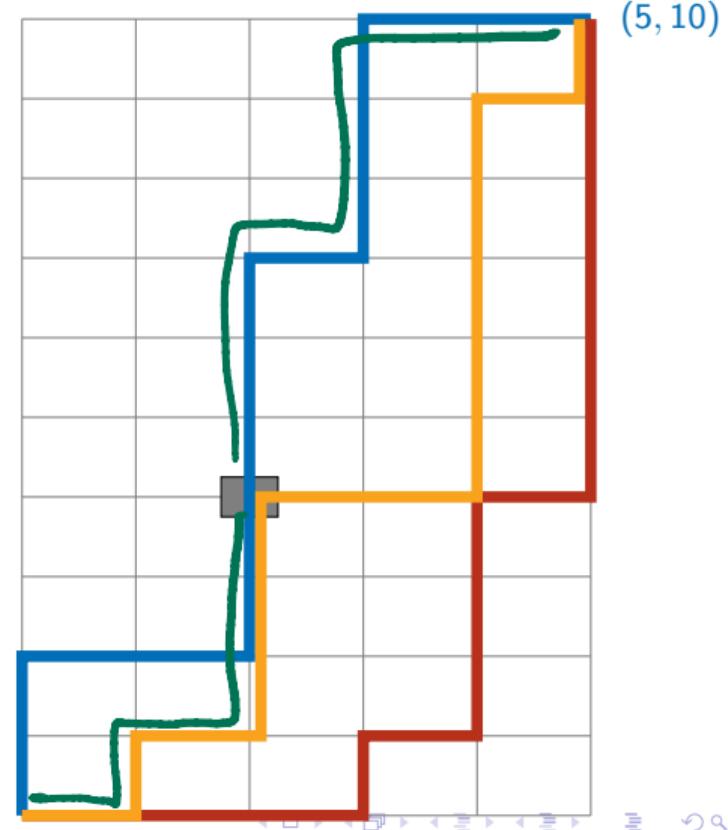
Holes

- What if an intersection is blocked?
 - For instance, $(2, 4)$
- Need to discard paths passing through $(2, 4)$
 - Two of our earlier examples are invalid paths



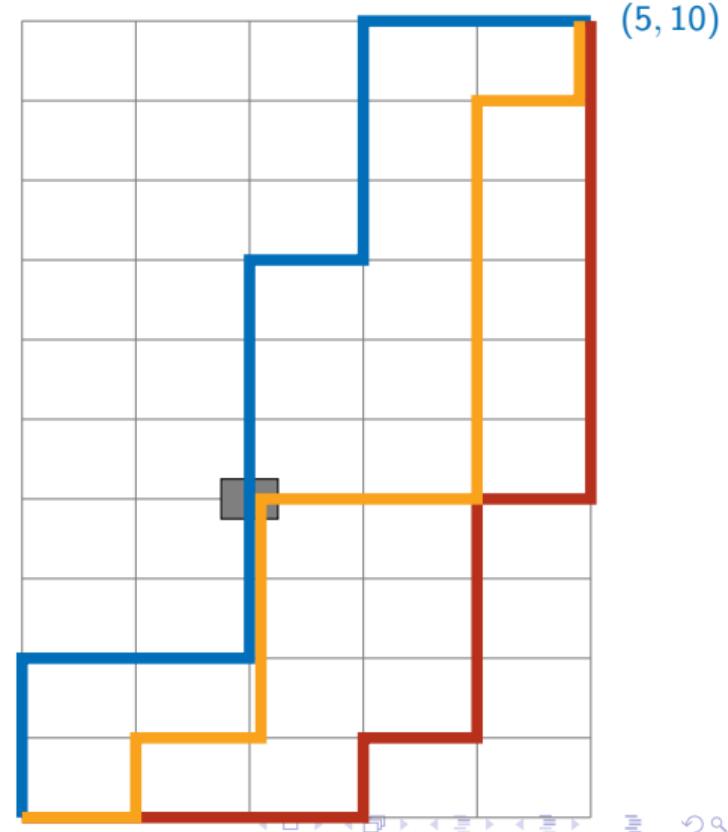
Combinatorial solution for holes

- Discard paths passing through $(2, 4)$



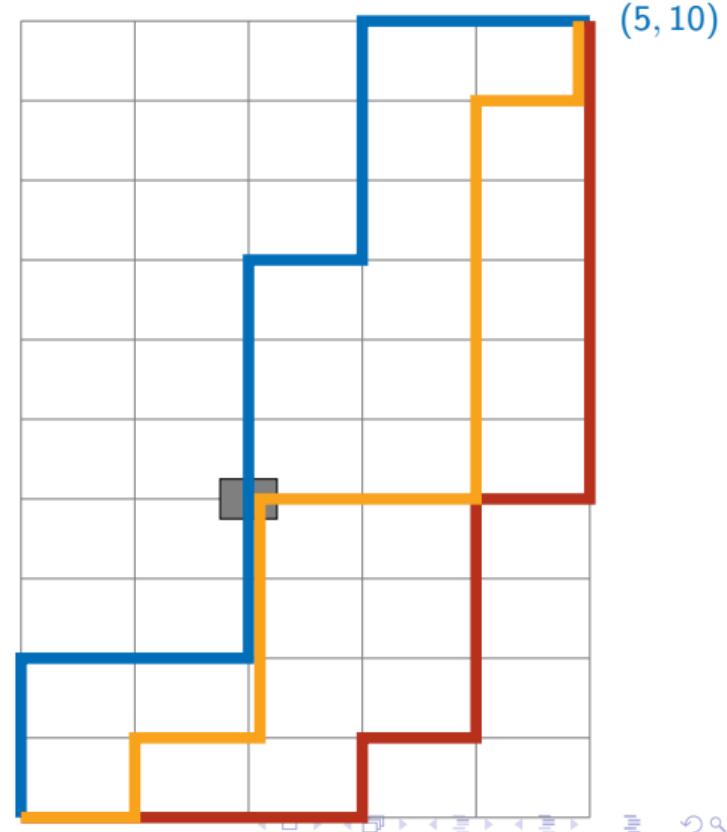
Combinatorial solution for holes

- Discard paths passing through $(2, 4)$
- Every path via $(2, 4)$ combines a path from $(0, 0)$ to $(2, 4)$ with a path from $(2, 4)$ to $(5, 10)$
 - Count these separately
 - $\binom{2+4}{2} = 15$ paths $(0, 0)$ to $(2, 4)$
 - $\binom{3+6}{3} = 84$ paths $(2, 4)$ to $(5, 10)$



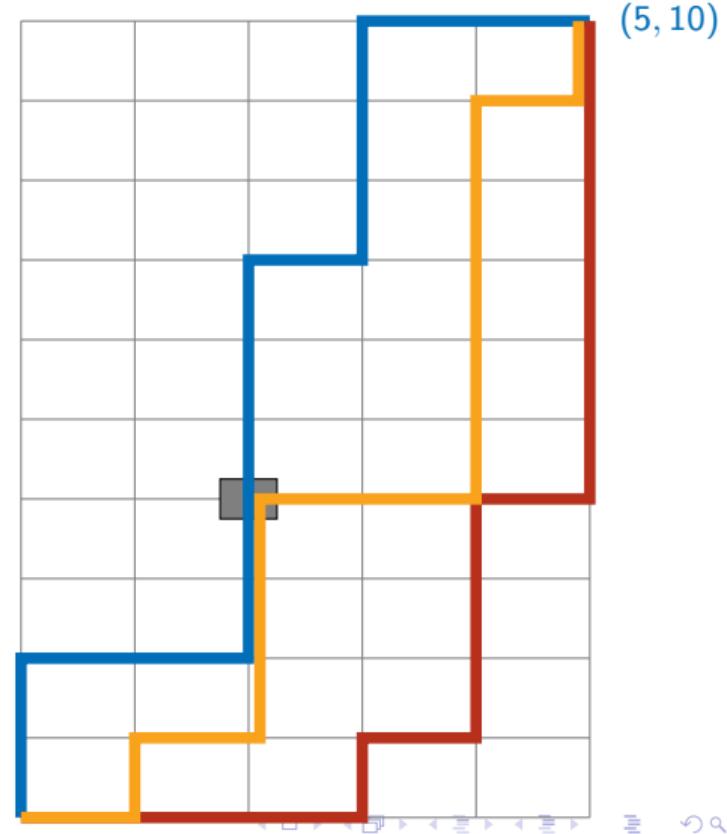
Combinatorial solution for holes

- Discard paths passing through $(2, 4)$
- Every path via $(2, 4)$ combines a path from $(0, 0)$ to $(2, 4)$ with a path from $(2, 4)$ to $(5, 10)$
 - Count these separately
 - $\binom{2+4}{2} = 15$ paths $(0, 0)$ to $(2, 4)$
 - $\binom{3+6}{3} = 84$ paths $(2, 4)$ to $(5, 10)$
- $15 \times 84 = 1260$ paths via $(2, 4)$



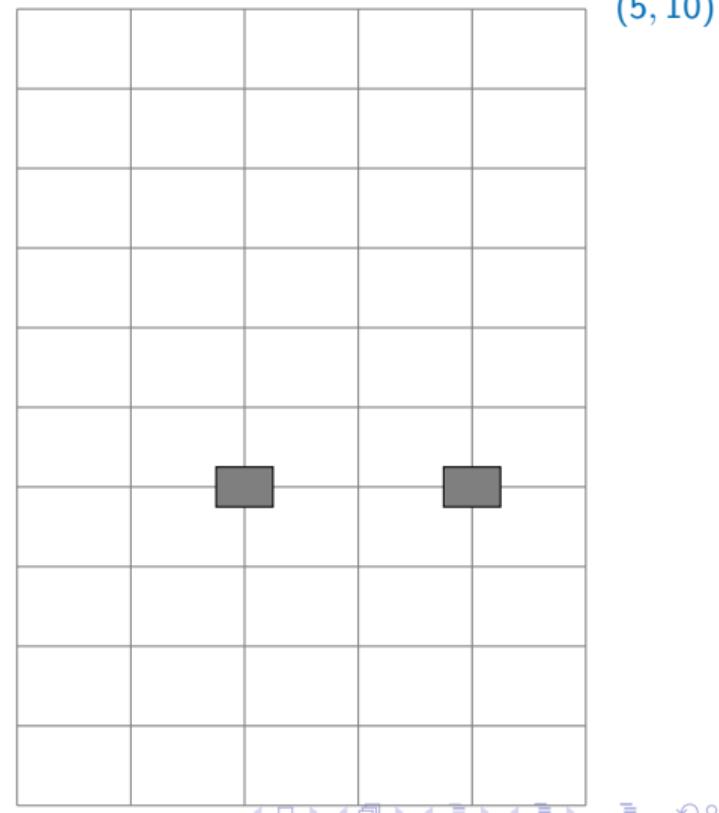
Combinatorial solution for holes

- Discard paths passing through $(2, 4)$
- Every path via $(2, 4)$ combines a path from $(0, 0)$ to $(2, 4)$ with a path from $(2, 4)$ to $(5, 10)$
 - Count these separately
 - $\binom{2+4}{2} = 15$ paths $(0, 0)$ to $(2, 4)$
 - $\binom{3+6}{3} = 84$ paths $(2, 4)$ to $(5, 10)$
- $15 \times 84 = 1260$ paths via $(2, 4)$
- $3003 - 1260 = 1743$ valid paths avoiding $(2, 4)$



More holes

- What if two intersections are blocked?

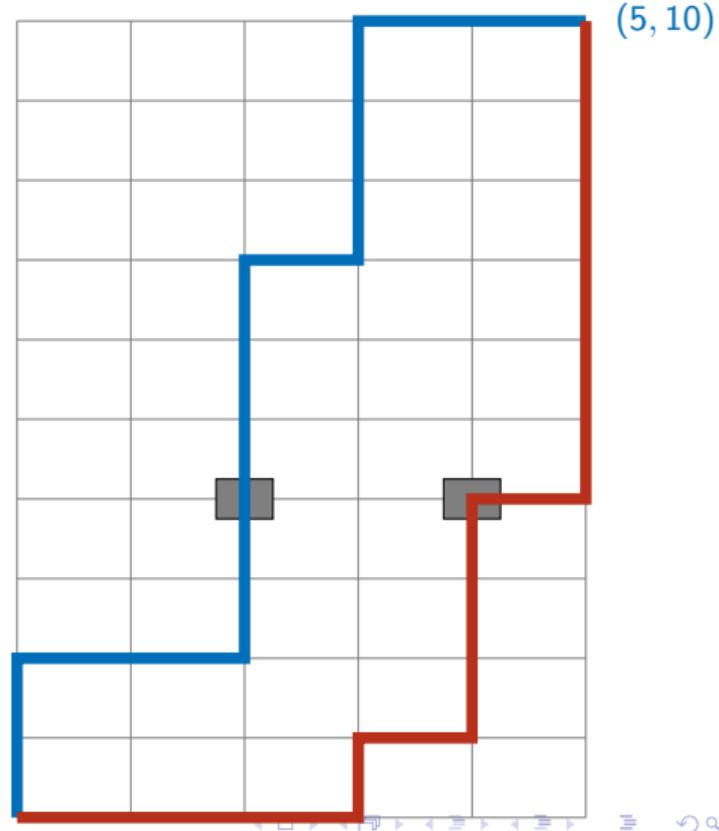


$(5, 10)$

$(0, 0)$

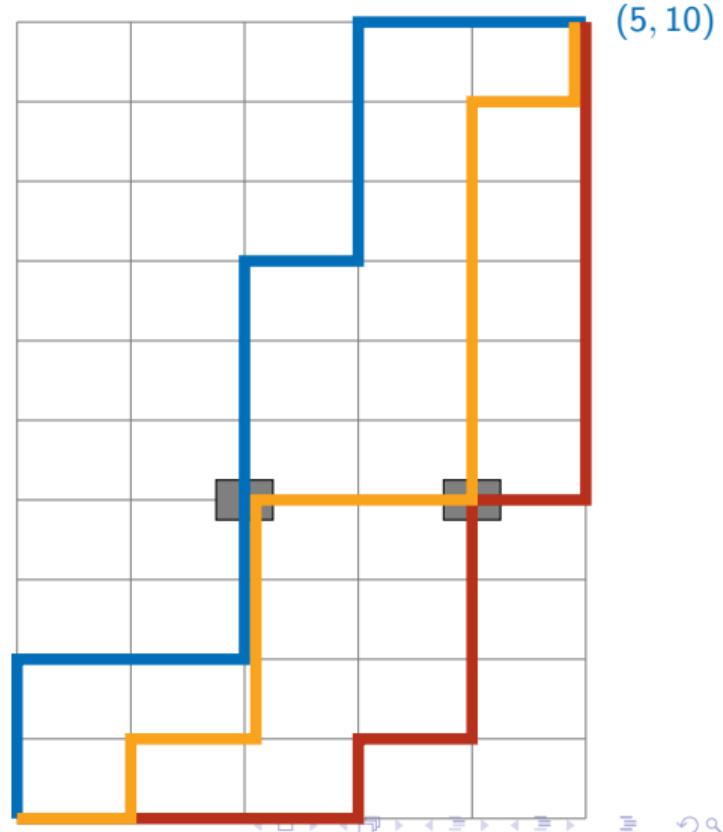
More holes

- What if two intersections are blocked?
- Discard paths via $(2, 4)$, $(4, 4)$



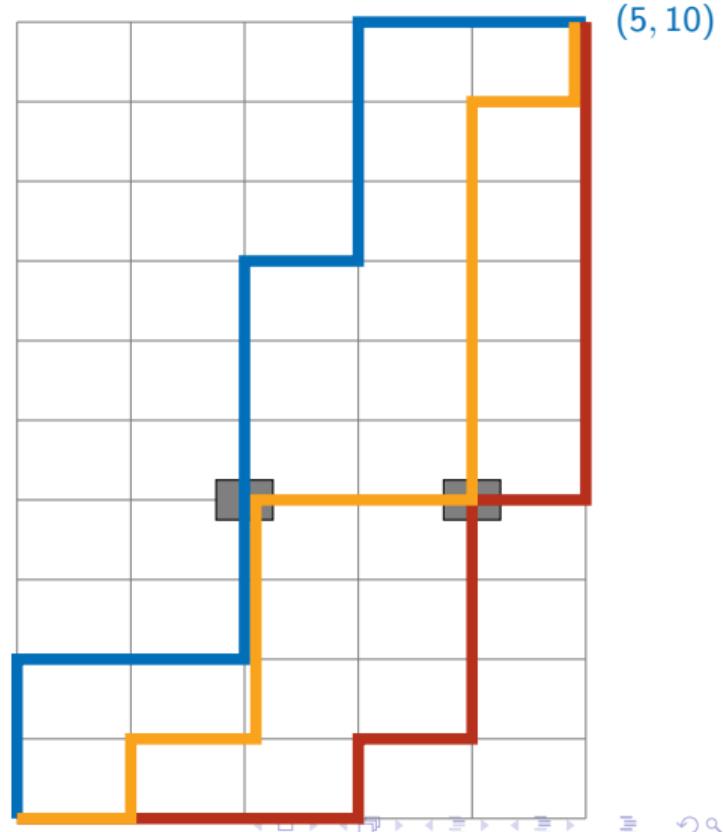
More holes

- What if two intersections are blocked?
- Discard paths via $(2, 4)$, $(4, 4)$
 - Some paths are counted twice



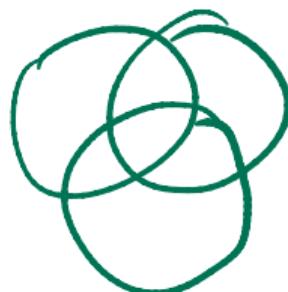
More holes

- What if two intersections are blocked?
- Discard paths via $(2, 4)$, $(4, 4)$
 - Some paths are counted twice
- Add back the paths that pass through both holes

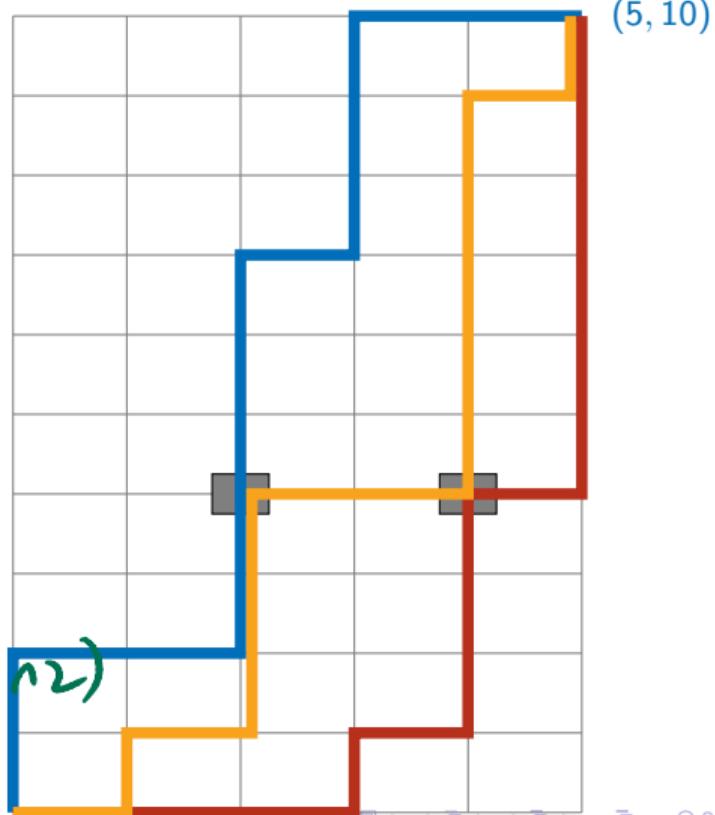


More holes

- What if two intersections are blocked?
- Discard paths via $(2, 4)$, $(4, 4)$
 - Some paths are counted twice
- Add back the paths that pass through both holes
- Inclusion-exclusion — counting is messy

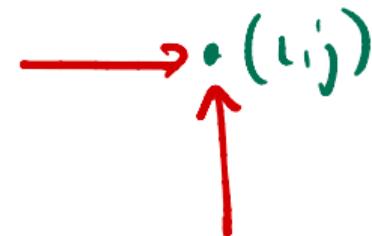


$$\begin{aligned} & (X \cup Y \cup Z) \\ & - (X \cap Y) \cup (X \cap Z) \cup (Y \cap Z) \\ & + (X \cap Y \cap Z) \end{aligned}$$



Inductive formulation

- How can a path reach (i, j)

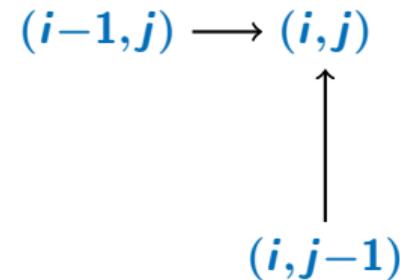


Inductive formulation

- How can a path reach (i, j)
 - Move up from $(i, j - 1)$

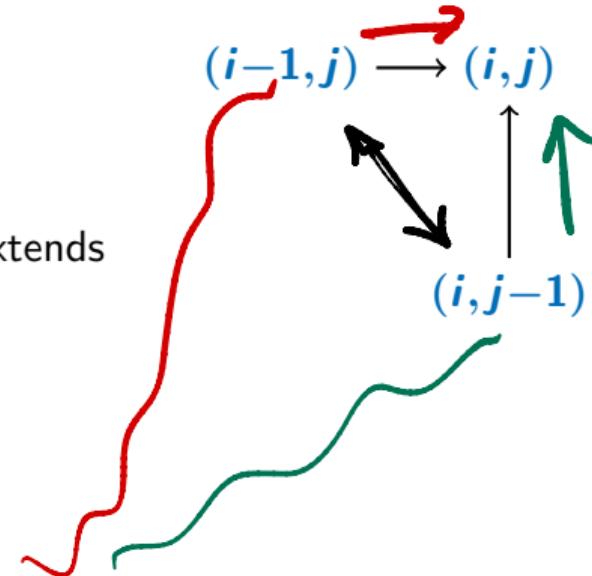
Inductive formulation

- How can a path reach (i, j)
 - Move up from $(i, j - 1)$
 - Move right from $(i - 1, j)$



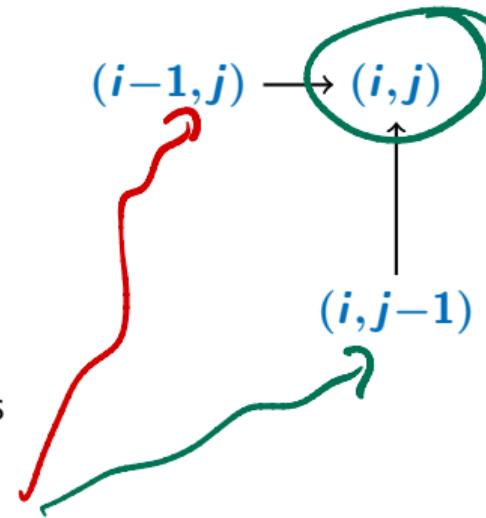
Inductive formulation

- How can a path reach (i, j)
 - Move up from $(i, j - 1)$
 - Move right from $(i - 1, j)$
- Each path to these neighbours extends to a unique path to (i, j)



Inductive formulation

- How can a path reach (i, j)
 - Move up from $(i, j - 1)$
 - Move right from $(i - 1, j)$
- Each path to these neighbours extends to a unique path to (i, j)
- Recurrence for $P(i, j)$, number of paths from $(0, 0)$ to (i, j)
 - $P(i, j) = P(i - 1, j) + P(i, j - 1)$

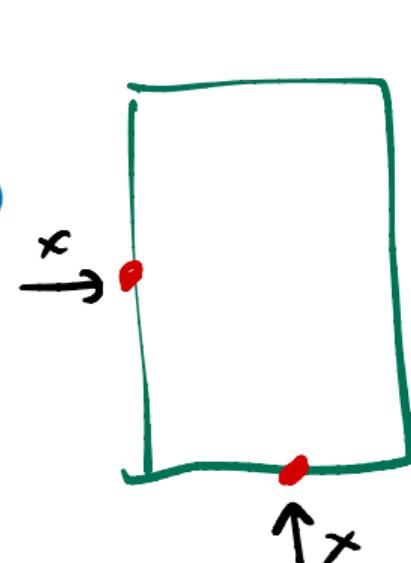


Inductive formulation

- How can a path reach (i, j)
 - Move up from $(i, j - 1)$
 - Move right from $(i - 1, j)$
 - Each path to these neighbours extends to a unique path to (i, j)
 - Recurrence for $P(i, j)$, number of paths from $(0, 0)$ to (i, j)
 - $P(i, j) = P(i - 1, j) + P(i, j - 1)$
 - $P(0, 0) = 1$ — base case

$$(i-1, j) \rightarrow (i, j)$$

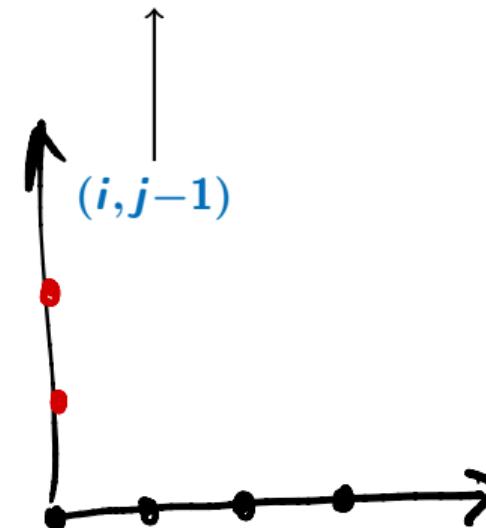
\uparrow



Inductive formulation

- How can a path reach (i, j)
 - Move up from $(i, j - 1)$
 - Move right from $(i - 1, j)$
- Each path to these neighbours extends to a unique path to (i, j)
- Recurrence for $P(i, j)$, number of paths from $(0, 0)$ to (i, j)
 - $P(i, j) = P(i - 1, j) + P(i, j - 1)$
 - $P(0, 0) = 1$ — base case
 - $P(i, 0) = P(i - 1, 0)$ — bottom row

$$(i-1, j) \longrightarrow (i, j)$$



Inductive formulation

- How can a path reach (i, j)
 - Move up from $(i, j - 1)$
 - Move right from $(i - 1, j)$
- Each path to these neighbours extends to a unique path to (i, j)
- Recurrence for $P(i, j)$, number of paths from $(0, 0)$ to (i, j)
 - $P(i, j) = P(i - 1, j) + P(i, j - 1)$
 - $P(0, 0) = 1$ — base case
 - $P(i, 0) = P(i - 1, 0)$ — bottom row
 - $P(0, j) = P(0, j - 1)$ — left column

$(i-1, j) \longrightarrow (i, j)$

\uparrow
 $(i, j-1)$

$2,5$
↑
2,4 → 3,4

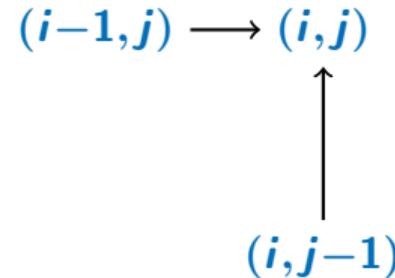
Obstacles ?

Here at $(2, 4)$

$P(2, 4) = ?$ 0

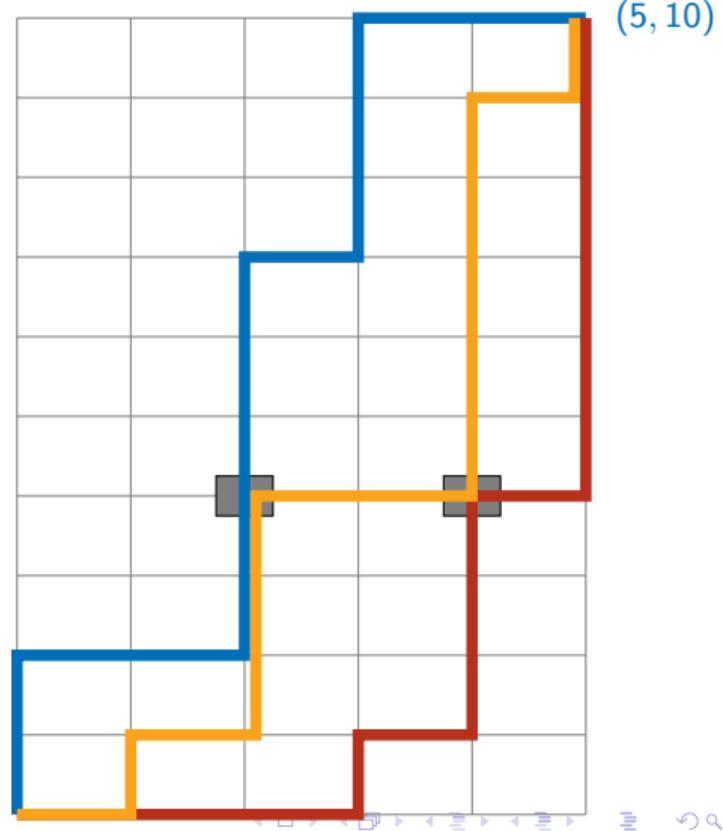
Inductive formulation

- How can a path reach (i, j)
 - Move up from $(i, j - 1)$
 - Move right from $(i - 1, j)$
- Each path to these neighbours extends to a unique path to (i, j)
- Recurrence for $P(i, j)$, number of paths from $(0, 0)$ to (i, j)
 - $P(i, j) = P(i - 1, j) + P(i, j - 1)$
 - $P(0, 0) = 1$ — base case
 - $P(i, 0) = P(i - 1, 0)$ — bottom row
 - $P(0, j) = P(0, j - 1)$ — left column
- $P(i, j) = 0$ if there is a hole at (i, j)



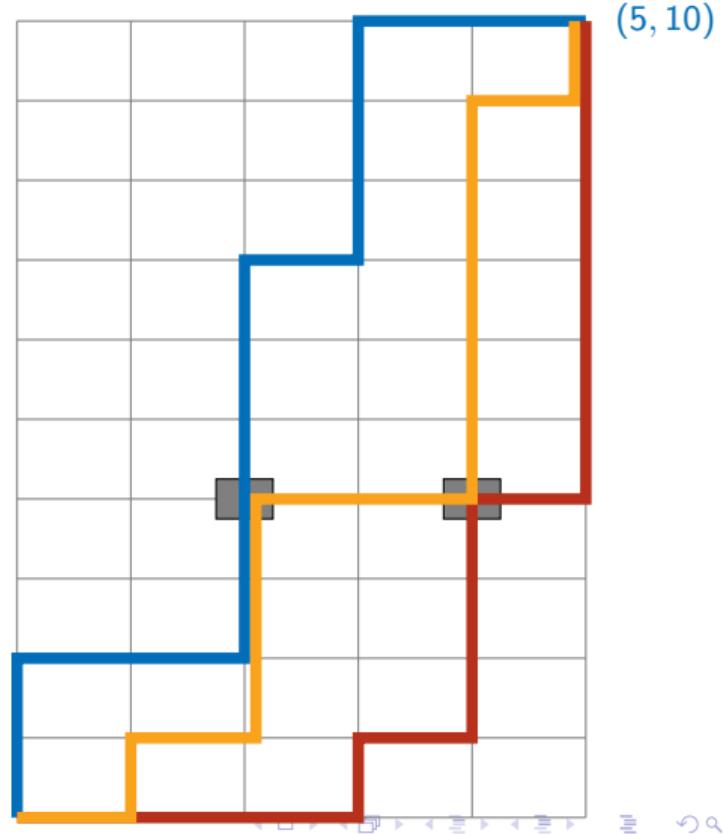
Computing $P(i,j)$

- Naive recursion recomputes same subproblem repeatedly



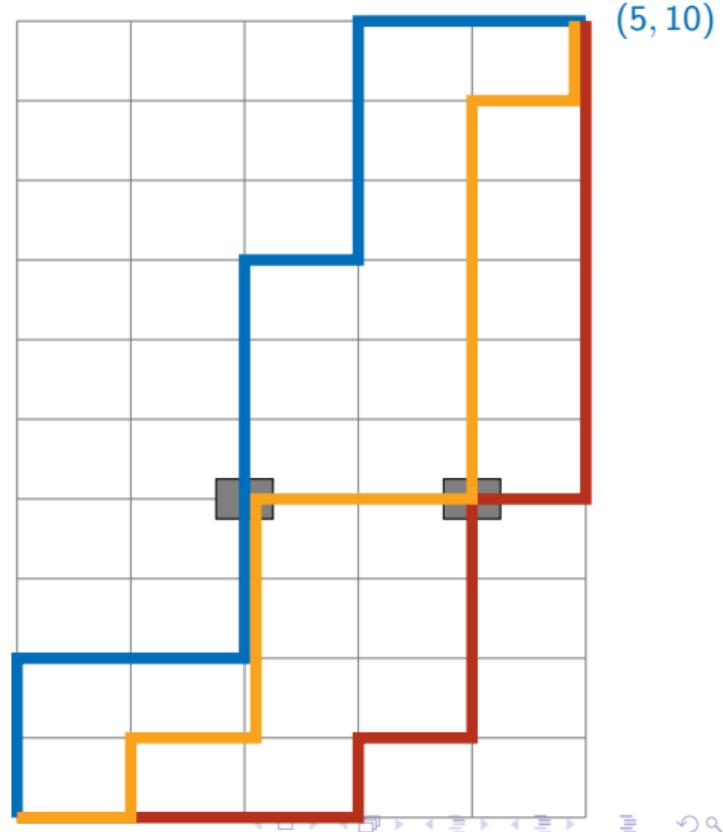
Computing $P(i,j)$

- Naive recursion recomputes same subproblem repeatedly
 - $P(5, 10)$ requires $P(4, 10), P(5, 9)$



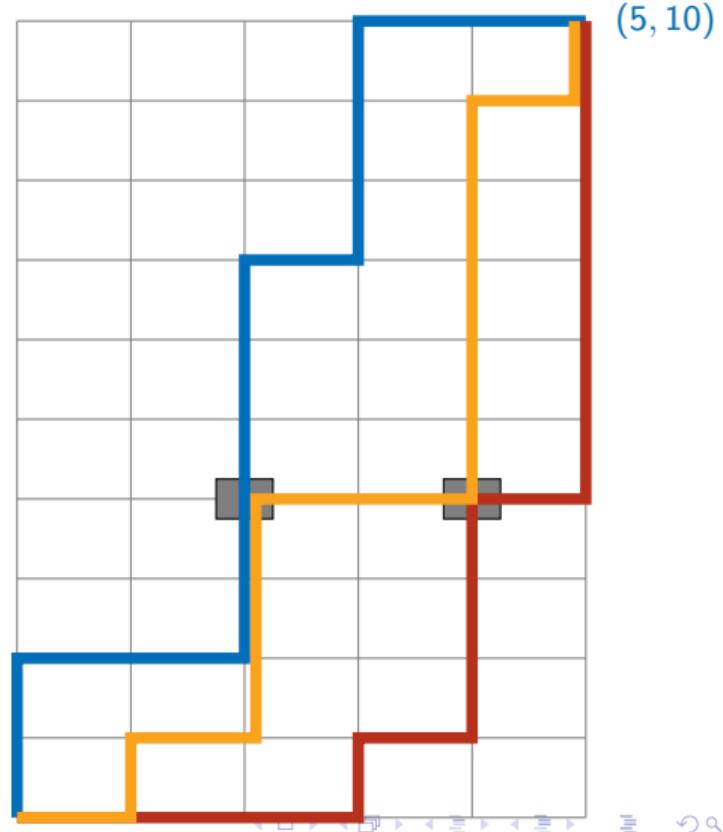
Computing $P(i, j)$

- Naive recursion recomputes same subproblem repeatedly
 - $P(5, 10)$ requires $P(4, 10), P(5, 9)$
 - Both $P(4, 10), P(5, 9)$ require $P(4, 9)$



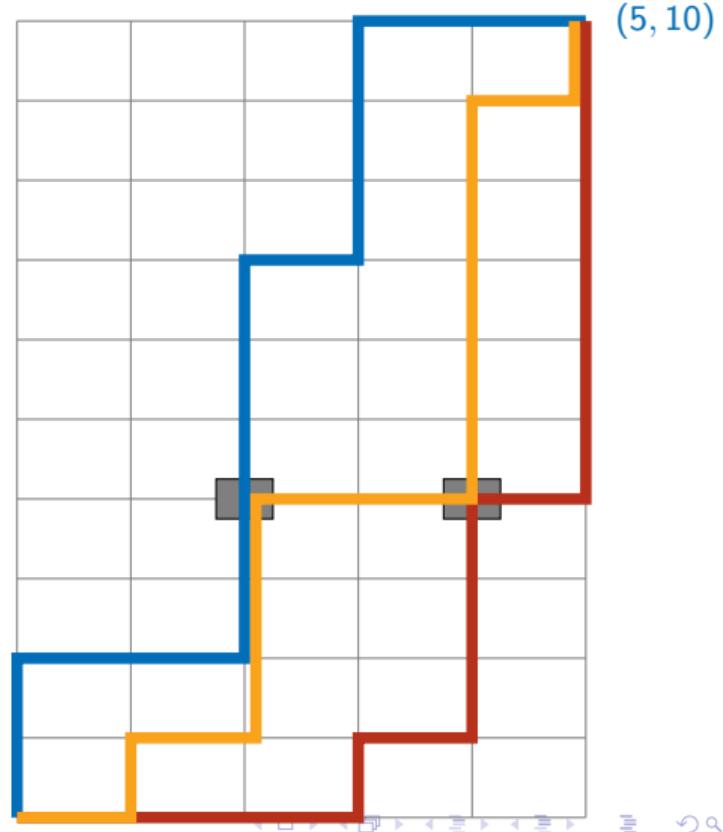
Computing $P(i,j)$

- Naive recursion recomputes same subproblem repeatedly
 - $P(5, 10)$ requires $P(4, 10), P(5, 9)$
 - Both $P(4, 10), P(5, 9)$ require $P(4, 9)$
- Use memoization . . .



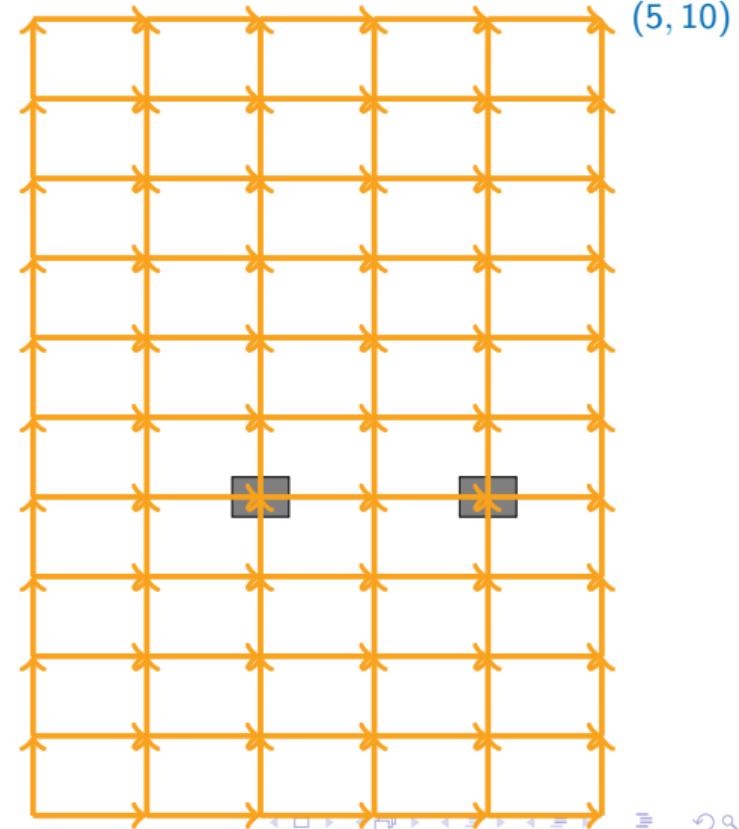
Computing $P(i,j)$

- Naive recursion recomputes same subproblem repeatedly
 - $P(5, 10)$ requires $P(4, 10), P(5, 9)$
 - Both $P(4, 10), P(5, 9)$ require $P(4, 9)$
- Use memoization ...
- ... or find a suitable order to compute the subproblems



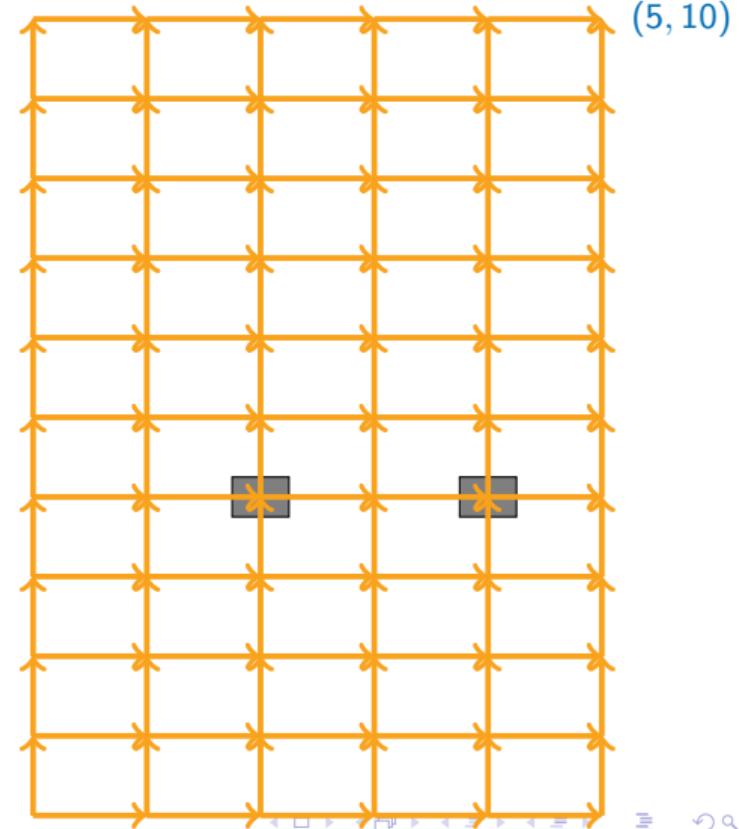
Dynamic programming

- Identify subproblem structure



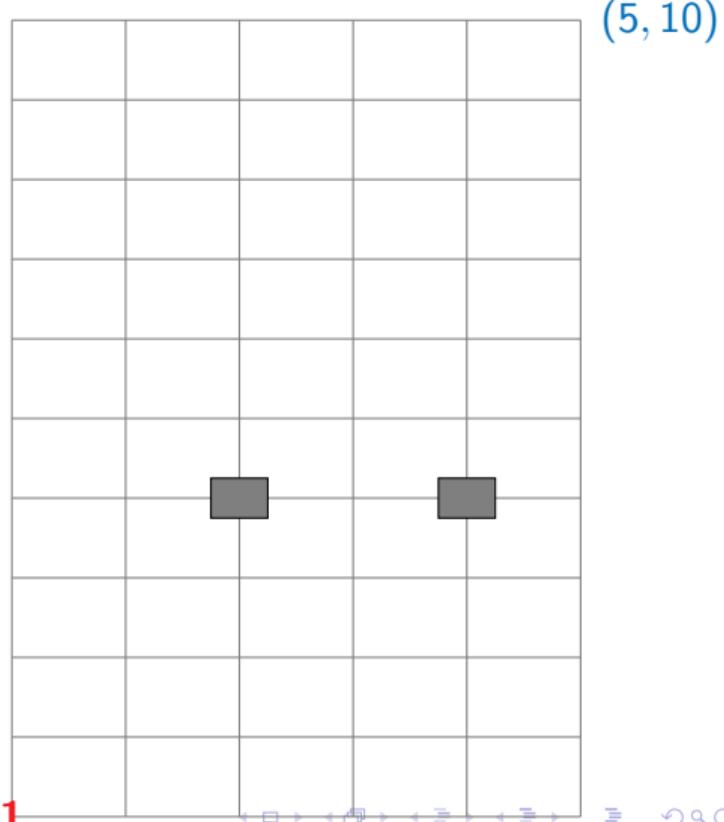
Dynamic programming

- Identify subproblem structure
- $P(0, 0)$ has no dependencies



Dynamic programming

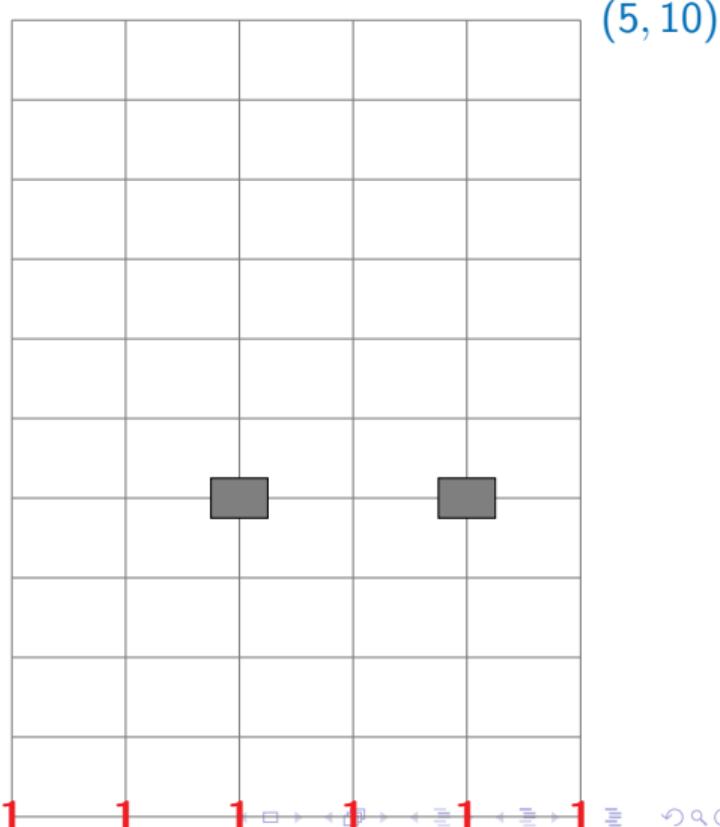
- Identify subproblem structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$



(0, 0) 1

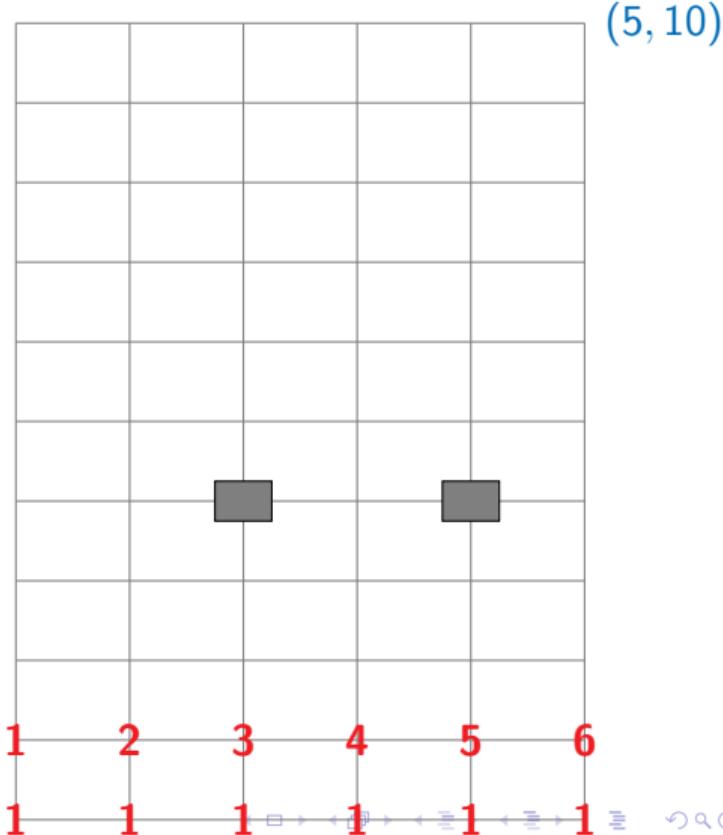
Dynamic programming

- Identify subproblem structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row



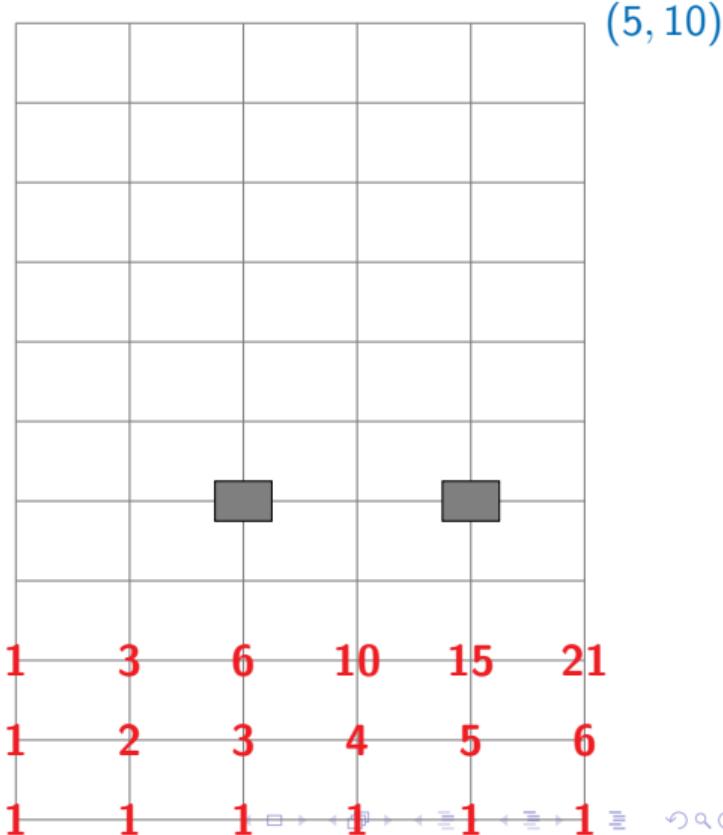
Dynamic programming

- Identify subproblem structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row



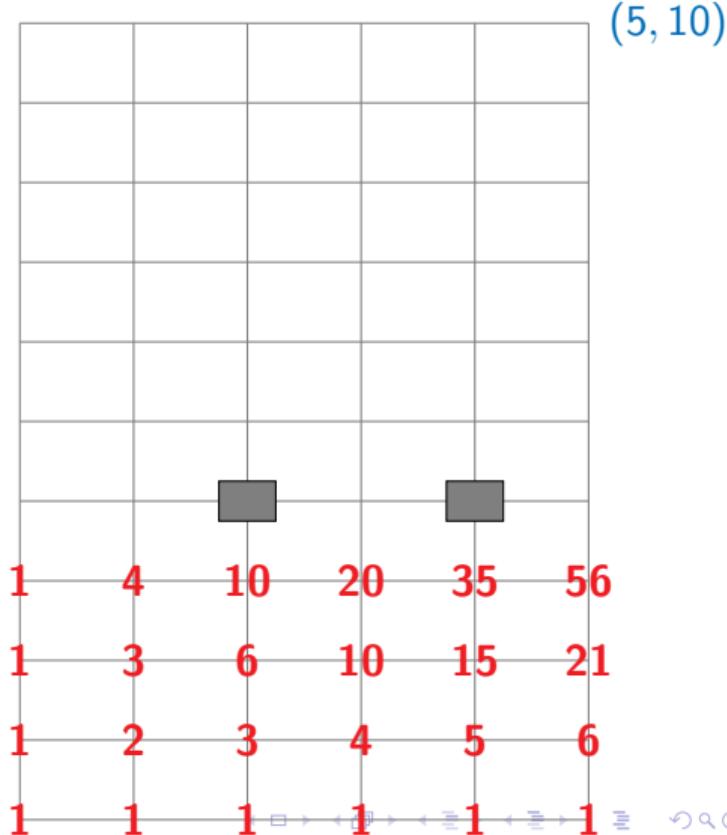
Dynamic programming

- Identify subproblem structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row



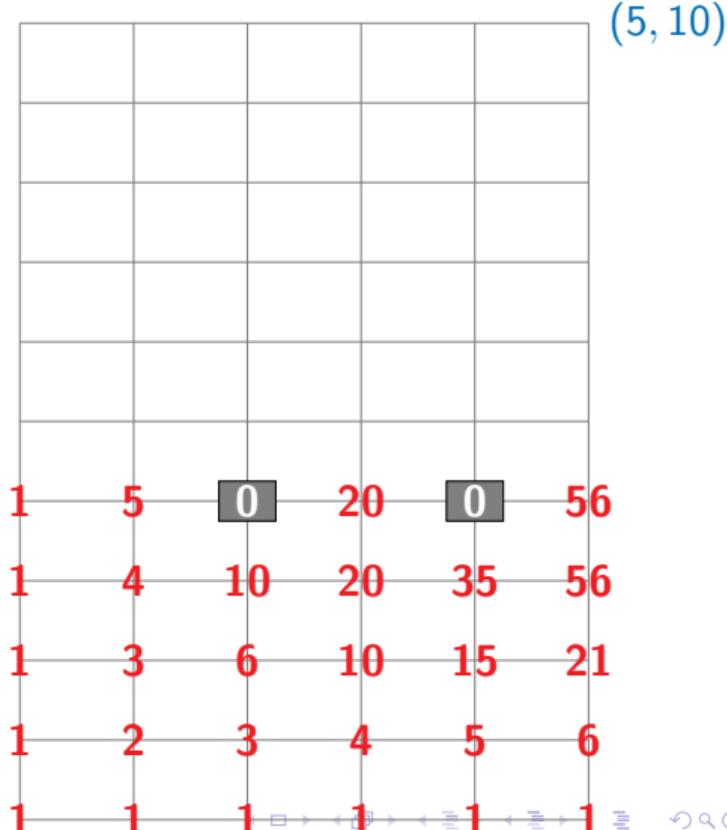
Dynamic programming

- Identify subproblem structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row



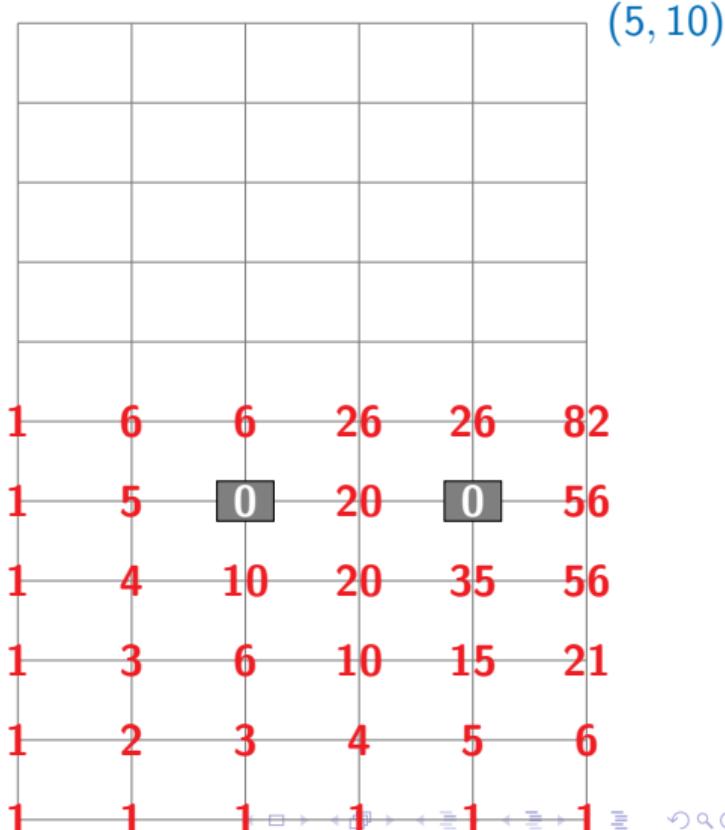
Dynamic programming

- Identify subproblem structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row



Dynamic programming

- Identify subproblem structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row



Dynamic programming

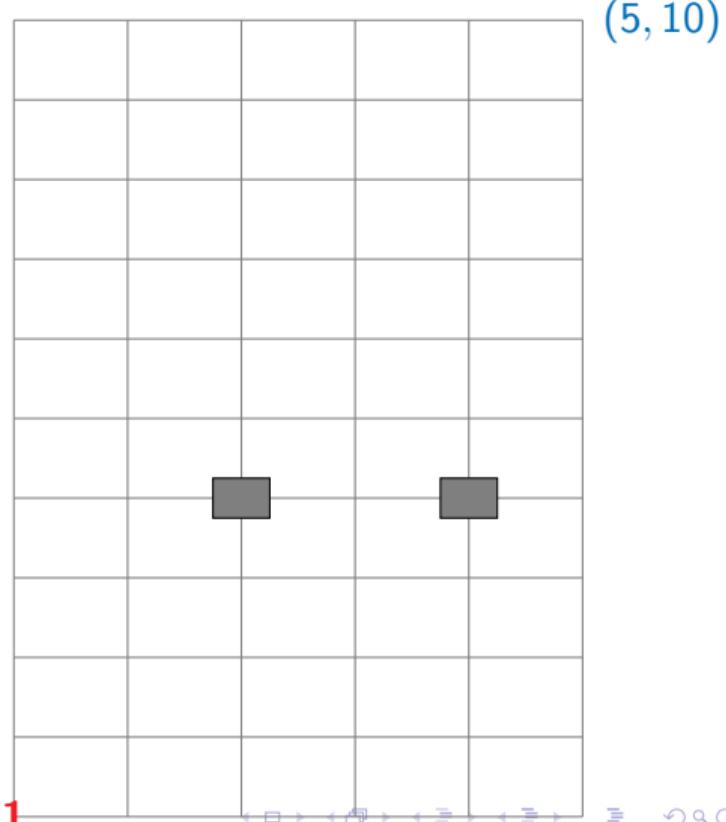
- Identify subproblem structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row

1	11	51	181	526	135	8	5, 10
1	10	40	130	345	832		
1	9	30	90	215	487		
1	8	21	60	125	272		
1	7	13	39	65	147		
1	6	6	26	26	82		
1	5	0	20	0	56		
1	4	10	20	35	56		
1	3	6	10	15	21		
1	2	3	4	5	6		
1	1	1	1	1	1		

(0, 0)

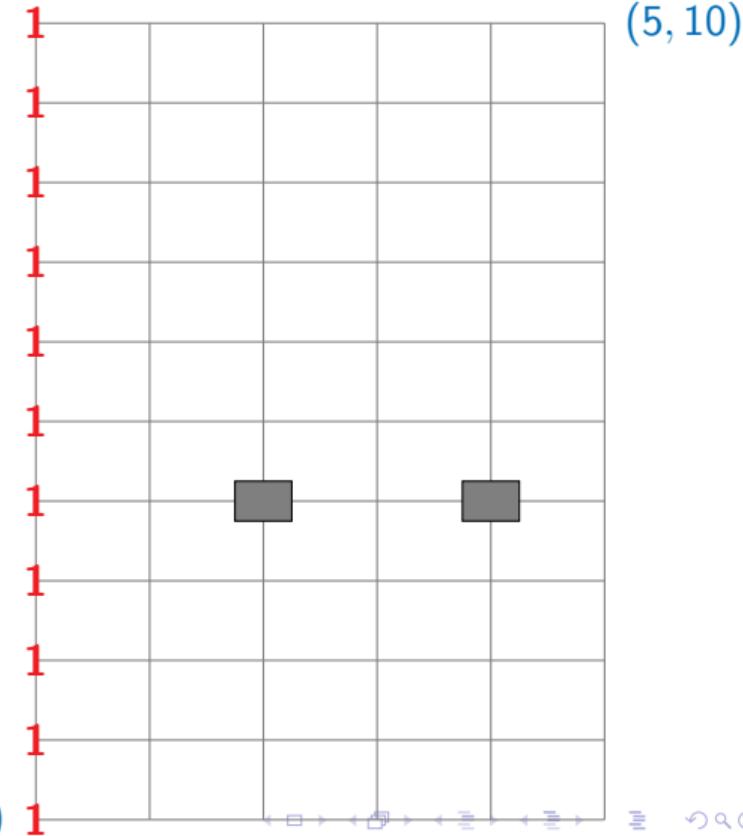
Dynamic programming

- Identify subproblem structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row
- Fill column by column



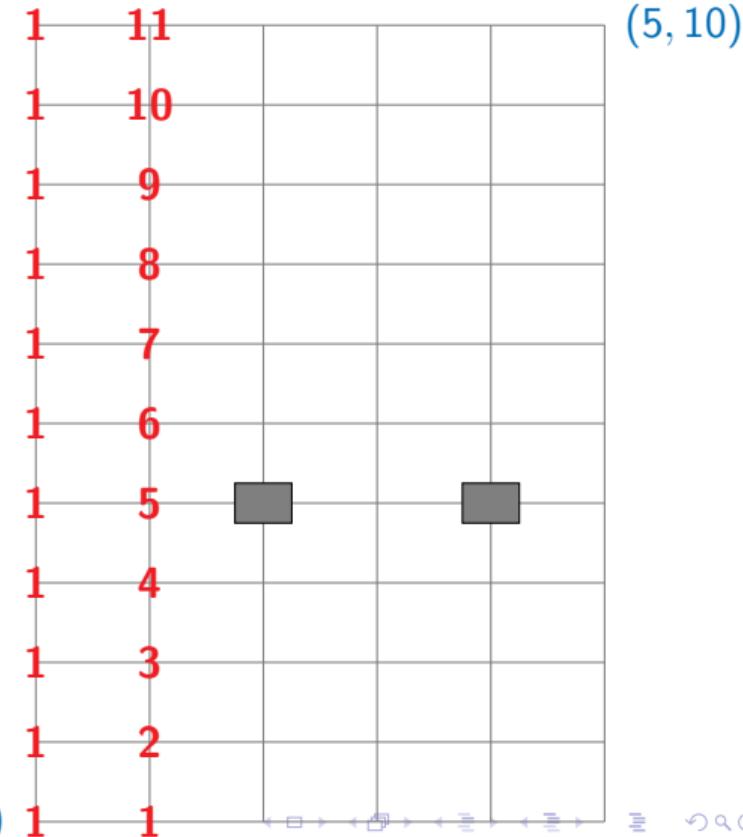
Dynamic programming

- Identify subproblem structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row
- Fill column by column



Dynamic programming

- Identify subproblem structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row
- Fill column by column



Dynamic programming

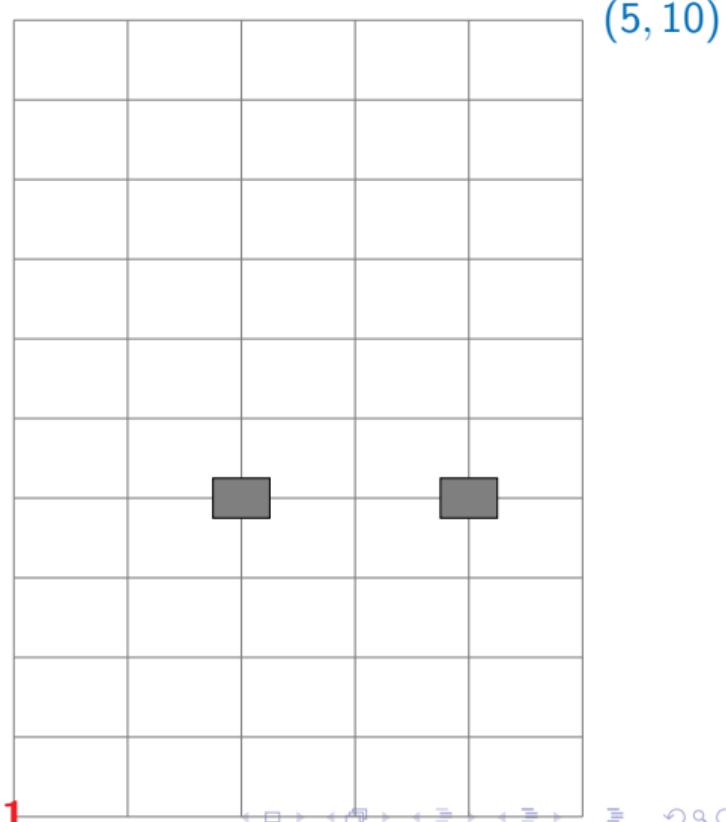
- Identify subproblem structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row
- Fill column by column

1	11	51	181	526	135	8	5, 10
1	10	40	130	345	832		
1	9	30	90	215	487		
1	8	21	60	125	272		
1	7	13	39	65	147		
1	6	6	26	26	82		
1	5	0	20	0	56		
1	4	10	20	35	56		
1	3	6	10	15	21		
1	2	3	4	5	6		
1	1	1	1	1	1		

(0, 0)

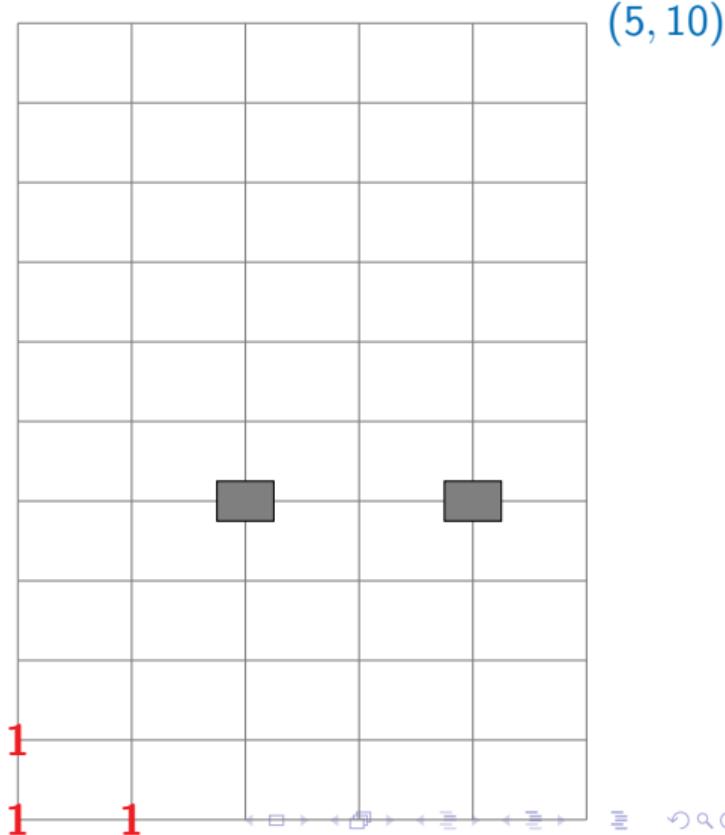
Dynamic programming

- Identify subproblem structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row
- Fill column by column
- Fill diagonal by diagonal



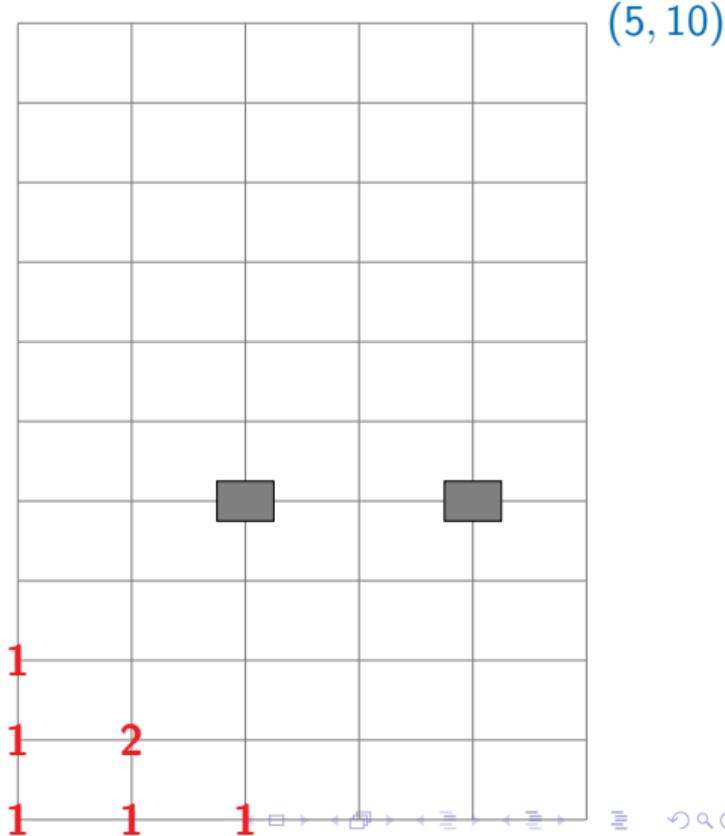
Dynamic programming

- Identify subproblem structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row
- Fill column by column
- Fill diagonal by diagonal



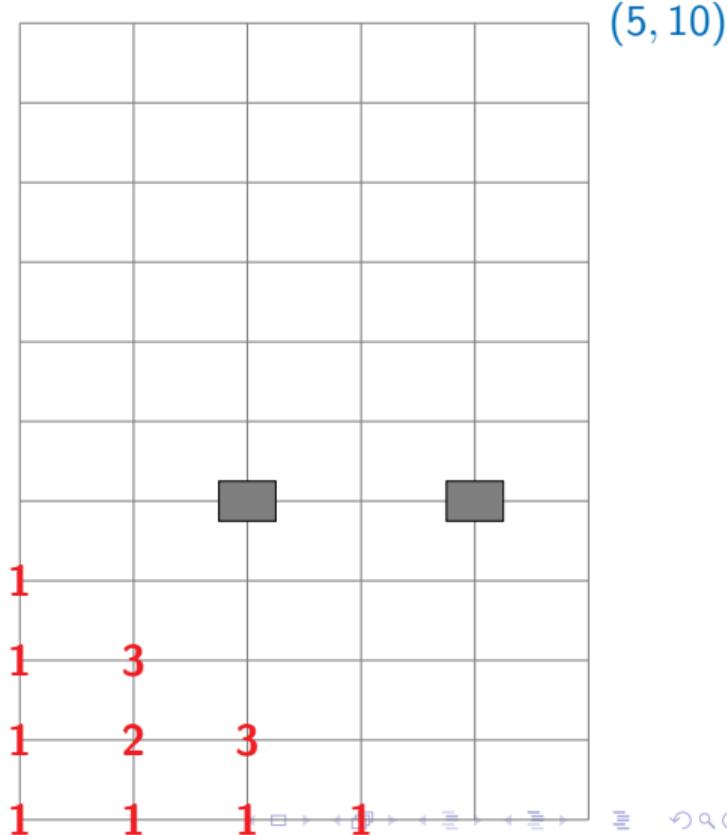
Dynamic programming

- Identify subproblem structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row
- Fill column by column
- Fill diagonal by diagonal



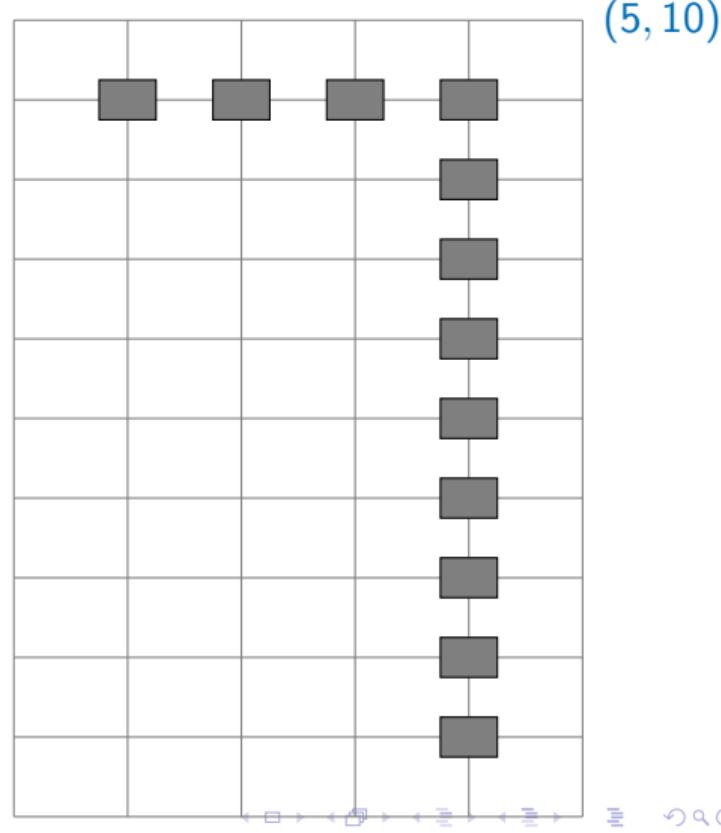
Dynamic programming

- Identify subproblem structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row
- Fill column by column
- Fill diagonal by diagonal



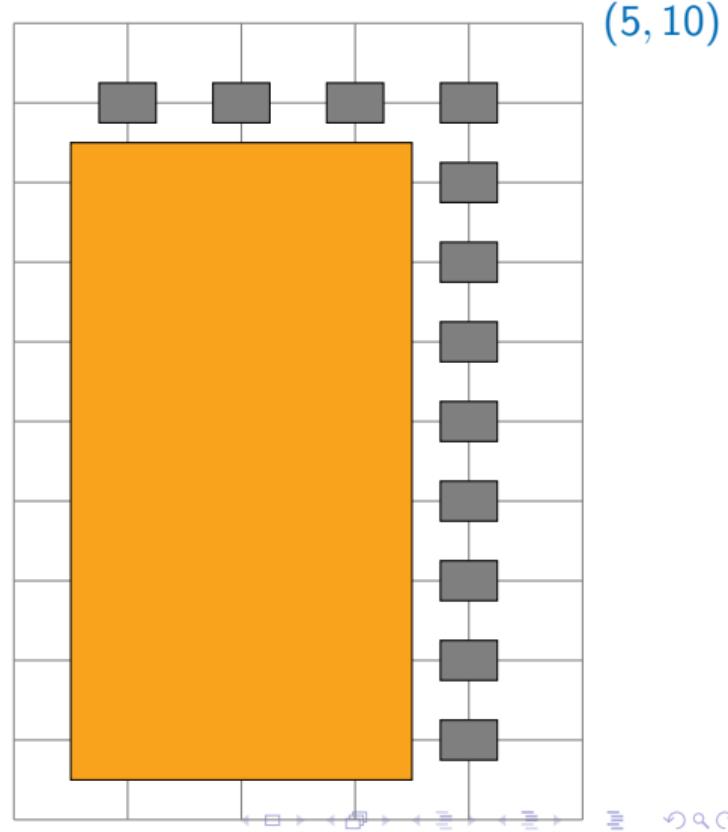
Memoization vs dynamic programming

- Barrier of holes just inside the border



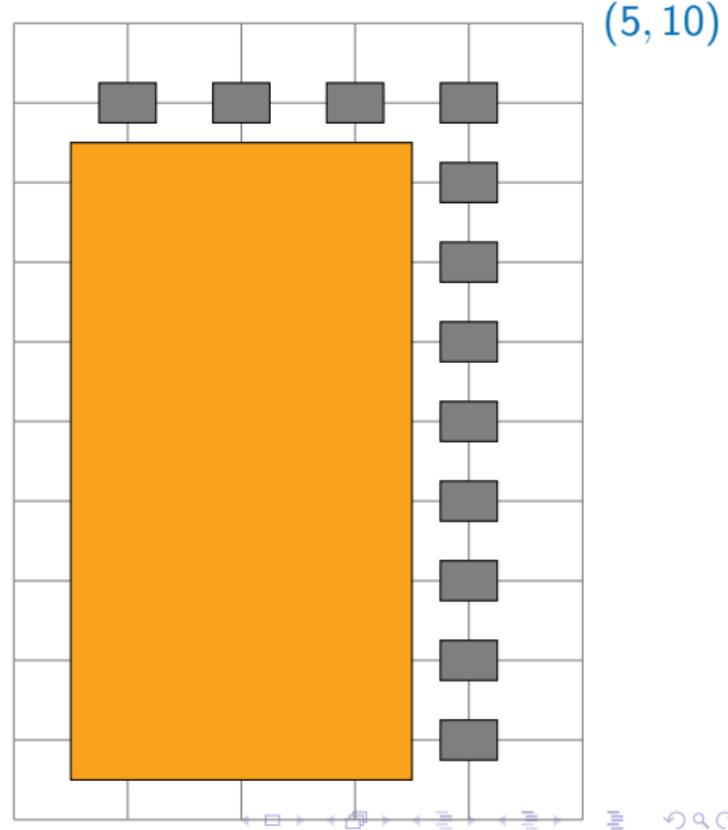
Memoization vs dynamic programming

- Barrier of holes just inside the border
- Memoization never explores the shaded region



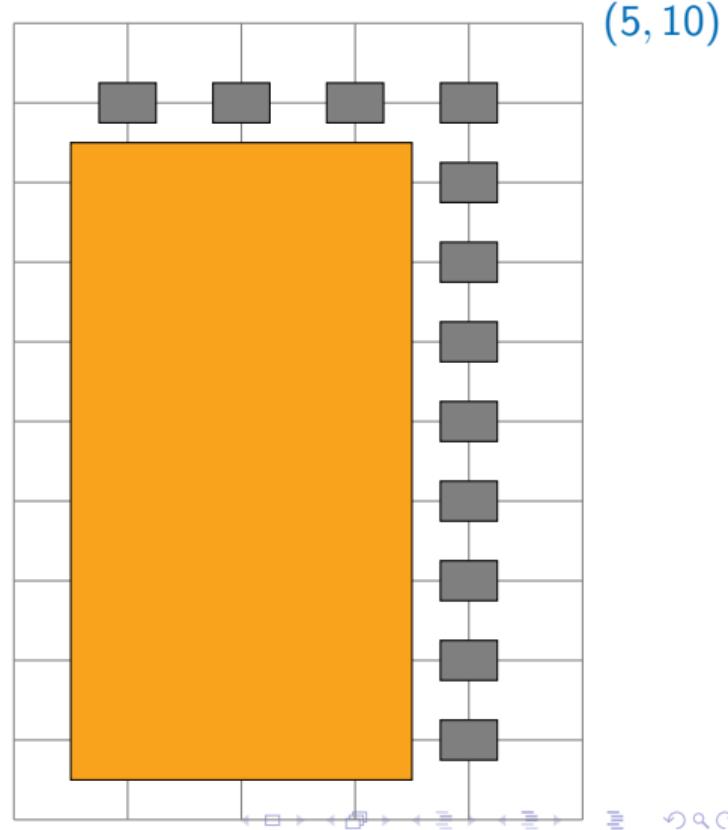
Memoization vs dynamic programming

- Barrier of holes just inside the border
- Memoization never explores the shaded region
- Memo table has $O(m + n)$ entries



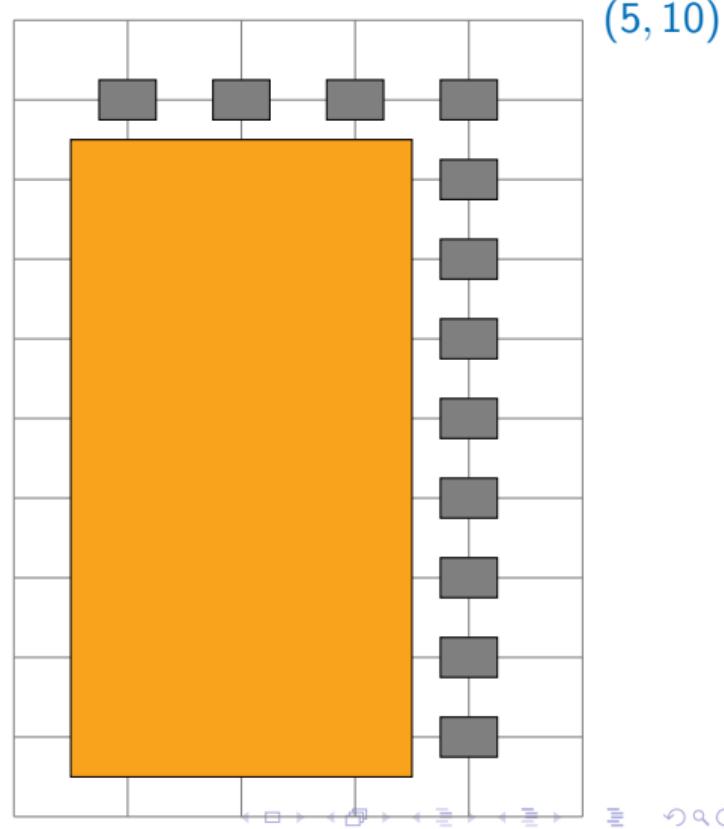
Memoization vs dynamic programming

- Barrier of holes just inside the border
- Memoization never explores the shaded region
- Memo table has $O(m + n)$ entries
- Dynamic programming blindly fills all mn cells of the table



Memoization vs dynamic programming

- Barrier of holes just inside the border
- Memoization never explores the shaded region
- Memo table has $O(m + n)$ entries
- Dynamic programming blindly fills all mn cells of the table
- Tradeoff between recursion and iteration
 - “Wasteful” dynamic programming still better in general



Longest common subword

- Given two strings, find the (length of the) longest common subword
 - "secret", "secretary" — "secret", length 6
 - "bise~~c~~t", "tri~~s~~e~~c~~t" — "ise~~c~~t", length 5
 - "bise~~c~~t", "se~~c~~ret" — "sec", length 3
 - "di~~r~~ector", "se~~c~~retary" — "ee", "re", length 2

—
—
ec

Longest common subword

- Given two strings, find the (length of the) longest common subword
 - "secret", "secretary" — "secret", length 6
 - "bisect", "trisect" — "isect", length 5
 - "bisect", "secret" — "sec", length 3
 - "director", "secretary" — "ee", "re", length 2
- Formally
 - $u = a_0 a_1 \dots a_{m-1}$
 - $v = b_0 b_1 \dots b_{n-1}$

Longest common subword

- Given two strings, find the (length of the) longest common subword
 - "secret", "secretary" — "secret", length 6
 - "bisect", "trisect" — "isect", length 5
 - "bisect", "secret" — "sec", length 3
 - "director", "secretary" — "ee", "re", length 2
- Formally
 - $u = a_0 a_1 \dots a_{m-1}$
 - $v = b_0 b_1 \dots b_{n-1}$
 - Common subword of length k — for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$

Longest common subword

- Given two strings, find the (length of the) longest common subword
 - "secret", "secretary" — "secret", length 6
 - "bise~~c~~t", "tri~~s~~e~~c~~t" — "ise~~c~~t", length 5
 - "bise~~c~~t", "se~~c~~ret" — "sec", length 3
 - "di~~r~~ector", "se~~c~~retary" — "ee", "re", length 2
- Formally
 - $u = a_0 a_1 \dots a_{m-1}$
 - $v = b_0 b_1 \dots b_{n-1}$
 - Common subword of length k — for some positions i and j ,
 $a_i a_{i+1} \dots a_{i+k-1} = b_j b_{j+1} \dots b_{j+k-1}$
 - Find the largest such k — length of the longest common subword

Brute force

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$

Brute force

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- Try every pair of starting positions i in u , j in v
 - Match $(a_i, b_j), (a_{i+1}, b_{j+1}), \dots$ as far as possible
 - Keep track of longest match

Brute force

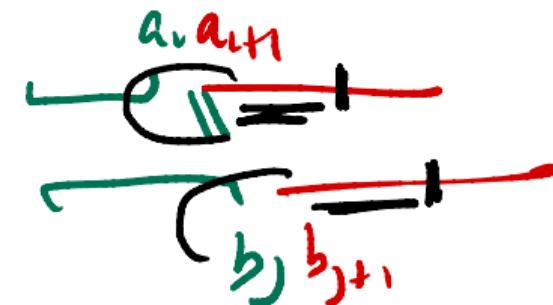
- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- Try every pair of starting positions i in u , j in v
 - Match $(a_i, b_j), (a_{i+1}, b_{j+1}), \dots$ as far as possible
 - Keep track of longest match
- Assuming $m > n$, this is $O(mn^2)$
 - mn pairs of starting positions
 - From each starting position, scan could be $O(n)$

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} \dots a_{i+k-1} = b_j b_{j+1} \dots b_{j+k-1}$
- $LCW(i, j)$ — length of longest common subword in $a_i a_{i+1} \dots a_{m-1}$, $b_j b_{j+1} \dots b_{n-1}$
 - If $a_i \neq b_j$, $LCW(i, j) = 0$
 - If $\underline{a_i} = \underline{b_j}$, $LCW(i, j) = 1 + LCW(i+1, j+1)$



Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- $LCW(i, j)$ — length of longest common subword in $a_i a_{i+1} \dots a_{m-1}$, $b_j b_{j+1} \dots b_{n-1}$
 - If $a_i \neq b_j$, $LCW(i, j) = 0$
 - If $a_i = b_j$, $LCW(i, j) = 1 + LCW(i+1, j+1)$
 - Base case: $LCW(m, n) = 0$

$\rightarrow LCW(u[i :], v[j :])$

$u[m :]$, $v[n :]$

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- $LCW(i, j)$ — length of longest common subword in $a_i a_{i+1} \dots a_{m-1}$, $b_j b_{j+1} \dots b_{n-1}$
 - If $a_i \neq b_j$, $LCW(i, j) = 0$
 - If $a_i = b_j$, $LCW(i, j) = 1 + LCW(i+1, j+1)$
 - Base case: $LCW(m, n) = 0$
 - In general, $LCW(i, n) = 0$ for all $0 \leq i \leq m$

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- $LCW(i, j)$ — length of longest common subword in $a_i a_{i+1} \dots a_{m-1}$, $b_j b_{j+1} \dots b_{n-1}$
 - If $a_i \neq b_j$, $LCW(i, j) = 0$
 - If $a_i = b_j$, $LCW(i, j) = 1 + LCW(i+1, j+1)$
 - Base case: $LCW(m, n) = 0$
 - In general, $LCW(i, n) = 0$ for all $0 \leq i \leq m$
 - In general, $LCW(m, j) = 0$ for all $0 \leq j \leq n$

Subproblem dependency

- Subproblems are $LCW(i, j)$, for
 $0 \leq i \leq m, 0 \leq j \leq n$

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values

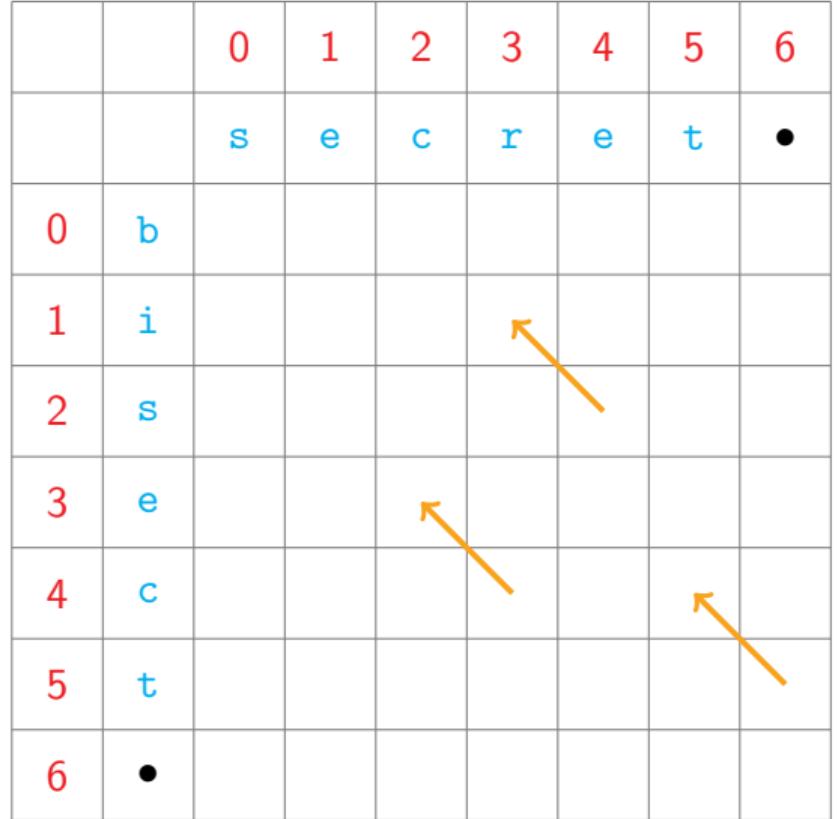
A grid diagram illustrating subproblem dependency in dynamic programming. The grid has 8 columns labeled 0 through 6 at the top, and 7 rows labeled 0 through 6 on the left. The first column contains a red question mark. The second column contains the letters 'b', 'i', 's', 'e', 'c', 't', and a black dot. The third column and beyond are empty. Handwritten annotations include a green arrow pointing right from the first column, a green arrow pointing down from the first row, and the letters 'i' and 'j' written vertically near the top-left corner of the grid.

?	b						
0	i						
1	s						
2	e						
3	c						
4	t						
5	•						
6							

Subproblem dependency

- Subproblems are $LCW(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCW(i,j)$ depends on $LCW(i+1,j+1)$

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							



Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b								0
1	i								0
2	s								0
3	e								0
4	c								0
5	t								0
6	•								0

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b						0	0
1	i						0	0
2	s						0	0
3	e						0	0
4	c						0	0
5	t						1	0
6	•						0	0

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b					0	0	0
1	i					0	0	0
2	s					0	0	0
3	e					1	0	0
4	c					0	0	0
5	t					0	1	0
6	•					0	0	0

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b				0	0	0	0
1	i				0	0	0	0
2	s				0	0	0	0
3	e				0	1	0	0
4	c				0	0	0	0
5	t				0	0	1	0
6	•				0	0	0	0

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b				0	0	0	0
1	i				0	0	0	0
2	s				0	0	0	0
3	e				0	0	1	0
4	c				1	0	0	0
5	t				0	0	0	1
6	•				0	0	0	0

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	0	0	0	0	0	0	0
3	e	2	0	0	1	0	0	0
4	c	0	1	0	0	0	0	0
5	t	0	0	0	0	1	0	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

Reading off the solution

- Find entry (i, j) with largest LCW value

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

Reading off the solution

- Find entry (i, j) with largest LCW value
- Read off the actual subword diagonally

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

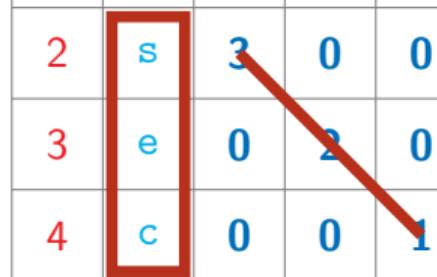
Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

Reading off the solution

- Find entry (i, j) with largest LCW value
- Read off the actual subword diagonally

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b	0	0	0	0	0	0	0	
1	i	0	0	0	0	0	0	0	
2	s	3	0	0	0	0	0	0	
3	e	0	2	0	0	1	0	0	
4	c	0	0	1	0	0	0	0	
5	t	0	0	0	0	0	1	0	
6	•	0	0	0	0	0	0	0	



Implementation

```
def LCW(u,v):
    import numpy as np
    (m,n) = (len(u),len(v))
    lcw = np.zeros((m+1,n+1))

    maxlcw = 0

    for j in range(n-1,-1,-1):
        for i in range(m-1,-1,-1):
            if u[i] == v[j]:
                lcw[i,j] = 1 + lcw[i+1,j+1]
            else:
                lcw[i,j] = 0
            if lcw[i,j] > maxlcw:
                maxlcw = lcw[i,j]

    return(maxlcw)
```

Implementation

```
def LCW(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    lcw = np.zeros((m+1,n+1))  
  
    maxlcw = 0  
  
    for j in range(n-1,-1,-1):  
        for i in range(m-1,-1,-1):  
            if u[i] == v[j]:  
                lcw[i,j] = 1 + lcw[i+1,j+1]  
            else:  
                lcw[i,j] = 0  
            if lcw[i,j] > maxlcw:  
                maxlcw = lcw[i,j]  
  
    return(maxlcw)
```

Complexity

Implementation

```
def LCW(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    lcw = np.zeros((m+1,n+1))  
  
    maxlcw = 0  
  
    for j in range(n-1,-1,-1):  
        for i in range(m-1,-1,-1):  
            if u[i] == v[j]:  
                lcw[i,j] = 1 + lcw[i+1,j+1]  
            else:  
                lcw[i,j] = 0  
            if lcw[i,j] > maxlcw:  
                maxlcw = lcw[i,j]  
  
    return(maxlcw)
```

Complexity

- Recall that brute force was $O(mn^2)$

Implementation

```
def LCW(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    lcw = np.zeros((m+1,n+1))  
  
    maxlcw = 0  
  
    for j in range(n-1,-1,-1):  
        for i in range(m-1,-1,-1):  
            if u[i] == v[j]:  
                lcw[i,j] = 1 + lcw[i+1,j+1]  
            else:  
                lcw[i,j] = 0  
            if lcw[i,j] > maxlcw:  
                maxlcw = lcw[i,j]  
  
    return(maxlcw)
```

Complexity

- Recall that brute force was $O(mn^2)$
- Inductive solution is $O(mn)$, using dynamic programming or memoization

Implementation

```
def LCW(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    lcw = np.zeros((m+1,n+1))  
  
    maxlcw = 0  
  
    for j in range(n-1,-1,-1):  
        for i in range(m-1,-1,-1):  
            if u[i] == v[j]:  
                lcw[i,j] = 1 + lcw[i+1,j+1]  
            else:  
                lcw[i,j] = 0  
            if lcw[i,j] > maxlcw:  
                maxlcw = lcw[i,j]  
  
    return(maxlcw)
```

Complexity

- Recall that brute force was $O(mn^2)$
- Inductive solution is $O(mn)$, using dynamic programming or memoization
 - Fill a table of size $O(mn)$
 - Each table entry takes constant time to compute

Longest common subsequence

- Subsequence — can drop some letters in between
- Given two strings, find the (length of the) longest common subsequence

- "secret", "secretary" —
"secret", length 6
- "bise~~c~~t", "tri~~s~~e~~c~~t" —
"ise~~c~~t", length 5
- "bi~~s~~e~~c~~t", "se~~c~~re~~t~~t" —
"sect", length 4
- "di~~r~~e~~c~~t~~o~~r", "se~~c~~re~~t~~ta~~r~~y" —
"ectr", "retr", length 4

ee re

Longest common subsequence

- Subsequence — can drop some letters in between
- Given two strings, find the (length of the) longest common subsequence
 - "secret", "secretary" — "secret", length 6
 - "bise~~c~~t", "tri~~s~~e~~c~~t" — "ise~~c~~t", length 5
 - "bi~~s~~e~~c~~t", "secret" — "sect", length 4
 - "director", "secretary" — "ectr", "retr", length 4
- LCS is the longest path connecting non-zero LCW entries, moving right/down

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

Longest common subsequence

- Subsequence — can drop some letters in between
- Given two strings, find the (length of the) longest common subsequence
 - "secret", "secretary" — "secret", length 6
 - "bisect", "trisect" — "isect", length 5
 - "bisect", "secret" — "sect", length 4
 - "director", "secretary" — "ectr", "retr", length 4
- LCS is the longest path connecting non-zero LCW entries, moving right/down

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

The diagram shows a dynamic programming table for finding the longest common subsequence (LCS) of the strings "secret" and "secretary". The rows are labeled 0 through 6, and the columns are labeled 0 through 6. The first row and column are empty. The string "secret" is in row 2 and "secretary" is in column 2. The table contains numerical values representing the length of the LCS up to that point. A red path highlights the sequence "secret" (values 3, 0, 0, 0, 0, 0), and an orange path highlights the sequence "secretary" (values 0, 0, 2, 0, 1, 0). Both paths start at (2,2) and end at (6,6).

Applications

- Analyzing genes
 - DNA is a long string over A, T, G, C
 - Two species are similar if their DNA has long common subsequences

		0	1	2	3	4	5	6
		s	e	c	r	e	t	*
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	*	0	0	0	0	0	0	0

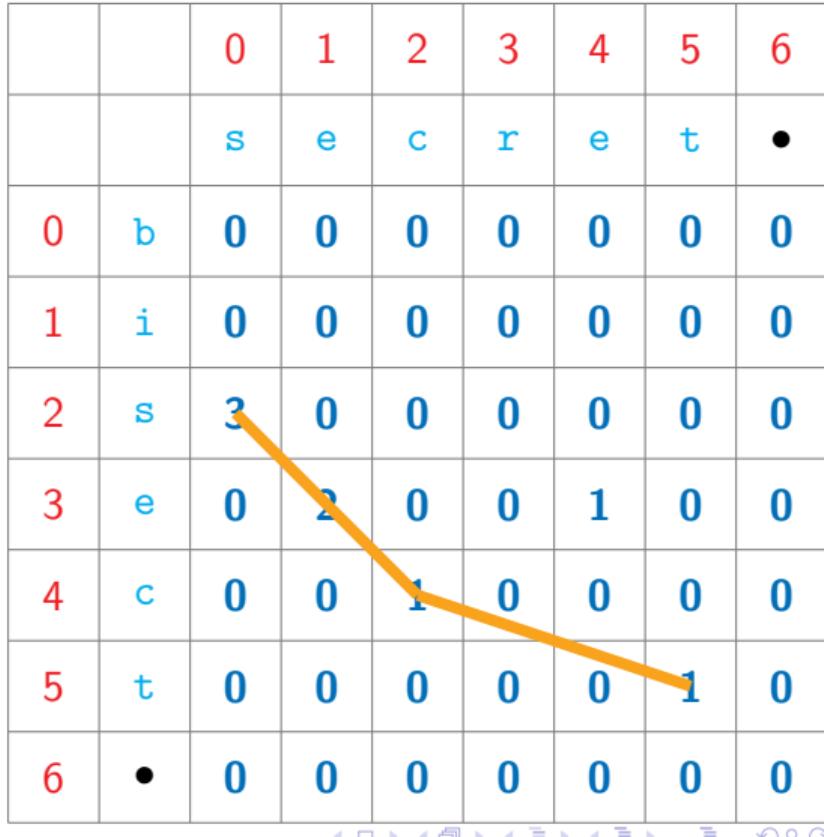
The diagram shows a 7x9 grid representing a dynamic programming table for sequence alignment. The columns are labeled 0 through 6 at the top, and the rows are labeled 0 through 6 on the left. The first two columns are empty. The third column contains 's', 'e', 'c', 'r', 'e', 't', and '*' respectively. The fourth column contains 'e', 'c', 'r', 'e', 't', and '*' respectively. The fifth column contains 'c', 'r', 'e', 't', and '*' respectively. The sixth column contains 'r', 'e', 't', and '*' respectively. The seventh column contains 'e', 't', and '*' respectively. The eighth column contains 't', and the ninth column contains '*'.

A yellow arrow points from the cell (2,3) containing '3' towards the bottom-right corner, indicating a path or sequence of operations across the grid.

Applications

- Analyzing genes
 - DNA is a long string over A, T, G, C
 - Two species are similar if their DNA has long common subsequences
- `diff` command in Unix/Linux
 - Compares text files
 - Find the longest matching subsequence of lines
 - Each line of text is a “character”

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0



The diagram illustrates a dynamic programming grid for sequence alignment. The rows represent the sequence "secret" and the columns represent the sequence "base". The grid contains numerical values representing the length of the common subsequence up to that point. A path is highlighted in orange, starting at (2,3) and ending at (6,6), representing a common subsequence of length 4.

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$

Inductive structure

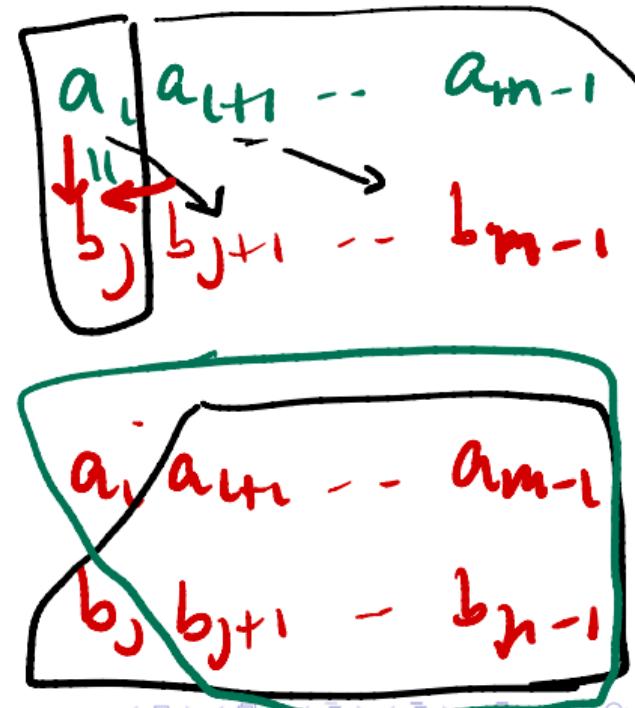
- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in
 $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in
 $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If $a_i = b_j$, $LCS(i, j) = 1 + LCS(i+1, j+1)$
 - Can assume (a_i, b_j) is part of LCS

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If $a_i = b_j$, $LCS(i, j) = 1 + LCS(i+1, j+1)$
 - Can assume (a_i, b_j) is part of LCS
- If $a_i \neq b_j$, a_i and b_j cannot both be part of the LCS



Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in
 $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If $a_i = b_j$, $LCS(i, j) = 1 + LCS(i+1, j+1)$
 - Can assume (a_i, b_j) is part of LCS
- If $a_i \neq b_j$, a_i and b_j cannot both be part of the LCS
 - Which one should we drop?

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in
 $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If $a_i = b_j$, $LCS(i, j) = 1 + LCS(i+1, j+1)$
 - Can assume (a_i, b_j) is part of LCS
- If $a_i \neq b_j$, a_i and b_j cannot both be part of the LCS
 - Which one should we drop?
 - Solve $LCS(i, j+1)$ and $LCS(i+1, j)$ and take the maximum

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in
 $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If $a_i = b_j$, $LCS(i, j) = 1 + \underline{LCS(i+1, j+1)}$
 - Can assume (a_i, b_j) is part of LCS
- If $a_i \neq b_j$, a_i and b_j cannot both be part of the LCS
 - Which one should we drop?
 - Solve $\underline{LCS(i, j+1)}$ and $\underline{LCS(i+1, j)}$ and take the maximum
- Base cases as with LCW
 - $LCS(i, n) = 0$ for all $0 \leq i \leq m$
 - $LCS(m, j) = 0$ for all $0 \leq j \leq n$

Subproblem dependency

- Subproblems are $LCS(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,

		0	1	2	3	4	5	6
		s	e	c	r	e	t	.
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	.							

The diagram shows a 7x9 grid representing a dynamic programming table for the LCS problem. The columns are labeled 0 through 6, and the rows are labeled 0 through 6. The sequence "secret" is written along the top row, and the sequence "base" is written vertically down the first column. The bottom-right cell contains a black dot. Orange arrows indicate dependencies: from (i+1, j+1) to (i, j), (i, j+1), and (i+1, j). A green path is highlighted, starting from the bottom-right cell and moving up and left, representing the reconstruction of the LCS.

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,
- No dependency for $LCS(m,n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							0
1	i							0
2	s							0
3	e							0
4	c							0
5	t							0
6	•							0

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,
- No dependency for $LCS(m,n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b						0	0
1	i						0	0
2	s						0	0
3	e						0	0
4	c						0	0
5	t						1	0
6	•						0	0

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,
- No dependency for $LCS(m,n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b					1	0	0
1	i					1	0	0
2	s					1	0	0
3	e					1	0	0
4	c					1	0	0
5	t					1	1	0
6	•					0	0	0

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,
- No dependency for $LCS(m,n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b				1	1	0	0
1	i				1	1	0	0
2	s				1	1	0	0
3	e				1	1	0	0
4	c				1	1	0	0
5	t				1	1	1	0
6	•				0	0	0	0

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,
- No dependency for $LCS(m,n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b			2	1	1	0	0
1	i			2	1	1	0	0
2	s			2	1	1	0	0
3	e			2	1	1	0	0
4	c			2	1	1	0	0
5	t			1	1	1	1	0
6	•			0	0	0	0	0

The diagram shows a dynamic programming table for the LCS problem. The table has 7 rows and 7 columns, indexed from 0 to 6. The first two columns are for sequences 's' and 'e'. The next three columns are for sequence 'c'. The last two columns are for sequences 'r', 'e', and 't'. The last row contains a symbol '•'. The values in the table represent the lengths of the longest common subsequence found so far. A red checkmark is placed above the 'c' in row 4, column 2. A red circle highlights the value '2' in row 4, column 3. A red arrow points from this circled '2' to the '1' in row 5, column 3.

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,
- No dependency for $LCS(m,n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b		3	2	1	1	0	0
1	i		3	2	1	1	0	0
2	s		3	2	1	1	0	0
3	e		3	2	1	1	0	0
4	c		2	2	1	1	0	0
5	t		1	1	1	1	1	0
6	•		0	0	0	0	0	0

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,
- No dependency for $LCS(m,n)$ — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	3	2	1	1	0	0
1	i	4	3	2	1	1	0	0
2	s	4	3	2	1	1	0	0
3	e	3	3	2	1	1	0	0
4	c	2	2	2	1	1	0	0
5	t	1	1	1	1	1	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,
- No dependency for $LCS(m,n)$ — start at bottom right and fill by row, column or diagonal

Reading off the solution

- Trace back the path by which each entry was filled

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b	4	3	2	1	1	0	0	
1	i		3	2	1	1	0	0	
2	s		3	2	1	1	0	0	
3	e	3	3	2	1	1	0	0	
4	c	2	2	2	1	1	0	0	
5	t	1	1	1	1	1	1	0	
6	•	0	0	0	0	0	0	0	

The diagram shows a grid of numbers representing the lengths of the Longest Common Subsequence (LCS) for two strings. The columns are labeled 0, 1, 2, 3, 4, 5, 6 and the rows are labeled 0, 1, 2, 3, 4, 5, 6. The grid has a black border around the values. A thick orange arrow traces a path from the bottom-right cell (6,0) to the top-left cell (0,0). The path starts at (6,0) and moves diagonally up and left through cells (5,1), (4,2), (3,3), (2,4), (1,5), and finally reaches (0,6). Each cell contains its value: (6,0)=0, (5,1)=1, (4,2)=2, (3,3)=3, (2,4)=2, (1,5)=1, (0,6)=0. The first few cells (0,0) through (1,1) contain letters: (0,0)=b, (1,0)=i, (0,1)=s, (1,1)=e, (0,2)=c, (1,2)=t, (0,3)=•.

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCS(i,j)$ depends on $LCS(i+1,j+1)$, $LCS(i,j+1), LCS(i+1,j)$,
- No dependency for $LCS(m,n)$ — start at bottom right and fill by row, column or diagonal

Reading off the solution

- Trace back the path by which each entry was filled
- Each diagonal step is an element of LCS

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b	4	3	2	1	1	0	0	
1	i	4	3	2	1	1	0	0	
2	s	4	3	2	1	1	0	0	
3	e	3	3	2	1	1	0	0	
4	c	2	2	2	1	1	0	0	
5	t	1	1	1	1	1	1	0	
6	•	0	0	0	0	0	0	0	

Implementation

```
def LCS(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    lcs = np.zeros((m+1,n+1))  
  
    for j in range(n-1,-1,-1):  
        for i in range(m-1,-1,-1):  
            if u[i] == v[j]:  
                lcs[i,j] = 1 + lcs[i+1,j+1]  
            else:  
                lcs[i,j] = max(lcs[i+1,j],  
                                lcs[i,j+1])  
    return(lcs[0,0])
```

Implementation

```
def LCS(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    lcs = np.zeros((m+1,n+1))  
  
    for j in range(n-1,-1,-1):  
        for i in range(m-1,-1,-1):  
            if u[i] == v[j]:  
                lcs[i,j] = 1 + lcs[i+1,j+1]  
            else:  
                lcs[i,j] = max(lcs[i+1,j],  
                                lcs[i,j+1])  
    return(lcs[0,0])
```

Complexity

Implementation

```
def LCS(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    lcs = np.zeros((m+1,n+1))  
  
    for j in range(n-1,-1,-1):  
        for i in range(m-1,-1,-1):  
            if u[i] == v[j]:  
                lcs[i,j] = 1 + lcs[i+1,j+1]  
            else:  
                lcs[i,j] = max(lcs[i+1,j],  
                                lcs[i,j+1])  
    return(lcs[0,0])
```

Complexity

- Again $O(mn)$, using dynamic programming or memoization

Implementation

```
def LCS(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    lcs = np.zeros((m+1,n+1))  
  
    for j in range(n-1,-1,-1):  
        for i in range(m-1,-1,-1):  
            if u[i] == v[j]:  
                lcs[i,j] = 1 + lcs[i+1,j+1]  
            else:  
                lcs[i,j] = max(lcs[i+1,j],  
                                lcs[i,j+1])  
  
    return(lcs[0,0])
```

Complexity

- Again $O(mn)$, using dynamic programming or memoization
 - Fill a table of size $O(mn)$
 - Each table entry takes constant time to compute