CSE 417 Autumn 2025

Lecture 14: Intro to Dynamic Programming

Nathan Brunelle

Algorithmic Paradigms

Divide-and-conquer: Break up a problem into sub-problems (each typically a constant factor smaller), solve each sub-problem *independently*, and combine solution to sub-problems to form solution to original problem.

Dynamic programming: Break up a problem into a series of ***overlapping*** sub-problems, and build up solutions to larger and larger sub-problems.

Greedy: Build up a solution incrementally, myopically optimizing some local criterion.

Algorithm Design Techniques

Dynamic Programming:

Technique for making building solution to a problem based on solutions to smaller subproblems (recursive ideas).

The subproblems just have to be smaller, but don’t need to be a constant-factor smaller like divide and conquer.

Useful when *the same subproblems show up over and over again*

The final solution is simple iterative code when the following holds:

- *The parameters to all the subproblems are predictable in advance*

Dynamic Programming History

Bellman. [1950s] Pioneered the systematic study of dynamic programming.

"it's impossible to use dynamic in a pejorative sense"
"something not even a Congressman could object to"

Etymology

Dynamic programming = planning over time.

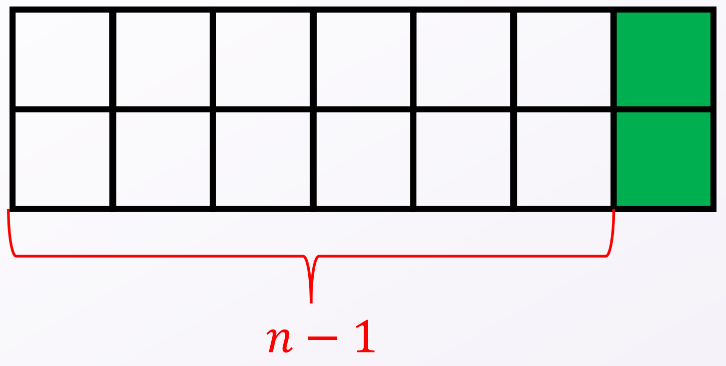
Secretary of Defense was hostile to mathematical research.

Bellman sought an impressive name to avoid confrontation.

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography.*

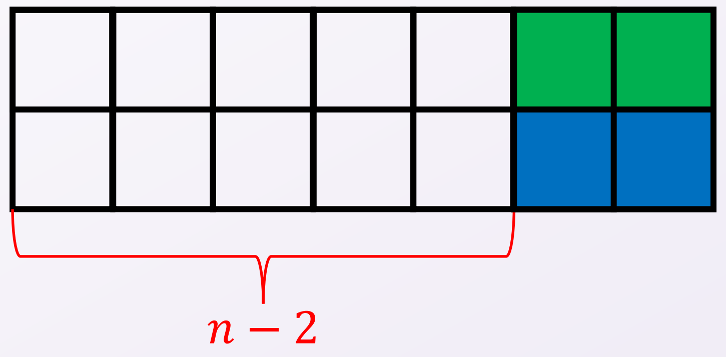
How many ways are there to tile a $2 \times n$ board with dominoes?

Two ways to fill the final column:



$$Tile(n) = Tile(n-1) + Tile(n-2)$$

$$Tile(0) = Tile(1) = 1$$

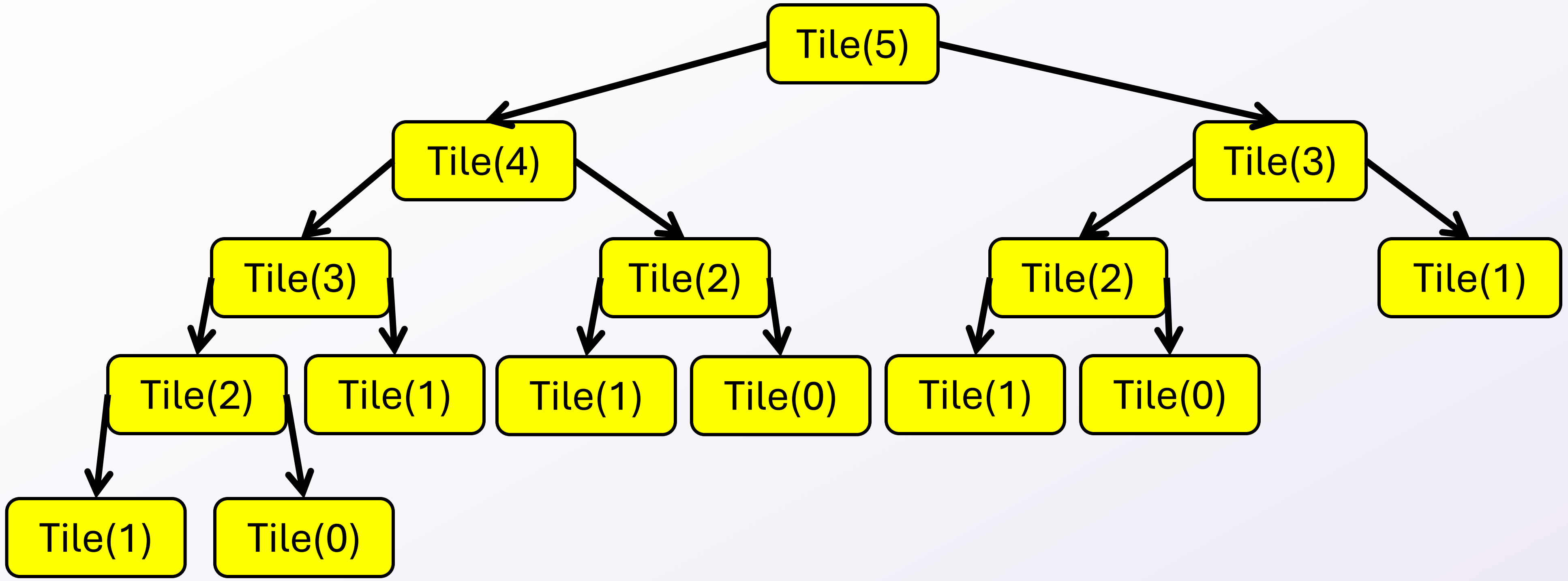


How to compute $Tile(n)$?

```
Tile(n):
    if n < 2:
        return 1
    return Tile(n-1)+Tile(n-2)
```

Running Time?

Recursion Tree

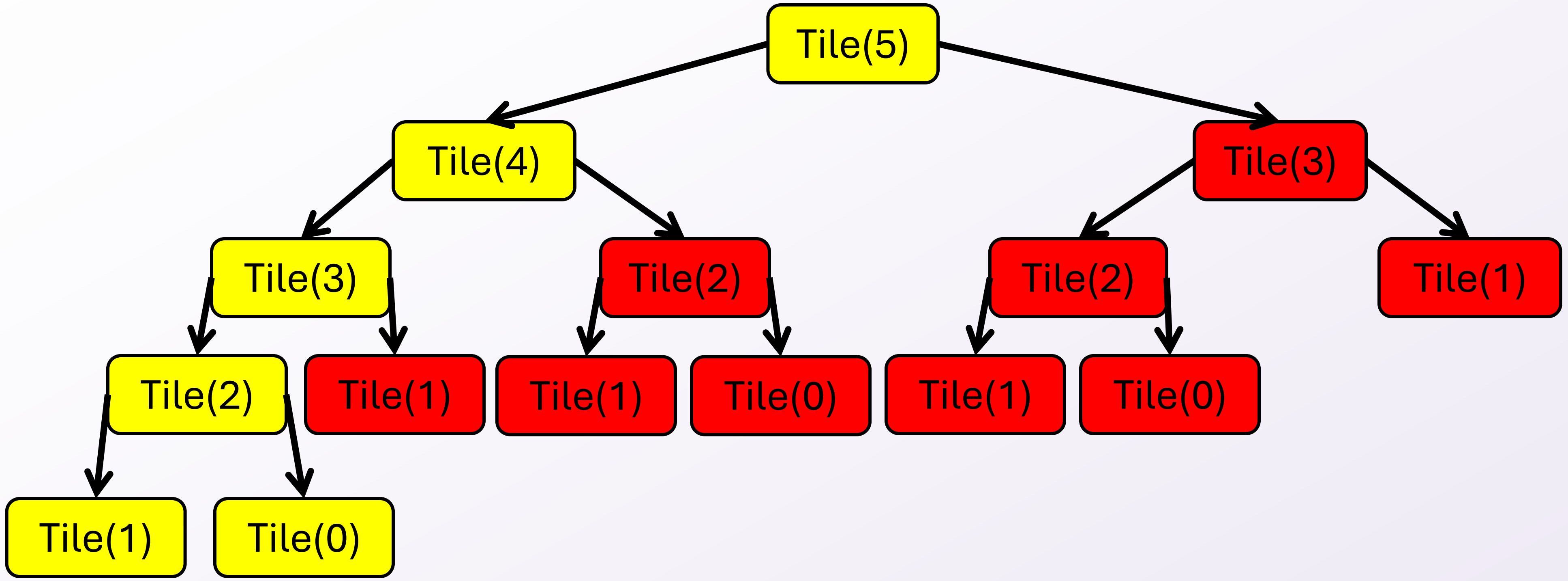


Many redundant calls!

Run time: $\Omega(2^n)$

Better way: Use Memory!

Recursion Tree Redundancy



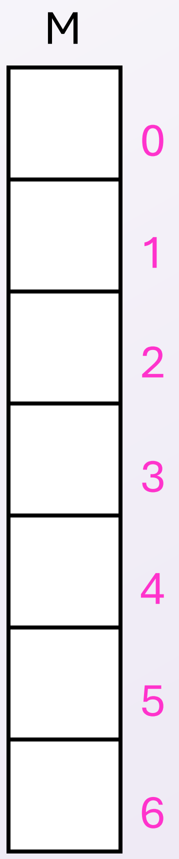
Many redundant calls!

Run time: $\Omega(2^n)$

Better way: Use Memory!

Computing $Tile(n)$ with Memory

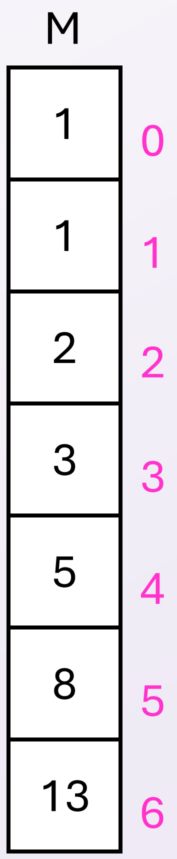
```
Initialize Memory M
Tile(n):
    if n < 2:
        return 1
    if M[n] is filled:
        return M[n]
    M[n] = Tile(n-1)+Tile(n-2)
    return M[n]
```



Technique: “memoization” (note no “r”)

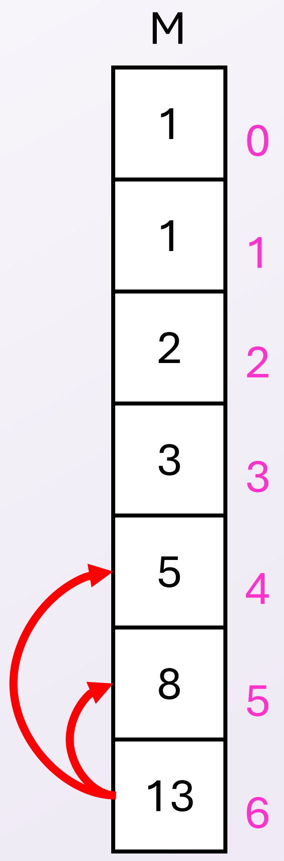
Computing $Tile(n)$ with Memory - “Top Down”

```
Initialize Memory M
Tile(n):
    if n < 2:
        return 1
    if M[n] is filled:
        return M[n]
    M[n] = Tile(n-1)+Tile(n-2)
    return M[n]
```



Computing $Tile(n)$ with Memory - “Top Down”

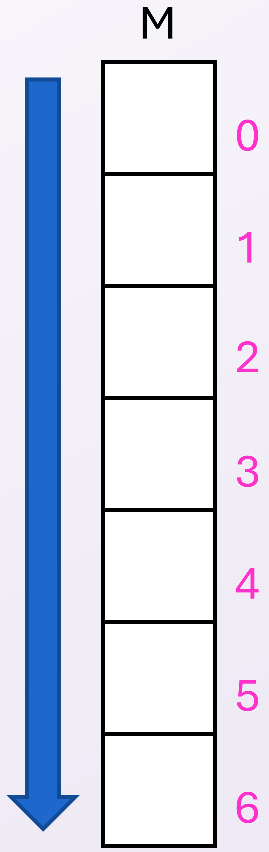
```
Initialize Memory M
Tile(n):
        if n < 2:
                return 1
        if M[n] is filled:
                return M[n]
        M[n] = Tile(n-1)+Tile(n-2)
        return M[n]
```



Recursive calls happen in a predictable order

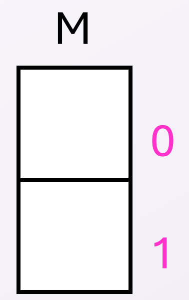
$Tile(n)$ with Memory - “Bottom Up”

```
Tile(n):
    Initialize Memory M
    M[0] = 1
    M[1] = 1
    for i = 2 to n:
        M[i] = M[i-1] + M[i-2]
    return M[n]
```



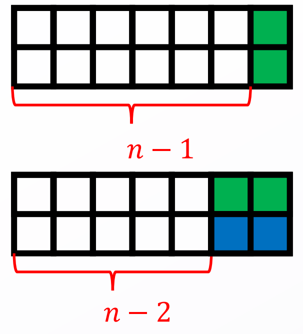
Better $Tile(n)$ with Memory - "Bottom Up"

```
Tile(n):
    M[0] = 1
    M[1] = 1
    answer = -1
    for i = 2 to n:
        answer = M[0]+M[1]
        M[0] = M[1]
        M[1] = answer
    return M[1]
```

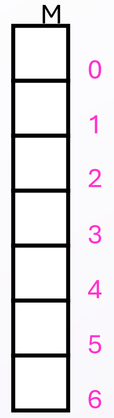


Observation: We only need to remember the last two subproblems!

Four Steps to Dynamic Programming

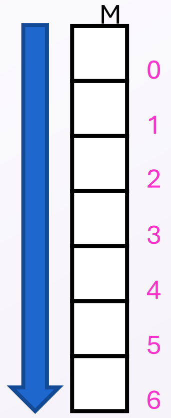


Conclusion: a 1-dimensional memory of size n

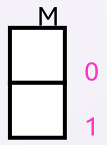


1. Formulate the answer with a recursive structure
   What are the options for the last choice?
   For each such option, what does the subproblem look like? How do we use it?

2. Choose a memory structure.
   Figure out the possible values of all parameters in the recursive calls.
   How many subproblems (options for last choice) are there?
   What are the parameters needed to identify each?
   How many different values could there be per parameter?



3. Specify an order of evaluation. (Optional)
   Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   With this step: a "Bottom-up" (iterative) algorithm
   Without this step: a "Top-down" (recursive) algorithm



4. See if there's a way to save space (Optional)
   Is it possible to reuse some memory locations?

Top-Down DP Idea

```
def myDPalgo(problem):
    if mem[problem] not blank:    // Check if we've solved this already
        return mem[problem]
    if baseCase(problem):    // Check if this is a base case
        solution = solve(problem)
        mem[problem] = solution    // Always save your solution before returning
        return solution
    for subproblem of problem:
        subsolutions.append(myDPalgo(subproblem)) // solve each subproblem
    solution = selectAndExtend(subsolutions) // Pick the subproblem to use
    mem[problem] = solution    // Always save your solution before returning
    return solution
```

Bottom-Up DP Idea

```
def myDPalgo(problem):
    for each baseCase:    // Identify which subproblems are base cases
        solution = solve(baseCase)
        mem[baseCase] = solution    // Save the solution for reuse
    for each subproblem in bottom-up order:
        // The order should be chosen so that every subsolution is
        // guaranteed to already be in memory when it's needed
        solution = selectAndExtend(subsolutions)
        mem[subproblem] = solution // Save the solution for reuse
    return mem[problem]
```

Log Cutting

Given a log of length n
A list (of length n) of prices P ($P[i]$ is the price of a cut of size i)
Find the best way to cut the log

Length:	1	2	3	4	5	6	7	8	9	10
Price:	1	5	8	9	10	17	17	20	24	30

Select a list of lengths $\ell_1, \ldots, \ell_k$ such that:
$\sum \ell_i = n$
to maximize $\sum P[\ell_i]$

Brute Force: $O(2^n)$

Greedy won't work – Idea 1

- Greedy algorithms (next unit) build a solution by picking the best option "right now"
  - Select the most profitable cut first

Price:	1	18	24	36	50	50
Length:	1	2	3	4	5	6

Greedy: Lengths: 5, 1
Profit: 51

Better: Lengths: 2, 4
Profit: 54

Greedy won't work – Idea 2

Greedy algorithms (next unit) build a solution by picking the best option "right now"

Select the "most bang for your buck"

- (best price / length ratio)

Price:	1	18	24	36	50	50
Length:	1	2	3	4	5	6

Greedy: Lengths: 5, 1
Profit: 51

Better: Lengths: 2, 4
Profit: 54

DP’s Four Steps – Log Cutting – Step 1

1. Formulate the answer with a recursive structure
   What are the options for the last choice?
   For each such option, what does the subproblem look like? How do we use it?

2. Choose a memory structure.
   Figure out the possible values of all parameters in the recursive calls.
   How many subproblems (options for last choice) are there?
   What are the parameters needed to identify each?
   How many different values could there be per parameter?

3. Specify an order of evaluation. (Optional)
   Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   With this step: a “Bottom-up” (iterative) algorithm
   Without this step: a “Top-down” (recursive) algorithm

4. See if there’s a way to save space (Optional)
   Is it possible to reuse some memory locations?

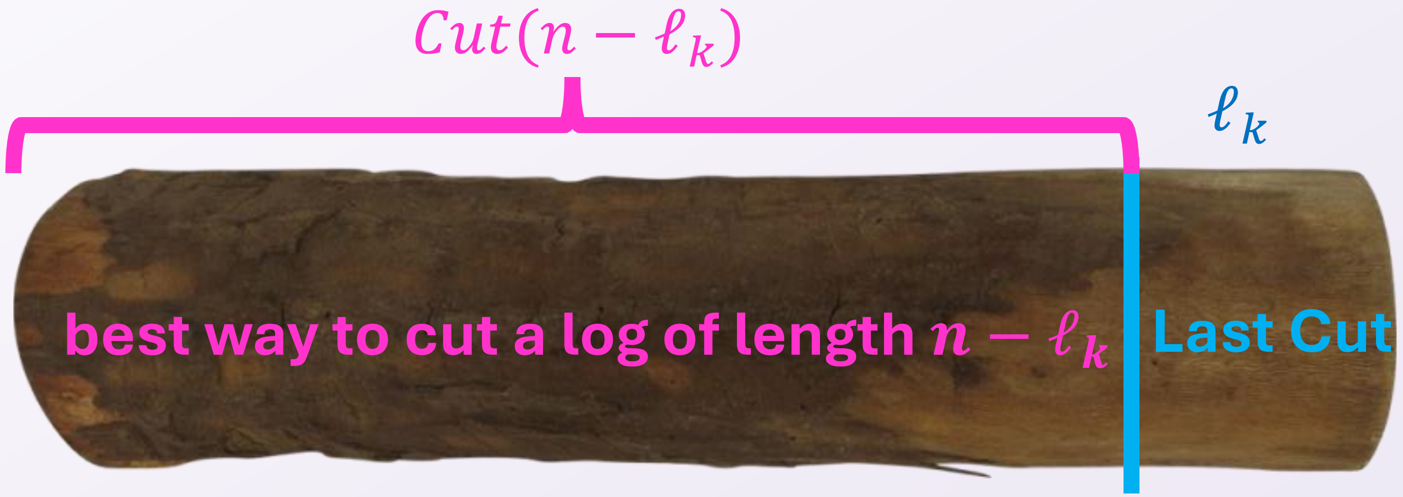
1. Identify Recursive Structure

$P[i]$ = value of a cut of length i

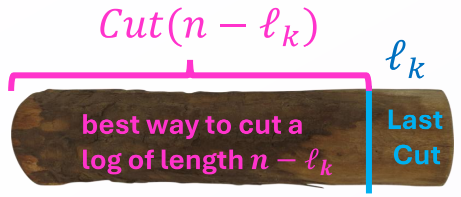
$Cut(n)$ = value of best way to cut a log of length n

$$Cut(n) = \max \begin{cases} Cut(n-1) + P[1] \\ Cut(n-2) + P[2] \\ \dots \\ Cut(0) + P[n] \end{cases}$$

Base Case:
$Cut(0) = 0$



DP's Four Steps – Log Cutting – Step 2



1. Formulate the answer with a recursive structure
   What are the options for the last choice?
   For each such option, what does the subproblem look like? How do we use it?

2. Choose a memory structure.
   Figure out the possible values of all parameters in the recursive calls.
   How many subproblems (options for last choice) are there?
   What are the parameters needed to identify each?
   How many different values could there be per parameter?

3. Specify an order of evaluation. (Optional)
   Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   With this step: a "Bottom-up" (iterative) algorithm
   Without this step: a "Top-down" (recursive) algorithm

4. See if there's a way to save space (Optional)
   Is it possible to reuse some memory locations?

2. Design You Memory

$P[i]$ = value of a cut of length i

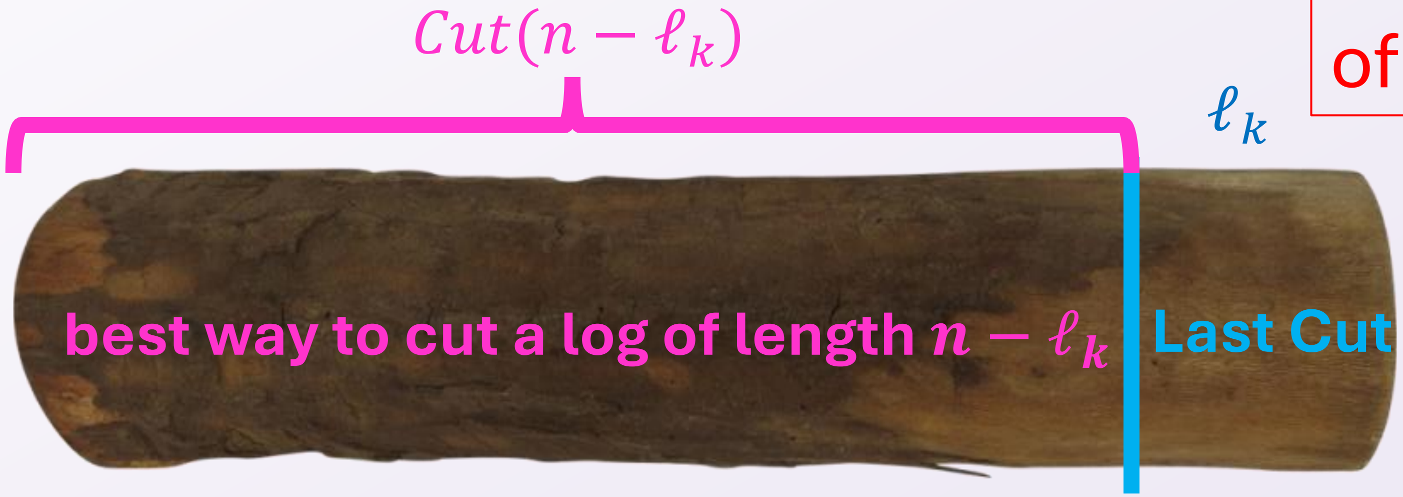
$Cut(n)$ = value of best way to cut a log of length n

$$Cut(n) = \max \begin{cases} Cut(n-1) + P[1] \\ Cut(n-2) + P[2] \\ \dots \\ Cut(0) + P[n] \end{cases}$$

Must store solutions to each $Cut(i)$

So each choice of i needs its own memory location

1-dimensional array of length n



Generic Top-Down Dynamic Programming Soln

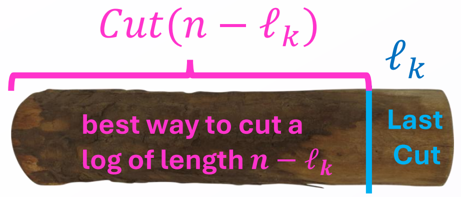
```
mem = {}
def myDPalgo(problem):
    if mem[problem] not blank:
        return mem[problem]
    if baseCase(problem):
        solution = solve(problem)
        mem[problem] = solution
        return solution
    for subproblem of problem:
        subsolutions.append(myDPalgo(subproblem))
    solution = OptimalSubstructure(subsolutions)
    mem[problem] = solution
    return solution
```

Log Cutting Top-Down

```
mem = [-1 for value in P]
def cut_log(n, P):
        if mem[n] < 0:
                return mem[n]
        if n == 0:
                solution = 0
                mem[n] = solution
                return solution
        subsolutions = []
        for i from 1 to n:
                subsolutions.append(cut_log(n-i,P)+P[i])
        solution = max(subsolutions)
        mem[n] = solution
        return solution
```

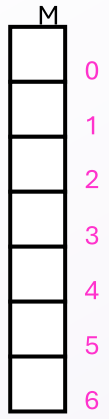
$$Cut(n) = \max \begin{cases} Cut(n-1) + P[1] \\ Cut(n-2) + P[2] \\ \ldots \\ Cut(0) + P[n] \end{cases}$$

DP's Four Steps – Log Cutting – Step 3



1-dimensional memory of size n

$M[i]$ = best value achievable for an i-foot long log



1. Formulate the answer with a recursive structure
   What are the options for the last choice?
   For each such option, what does the subproblem look like? How do we use it?

2. Choose a memory structure.
   Figure out the possible values of all parameters in the recursive calls.
   How many subproblems (options for last choice) are there?
   What are the parameters needed to identify each?
   How many different values could there be per parameter?

3. Specify an order of evaluation. (Optional)
   Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   With this step: a "Bottom-up" (iterative) algorithm
   Without this step: a "Top-down" (recursive) algorithm

4. See if there's a way to save space (Optional)
   Is it possible to reuse some memory locations?

3. Select an Order – $Cut(0)$

Solve Smallest subproblem first

$Cut(0) = 0$

$Cut(i)$:	0										
Length:	0	1	2	3	4	5	6	7	8	9	10

3. Select an Order - $Cut(1)$

Solve Smallest subproblem first

$$Cut(1) = Cut(0) + P[1]$$

$Cut(i)$:	0	🟨									
Length:	0	1	2	3	4	5	6	7	8	9	10

3. Select an Order - $Cut(2)$

Solve Smallest subproblem first

$$Cut(2) = \max \begin{cases} Cut(1) + P[1] \\ Cut(0) + P[2] \end{cases}$$

$Cut(i)$:	0										
Length:	0	1	2	3	4	5	6	7	8	9	10

3. Select an Order - $Cut(3)$

Solve Smallest subproblem first

$$Cut(3) = \max \begin{cases} Cut(2) + P[1] \\ Cut(1) + P[2] \\ Cut(0) + P[3] \end{cases}$$

$Cut(i)$:	0			🟨							
Length:	0	1	2	3	4	5	6	7	8	9	10

3

3. Select an Order - $Cut(4)$

Each subproblem uses all solutions to smaller inputs.

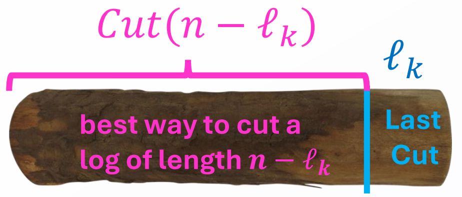
Conclusion, solve each $Cut(i)$ from smallest i to largest

Solve Smallest subproblem first

$$Cut(4) = \max \begin{cases} Cut(3) + P[1] \\ Cut(2) + P[2] \\ Cut(1) + P[3] \\ Cut(0) + P[4] \end{cases}$$

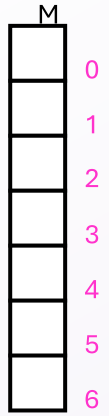
$Cut(i)$:	0				🟨						
Length:	0	1	2	3	4	5	6	7	8	9	10

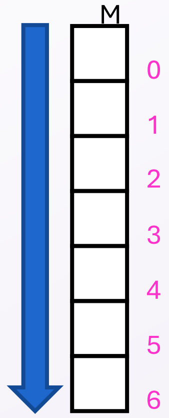
DP's Four Steps – Log Cutting – Step 4



1-dimensional memory of size n

$M[i] =$ best value achievable for an i-foot long log





1. Formulate the answer with a recursive structure
   What are the options for the last choice?
   For each such option, what does the subproblem look like? How do we use it?

2. Choose a memory structure.
   Figure out the possible values of all parameters in the recursive calls.
   How many subproblems (options for last choice) are there?
   What are the parameters needed to identify each?
   How many different values could there be per parameter?

3. Specify an order of evaluation. (Optional)
   Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   With this step: a "Bottom-up" (iterative) algorithm
   Without this step: a "Top-down" (recursive) algorithm

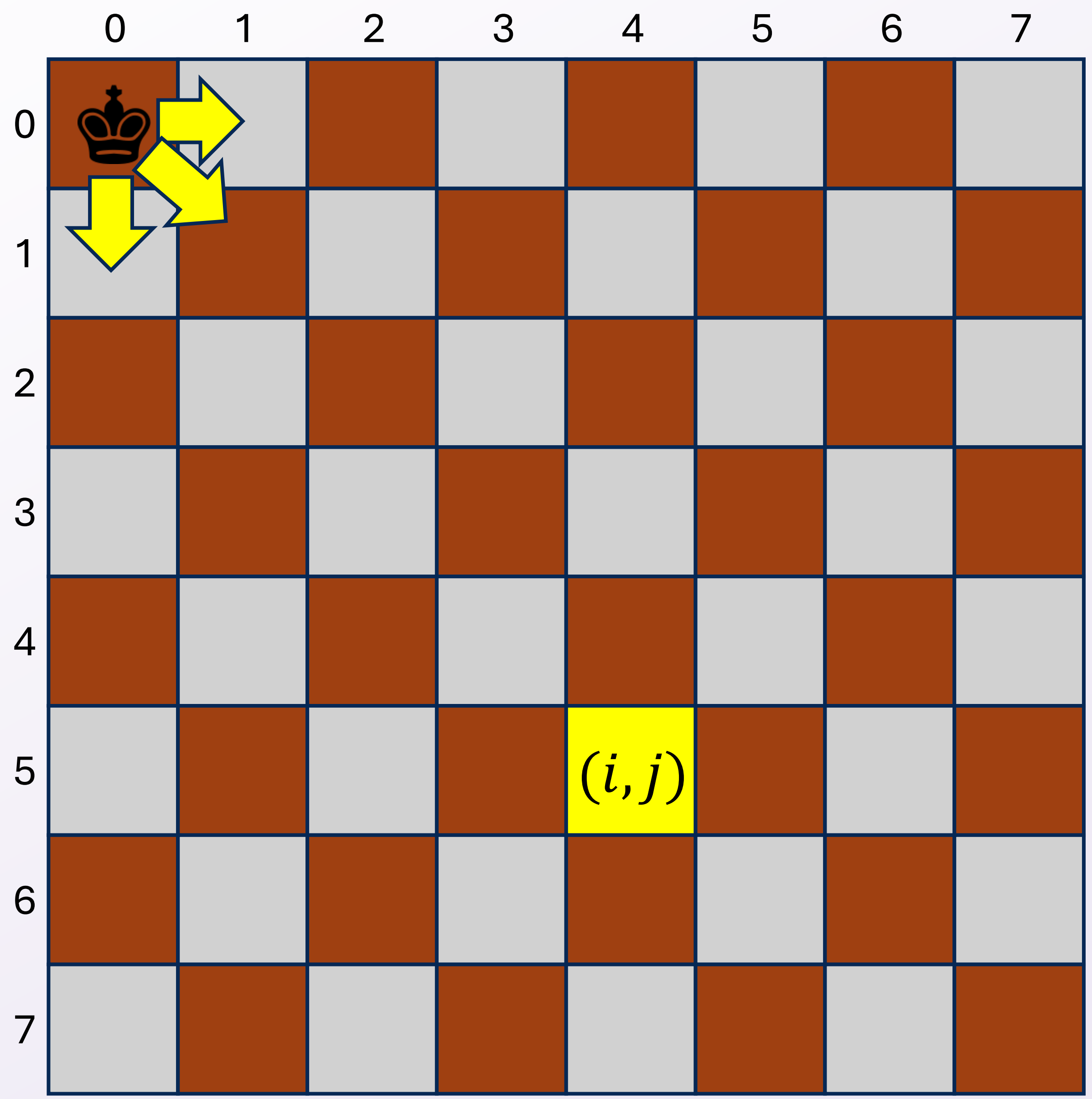
4. See if there's a way to save space (Optional)
   Is it possible to reuse some memory locations?

King Paths

Given an $n \times n$ grid and a cell (i, j), how many different direct paths are there for a king to travel from cell $(0,0)$ to (i, j)

We say a path is direct provided we only move in the positive direction in rows and columns

A king can move to any spot to its left, bottom, or bottom-left.



DP's Four Steps – King Paths – Step 1

1. Formulate the answer with a recursive structure

   What are the options for the last choice?

   For each such option, what does the subproblem look like? How do we use it?

2. Choose a memory structure.

   Figure out the possible values of all parameters in the recursive calls.

   How many subproblems (options for last choice) are there?

   What are the parameters needed to identify each?

   How many different values could there be per parameter?

3. Specify an order of evaluation. (Optional)

   Want to guarantee that the necessary subproblem solutions are in memory when you need them.

   With this step: a "Bottom-up" (iterative) algorithm

   Without this step: a "Top-down" (recursive) algorithm

4. See if there's a way to save space (Optional)

   Is it possible to reuse some memory locations?

King Paths – last choices

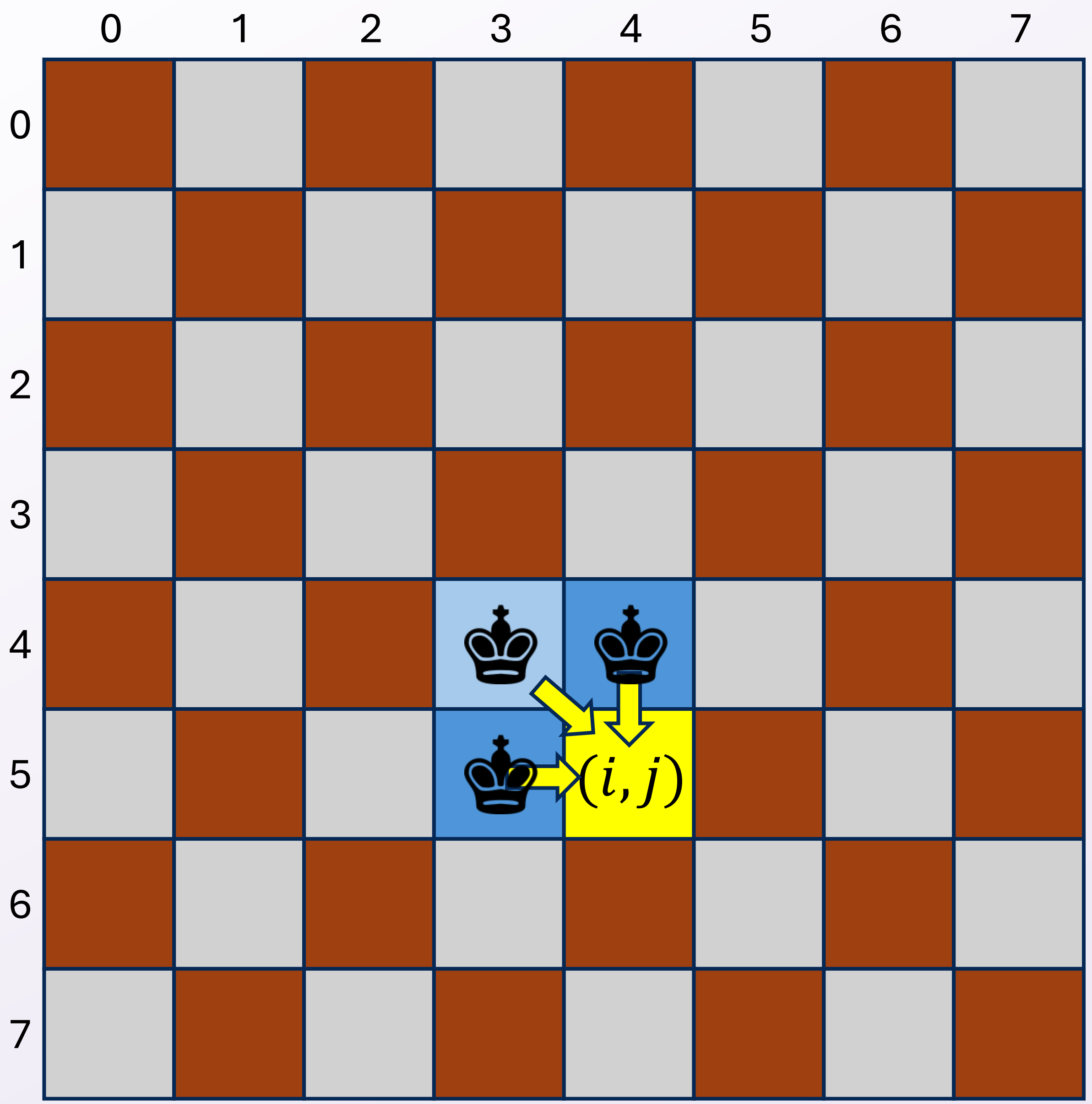
Before reaching (i, j), the king must have been in $(i-1, j)$, $(i, j-1)$, or $(i-1, j-1)$

Therefore:

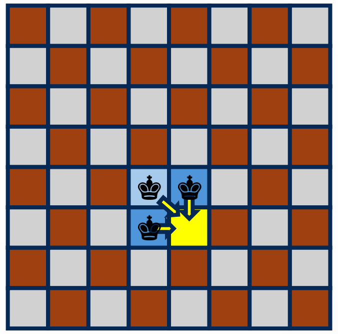
$$\begin{aligned} pathcount(i,j) = {} & pathcount(i-1,j) + \\ & pathcount(i,j-1) + \\ & pathcount(i-1,j-1) \end{aligned}$$

Base Cases:

$$pathcount(0,0) = 1$$
$$pathcount(0,1) = 1$$
$$pathcount(1,0) = 1$$



DP’s Four Steps – King Paths – Step 2



1. Formulate the answer with a recursive structure
   What are the options for the last choice?
   For each such option, what does the subproblem look like? How do we use it?

2. Choose a memory structure.
   Figure out the possible values of all parameters in the recursive calls.
   How many subproblems (options for last choice) are there?
   What are the parameters needed to identify each?
   How many different values could there be per parameter?

3. Specify an order of evaluation. (Optional)
   Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   With this step: a “Bottom-up” (iterative) algorithm
   Without this step: a “Top-down” (recursive) algorithm

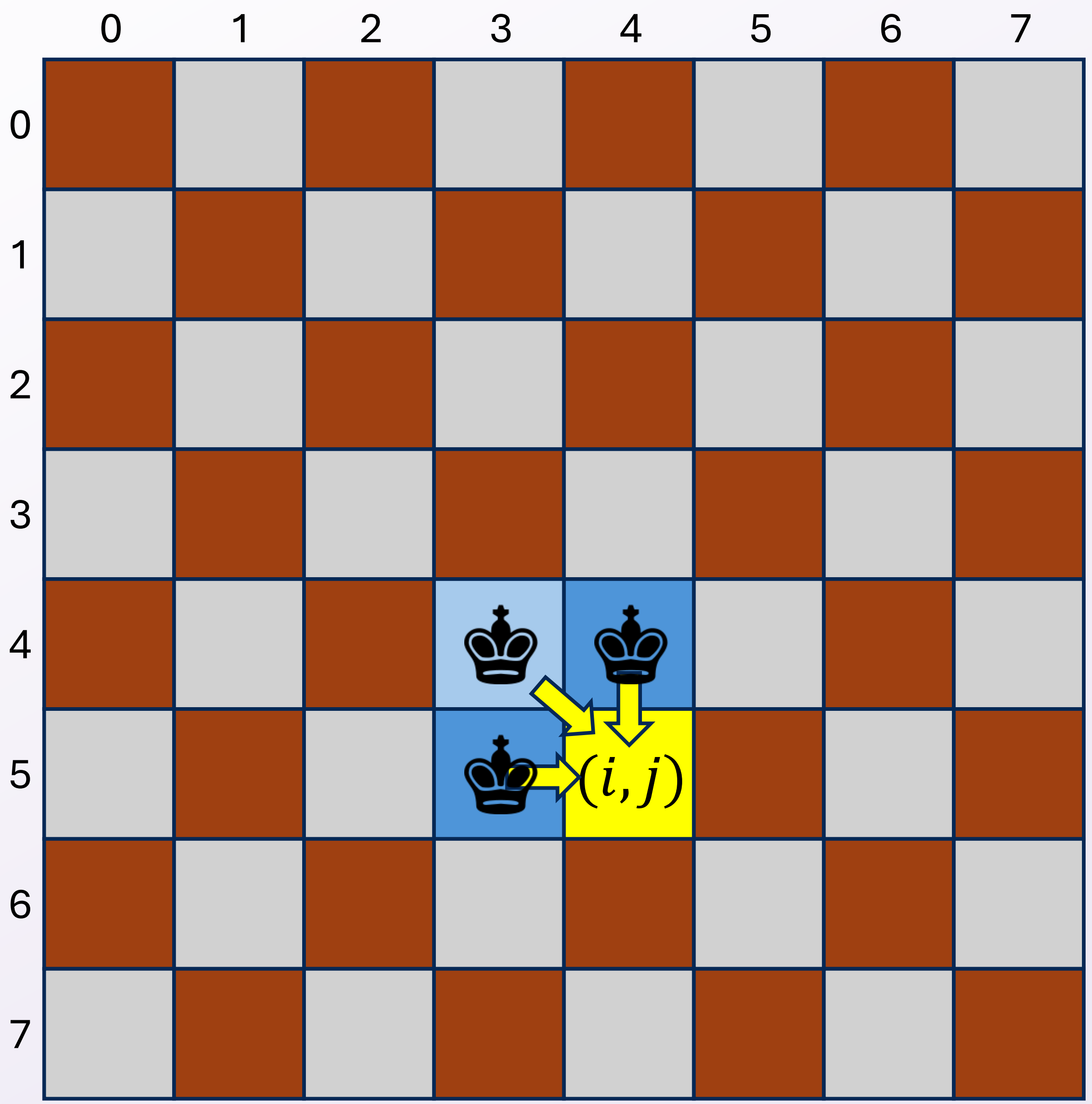
4. See if there’s a way to save space (Optional)
   Is it possible to reuse some memory locations?

King Paths – Memory

$$\begin{aligned} pathcount(i,j) = {} & pathcount(i-1,j) + \\ & pathcount(i,j-1) + \\ & pathcount(i-1,j-1) \end{aligned}$$

Each location (x, y) may have its own subproblem

Need a 2-d memory whose size is $i \times j$

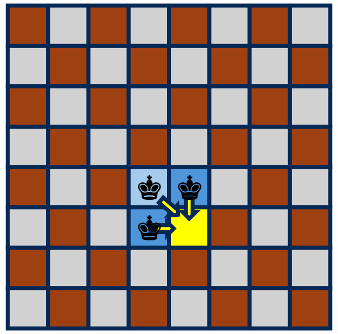


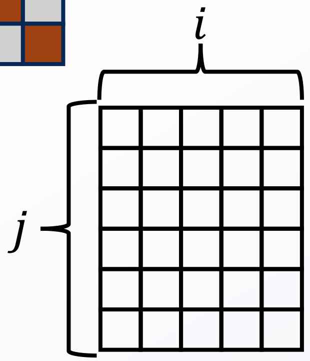
King Paths Top-Down

```
mem = $i \times j$ array full of $-1$
def pathcount(i,j):
    if mem[i][j] > 0:
        return mem[i][j]
    if i == 0 or i == 0:
        solution = 1
        mem[i][j] = solution
        return solution
    subsolutions = []
    upper = pathcount(i-1,j)
    left = pathcount(i,j-1)
    upperleft = pathcount(i-1,j-1)
    solution = upper + left + upperleft
    mem[i][j] = solution
    return solution
```

$$\begin{aligned} pathcount(i,j) = {} & pathcount(i-1,j) + \\ & pathcount(i,j-1) + \\ & pathcount(i-1,j-1) \end{aligned}$$

DP's Four Steps – King Paths – Step 3





1. Formulate the answer with a recursive structure
   What are the options for the last choice?
   For each such option, what does the subproblem look like? How do we use it?

2. Choose a memory structure.
   Figure out the possible values of all parameters in the recursive calls.
   How many subproblems (options for last choice) are there?
   What are the parameters needed to identify each?
   How many different values could there be per parameter?

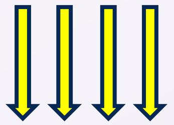
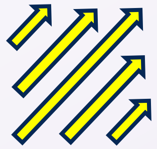
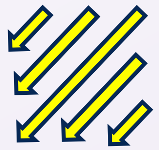
3. Specify an order of evaluation. (Optional)
   Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   With this step: a "Bottom-up" (iterative) algorithm
   Without this step: a "Top-down" (recursive) algorithm

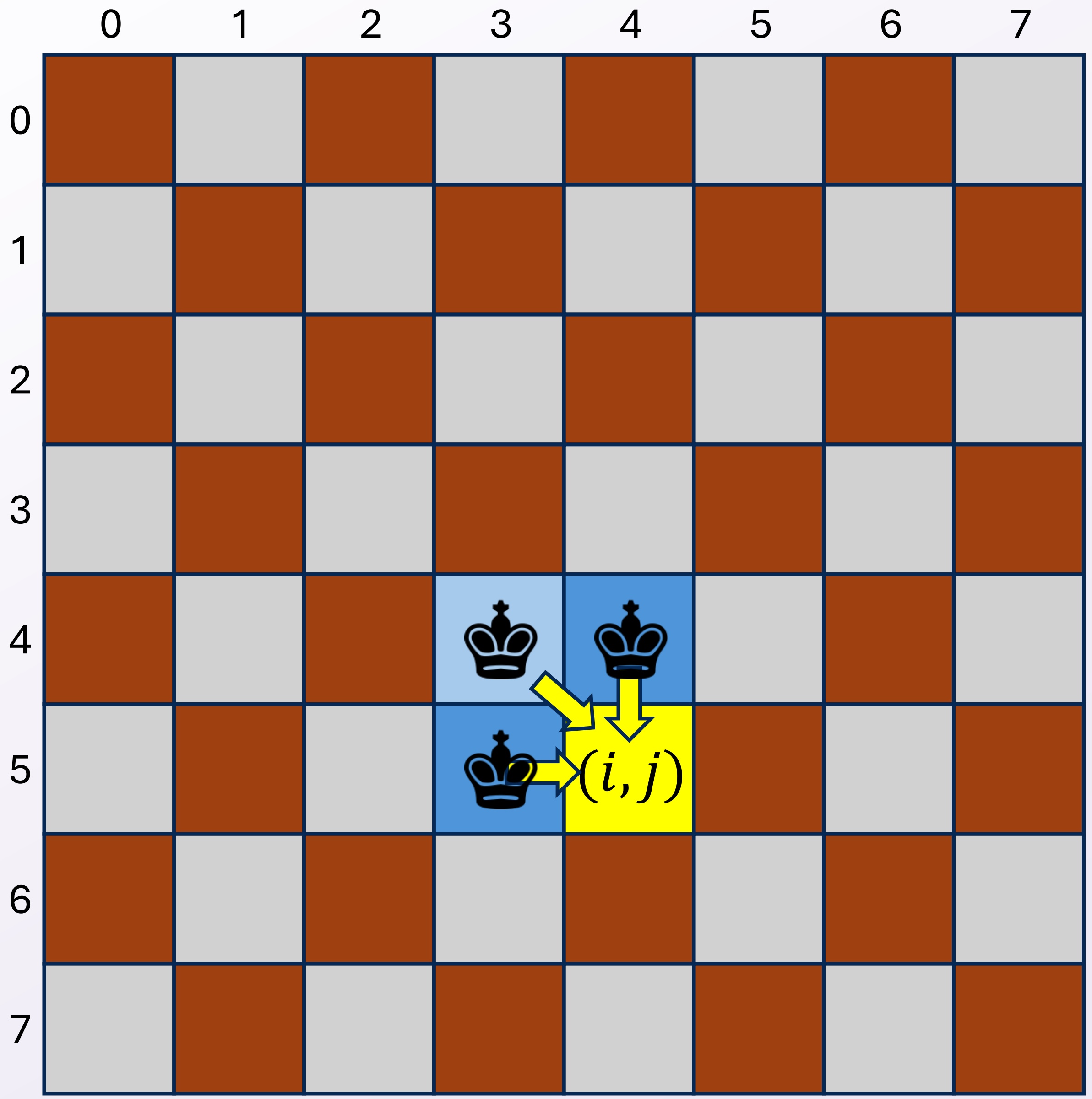
4. See if there's a way to save space (Optional)
   Is it possible to reuse some memory locations?

King Paths – Order

$$pathcount(i,j) = pathcount(i-1,j) + pathcount(i,j-1) + pathcount(i-1,j-1)$$

Each location (x, y) depends on:
$(x-1, y), (x, y-1), (x-1, y-1)$

So any of the following would work:
- Left to right, top to bottom
- Top to bottom, left to right 
- diagonally  or 



King Paths Bottom-up

$$pathcount(i,j) = pathcount(i-1,j) + pathcount(i,j-1) + pathcount(i-1,j-1)$$

```
mem = $i \times j$ array full of $-1$
def **pathcount**(i,j):
        for(int row = 0; row $\leq$ i; i++):
                mem[row][0]=1
        for(int col = 0; col $\leq$ j; j++):
                mem[0][col]=1
        for(int row = 1; row $\leq$ i; i++):
                for(int col = 1; col $\leq$ j; j++):
                        mem[row][col]= mem[row-1][col] +
                                       mem[row][col-1] +
                                       mem[row-1][col-1]
        return mem[i][j]
```

Final reminders

HW3 resubmissions due Wednesday @ 11:59pm.

HW4 due Wednesday @ 11:59pm.

I have OH now-12:30pm:

- Meet at front of classroom, we'll walk over together
- CSE (Allen) 434 if you're coming later

Glenn has online OH 12–1pm:

- https://washington.zoom.us/my/nathanbrunelle