CS 4102: Algorithms

Lecture 11: Dynamic Programming

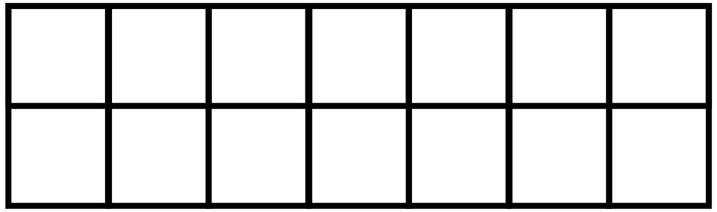
David Wu

Fall 2019

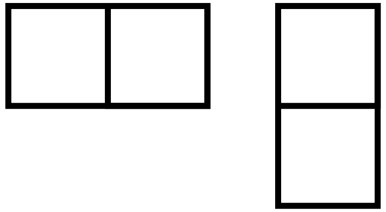
Warm Up

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

How many ways to tile a 2×7 board

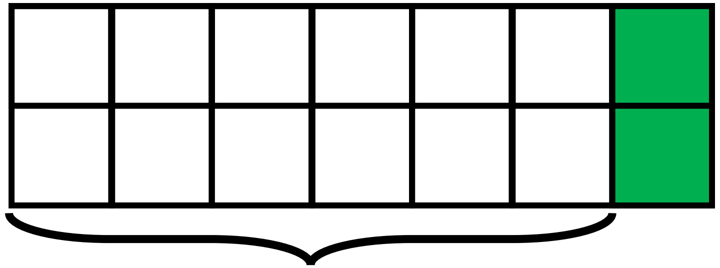


With these?

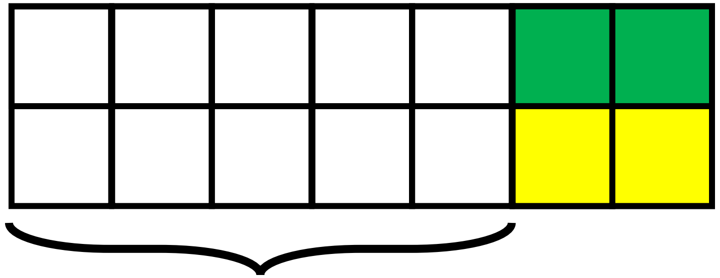


Tiling Dominoes

Two ways to fill the final column:



$n - 1$



$n - 2$

$$\text{Tile}(n) = \text{Tile}(n-1) + \text{Tile}(n-2)$$

$$\text{Tile}(0) = \text{Tile}(1) = 1$$

1	1	2	3	5	8	13	21
0	1	2	3	4	5	6	7

Today’s Keywords

Dynamic programming

Maximum sum contiguous subarray

Tiling dominoes

Log cutting

Matrix chaining

CLRS Readings: Chapter 14

Homework

- **HW3** due ~~**Tuesday, October 1, 11pm**~~ **Wednesday, October 2, 11pm**
  - Divide and conquer algorithms
  - Written (use LaTeX!) – Submit both **zip** and **pdf**!

- **Regrade office hours:**
  - Thursday 11am-12pm (Rice 210)
  - Thursday 4pm-5pm (Rice 501)

Maximum Sum Subarray Problem

5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Maximum sum contiguous subarray (MSCS) problem:

find the largest contiguous subarray that maximizes the sum of the values

Maximum Sum Subarray Problem

5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Maximum sum contiguous subarray (MSCS) problem:

find the largest contiguous subarray that maximizes the sum of the values

Divide and Conquer $\Theta(n \log n)$

5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Recursively solve on left

19

Divide in half

Recursively solve on right

25

Divide and Conquer $\Theta(n \log n)$

Largest sum that ends here + Largest sum that starts here

6	1	-7	-3	-6	-13	2	8	-12	5	13	-37	-42	-20
5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Recursively solve on left

19

Divide in half

Recursively solve on right

25

Combine: Find largest sum that spans the cut

Divide and Conquer $\Theta(n \log n)$

Largest sum that ends here + Largest sum that starts here

Sums	**6**	1	-7	-3	-6	-13	2	8	-12	5	**13**	-37	-42	-20
Values	5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Recursively solve on left

19

Divide in half

Recursively solve on right

25

Combine: Find largest sum that spans the cut

19

$T(n) = 2T(n/2) + \Theta(n) \in \Theta(n \log n)$

Unbalanced Divide and Conquer

Divide

- Make a subproblem of all but the last element

5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Unbalanced Divide and Conquer

Divide

- Make a subproblem of all but the last element

Conquer

- Find best subarray on the left ($BSL(n-1)$)
- Find the best subarray ending at the divide ($BED(n-1)$)

5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Best subarray ending at the divide is empty

Unbalanced Divide and Conquer

Divide

- Make a subproblem of all but the last element

Conquer

- Find best subarray on the left ($BSL(n-1)$)
- Find the best subarray ending at the divide ($BED(n-1)$)

5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Combine

- Find the best subarray that “spans the divide” and output best among all candidates

Unbalanced Divide and Conquer

Best subarray that spans divide must include last element: $BED(n)$

- $BED(n) = \max(BED(n-1) + arr[n], 0)$

Best subarray must either include or exclude the last element

- $BSL(n) = \max\big(BSL(n-1), BED(n)\big)$

5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Combine

- Find the best subarray that "spans the divide" and output best among all candidates

Unbalanced Divide and Conquer

Divide

- Make a subproblem of all but the last element

Conquer

- Find best subarray on the left ($BSL(n-1)$)
- Find the best subarray ending at the divide ($BED(n-1)$)

Combine

- New best subarray ending at the divide:
  - $BED(n) = \max(BED(n-1) + arr[n], 0)$
- New best on the left:
  - $BSL(n) = \max\big(BSL(n-1), BED(n)\big)$

If we compute $BED(n-1)$ and $BSL(n-1)$, then Combine is constant-time!

Unbalanced Divide and Conquer

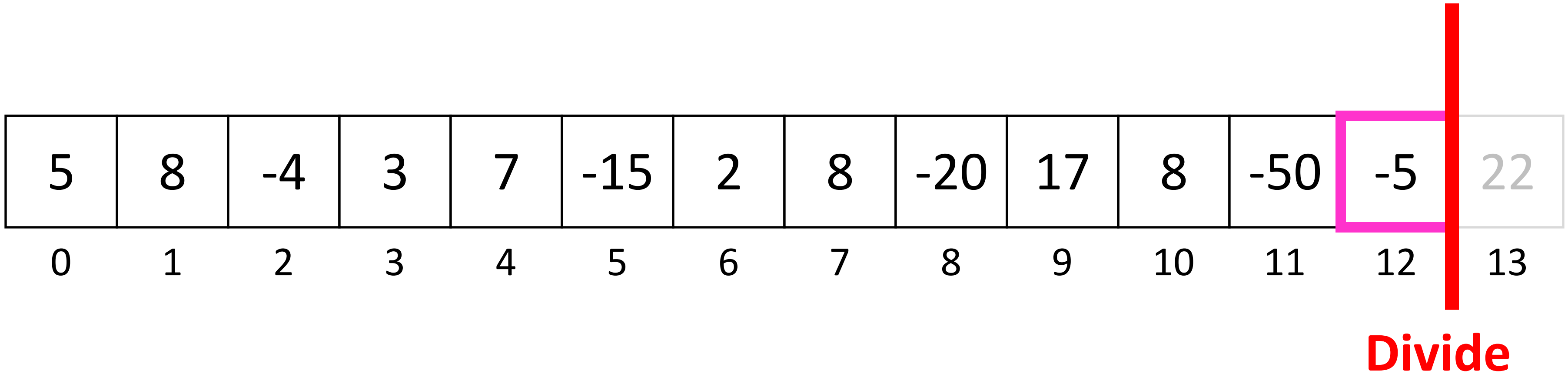
5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Divide

$$BED(n) = \max(BED(n-1) + arr[n], 0)$$

$$BSL(n) = \max\big(BSL(n-1), BED(n)\big)$$

Unbalanced Divide and Conquer



$$BED(n) = \max(BED(n-1) + arr[n], 0)$$

$$BSL(n) = \max\big(BSL(n-1), BED(n)\big)$$

Unbalanced Divide and Conquer

5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Divide

$$BED(n) = \max(BED(n-1) + arr[n], 0)$$

$$BSL(n) = \max\big(BSL(n-1), BED(n)\big)$$

Unbalanced Divide and Conquer

5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Divide

$$BED(n) = \max(BED(n-1) + arr[n], 0)$$

$$BSL(n) = \max\big(BSL(n-1), BED(n)\big)$$

Unbalanced Divide and Conquer

5

5

5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Divide

Find largest sum ending at the cut

5

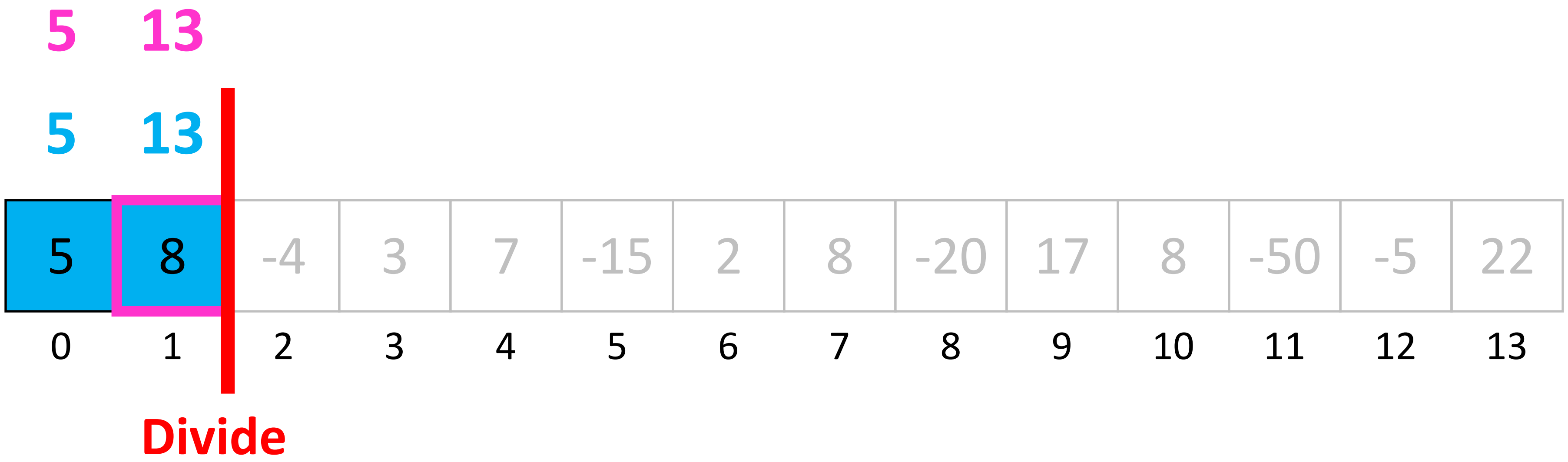
Recursively solve on left

5

$$BED(n) = \max(BED(n-1) + arr[n], 0)$$

$$BSL(n) = \max\big(BSL(n-1), BED(n)\big)$$

Unbalanced Divide and Conquer



Find largest sum ending at the cut

13

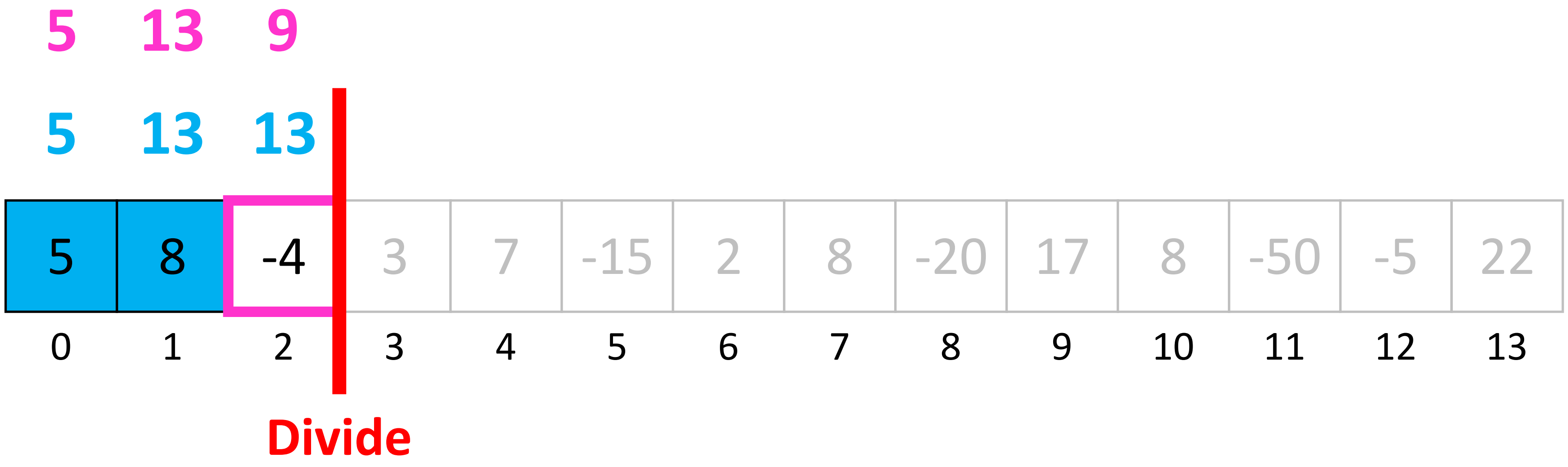
Recursively solve on left

13

$$BED(n) = \max(BED(n-1) + arr[n], 0)$$

$$BSL(n) = \max\big(BSL(n-1), BED(n)\big)$$

Unbalanced Divide and Conquer



Find largest sum ending at the cut

9

Recursively solve on left

13

$$BED(n) = \max(BED(n-1) + arr[n], 0)$$

$$BSL(n) = \max\big(BSL(n-1), BED(n)\big)$$

Unbalanced Divide and Conquer

5	13	9	12	19	4	6	14	0	17	25	0	0	22
5	13	13	13	19	19	19	19	19	19	25	25	25	25
5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Divide

Find largest sum ending at the cut

22

Recursively solve on left

25

$$T(n) = T(n-1) + \Theta(1) \in \Theta(n)$$

$$BED(n) = \max(BED(n-1) + arr[n], 0)$$

$$BSL(n) = \max\big(BSL(n-1), BED(n)\big)$$

Was Unbalanced Better?

Old:

$$T(n) = 2T(n/2) + \Theta(n) \in \Theta(n \log n)$$

- We split into 2 problems of size $n/2$
- Linear time combine (to find arrays that span the cut)

New:

$$T(n) = T(n-1) + T(1) + \Theta(1) \in \Theta(n)$$

- We split into 2 problems of size $n-1$ and 1
- Constant time combine

$\Theta(1)$

Another Look at the Recursion

5

5

5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Divide

Find largest sum ending at the cut

5

Recursively solve on left

5

$$BED(n) = \max(BED(n-1) + arr[n], 0)$$

$$BSL(n) = \max\big(BSL(n-1), BED(n)\big)$$

Another Look at the Recursion

5 13

5 13

5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Divide

Find largest sum ending at the cut

13

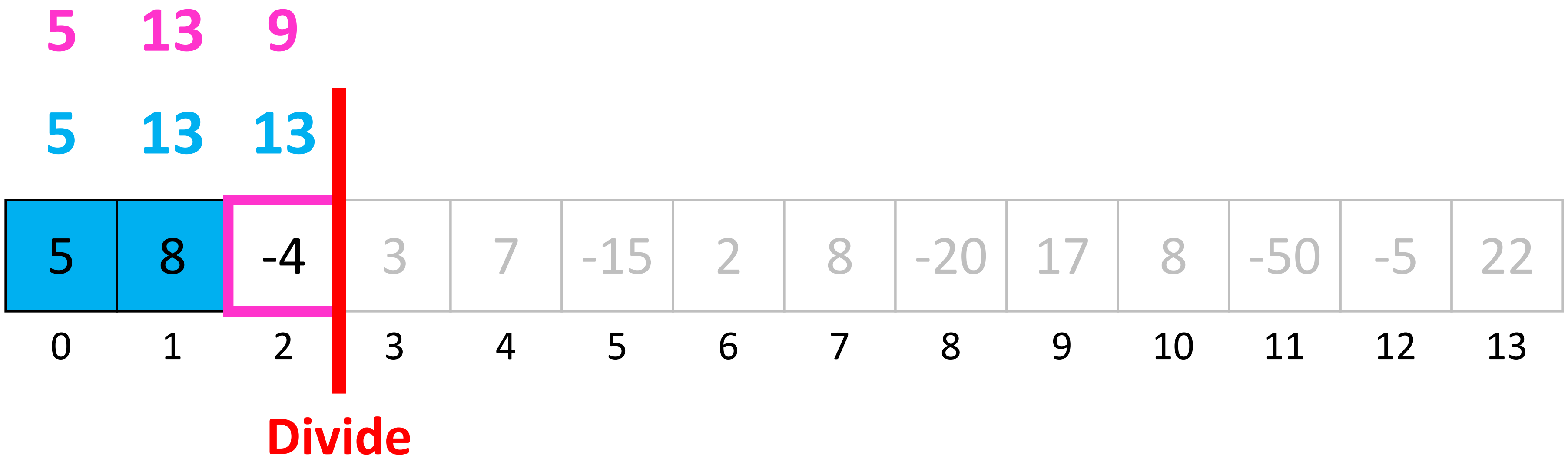
Recursively solve on left

13

$$BED(n) = \max(BED(n-1) + arr[n], 0)$$

$$BSL(n) = \max\big(BSL(n-1), BED(n)\big)$$

Another Look at the Recursion



Find largest sum ending at the cut

9

Recursively solve on left

13

$$BED(n) = \max(BED(n-1) + arr[n], 0)$$

$$BSL(n) = \max\big(BSL(n-1), BED(n)\big)$$

Another Look at the Recursion

5	**13**	**9**	**12**	**19**	**4**	**6**	**14**	**0**	**17**	**25**	**0**	**0**	**22**
5	**13**	**13**	**13**	**19**	**19**	**19**	**19**	**19**	**19**	**25**	**25**	**25**	**25**
5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Divide

Observation: No need to recurse! Just maintain two numbers and iterate from 1 to n: best value so far, best value ending at current position

$$BED(n) = \max(BED(n-1) + arr[n], 0)$$
$$BSL(n) = \max\left(BSL(n-1), BED(n)\right)$$

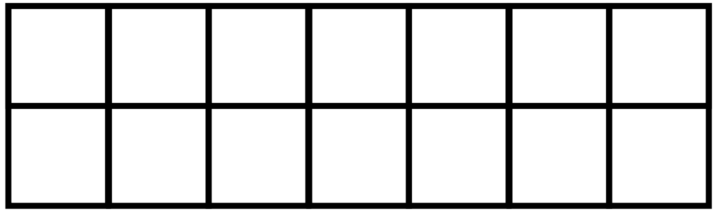
End of Midterm Exam Materials!

"Mr. Osborne, may I be excused? My brain is full."

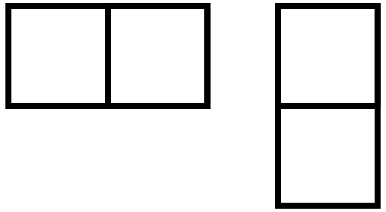
Tiling Dominoes

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

How many ways to tile a 2×7 board

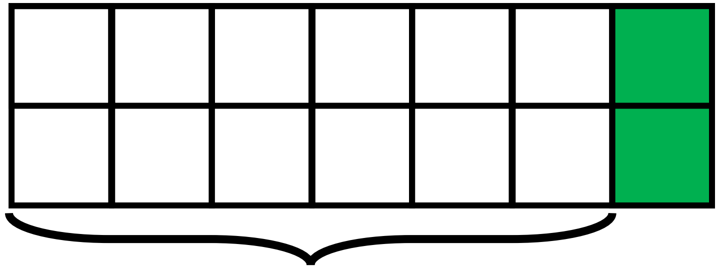


With these?



Tiling Dominoes

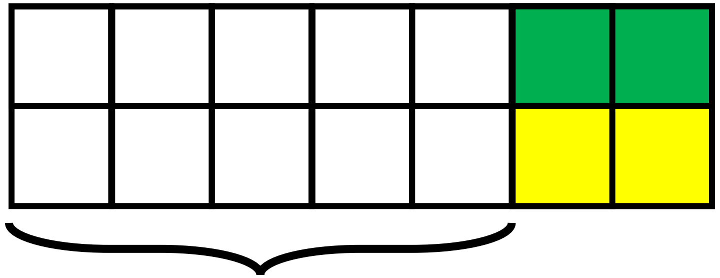
Two ways to fill the final column:



$n-1$

$$\text{Tile}(n) = \text{Tile}(n-1) + \text{Tile}(n-2)$$

$$\text{Tile}(0) = \text{Tile}(1) = 1$$



$n-2$

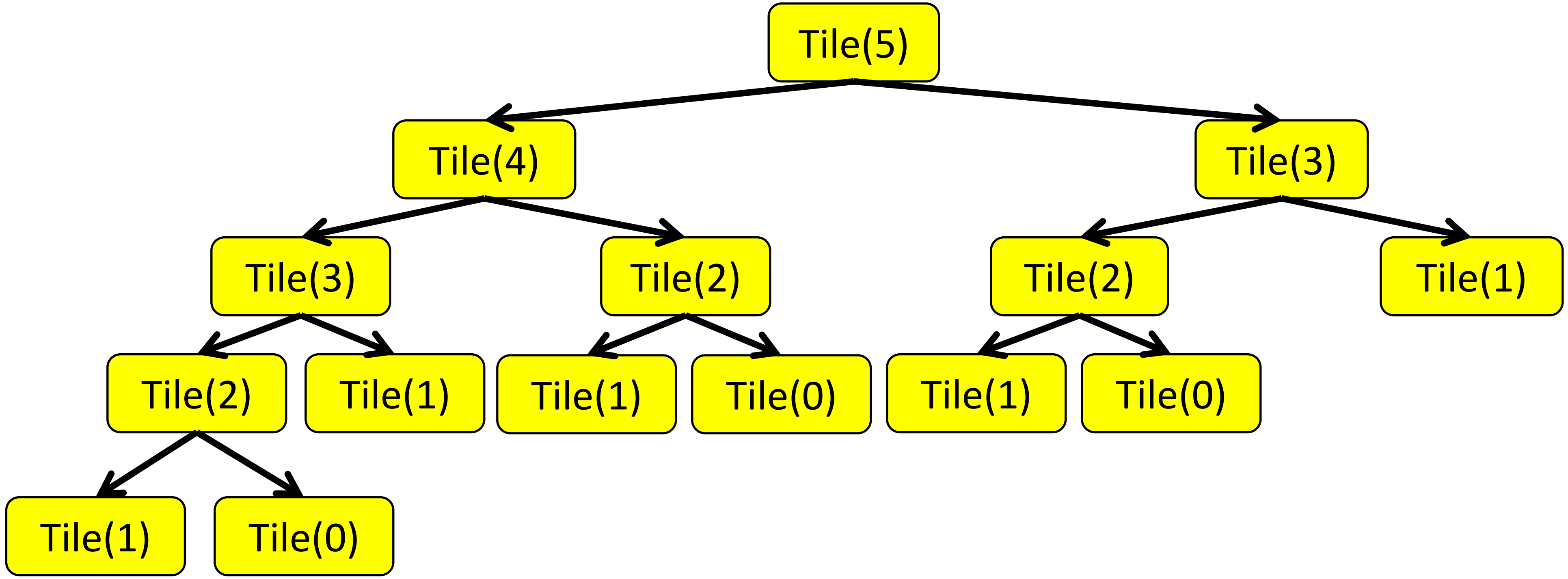
1	1	2	3	5	8	13	21
0	1	2	3	4	5	6	7

How to compute $\text{Tile}(n)$?

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

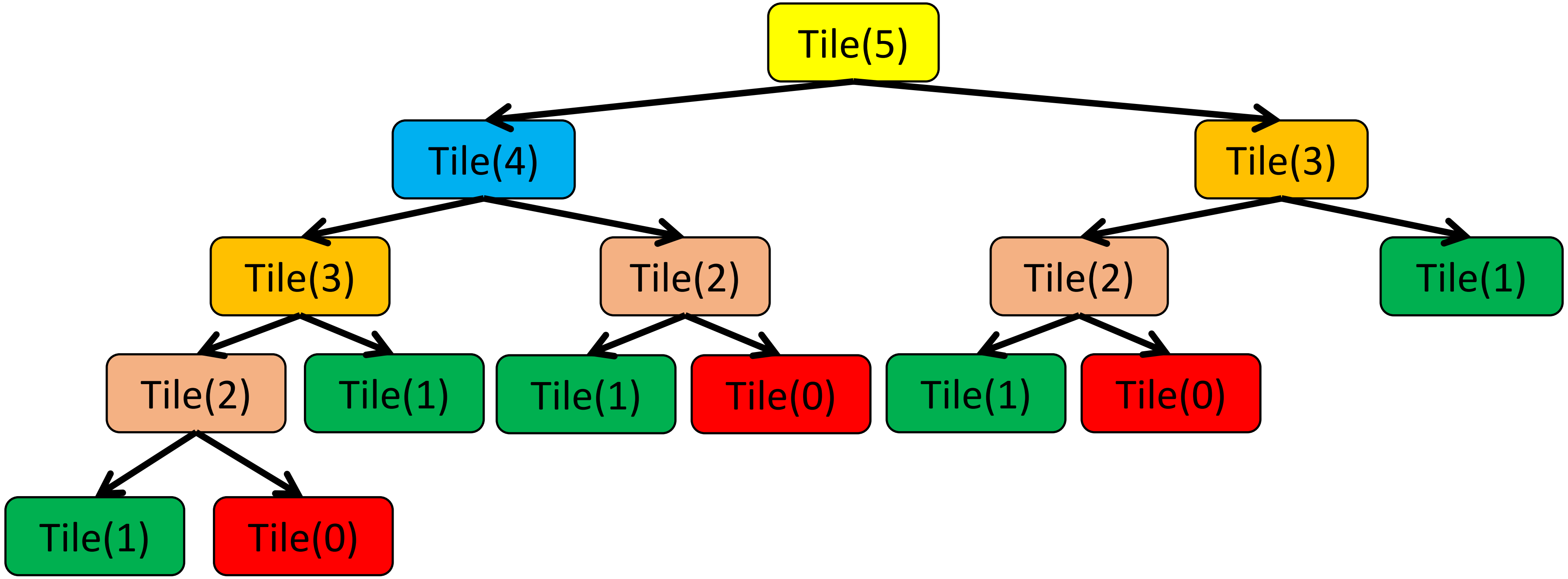
Problem?

Recursion Tree



Runtime: $\Omega(2^n)$

Recursion Tree



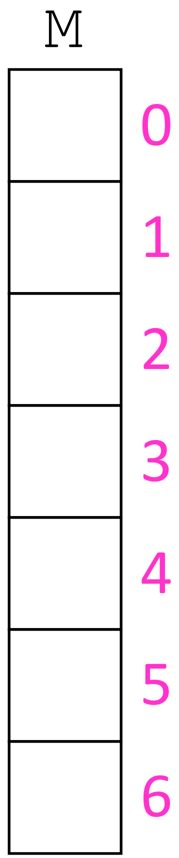
Runtime: $\Omega(2^n)$

But lots of redundant calls…

We only computed n distinct values

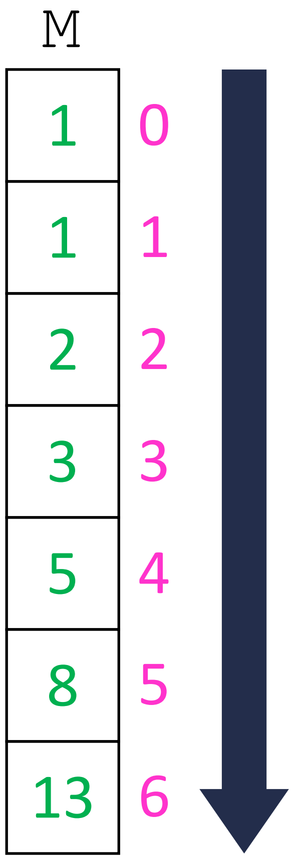
Computing $\mathrm{Tile}(n)$ with Memory ("Top Down")

```
initialize array M of size n
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 $\text{Tile}(n)$ with Memory ("Top Down")

```
initialize array M of size n
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]
```



Bottom-Up:
Fill in entries from small instances to large instances

Runtime: $\Theta(n)$

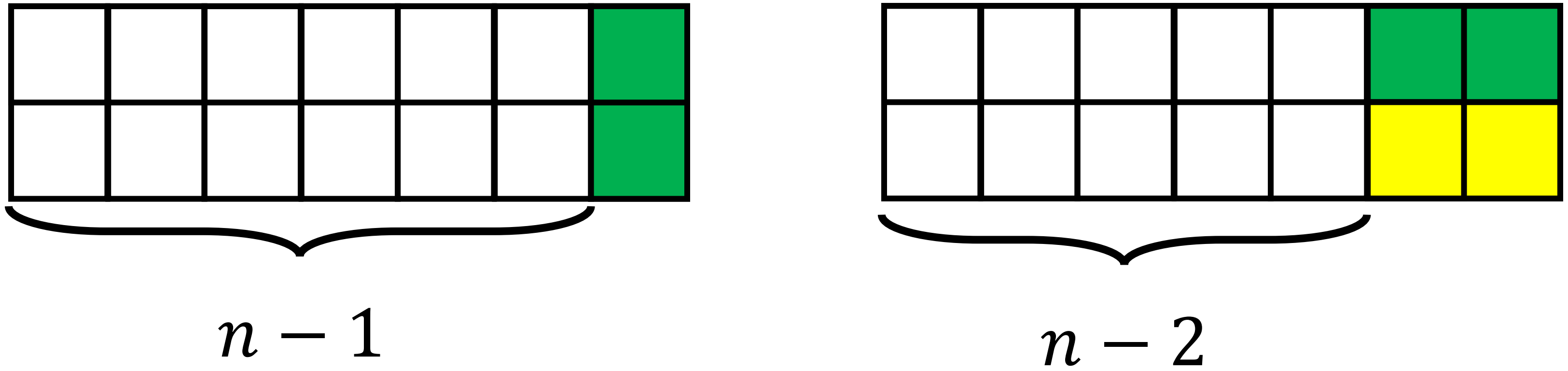
Bottom-Up: Can also iterate through M and fill in entries sequentially

Dynamic Programming

Requires optimal substructure

- Solution to larger problem contains the solutions to smaller ones ("overlapping subproblems")

General idea: Identify recursive structure of the problem and express solution to larger instances in terms of solutions to smaller instances



Generic Divide and Conquer

```
def myDCalgo(problem):


    if baseCase(problem):
        solution = solve(problem)

        return solution
    for subproblem of problem:   # After dividing
        subsolutions.append(myDCalgo(subproblem))
    solution = Combine(subsolutions)

    return solution
```

Generic Top-Down Dynamic Programming

```
mem = {}
def myDPalgo(problem):
    if mem[problem] not empty:
        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
```

Also called "memoization"

Dynamic Programming

Requires optimal substructure

- Solution to larger problem contains the solutions to smaller ones

General Blueprint:

1. Identify recursive structure of the problem
   - What is the “last thing” done?
2. Select a good order for solving subproblems
   - “Top Down:” Solve each problem recursively
   - “Bottom Up:” Iteratively solve each problem from smallest to largest

Log Cutting

Given: a log of length n, a list (of length n) of prices P

Problem: Find the best way to cut the log

$P[i]$ is the price of a cut of size i

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

Problem formulation: Find lengths $\ell_1, \dots, \ell_k$ that maximizes $\sum_{i \in [k]} P[\ell_i]$ and such that $\sum_{i \in [k]} \ell_i = n$

Brute Force: $O(2^n)$

A “Greedy” Approach

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

- **Possible strategy:** choose 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

A "Greedy" Approach

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

- **Possible strategy:** select the "most bang for your buck" (best price/length ratio)

Ratio:	1	9	8	9	10	8.3
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 solution is suboptimal

Dynamic Programming

Requires optimal substructure

- Solution to larger problem contains the solutions to smaller ones

General Blueprint:

1. Identify recursive structure of the problem
   - What is the “last thing” done?
2. Select a good order for solving subproblems
   - “Top Down:” Solve each problem recursively
   - “Bottom Up:” Iteratively solve each problem from smallest to largest

Dynamic Programming

Requires optimal substructure

- Solution to larger problem contains the solutions to smaller ones

General Blueprint:

1. Identify recursive structure of the problem
   - What is the "last thing" done?
2. Select a good order for solving subproblems
   - "Top Down:" Solve each problem recursively
   - "Bottom Up:" Iteratively solve each problem from smallest to largest

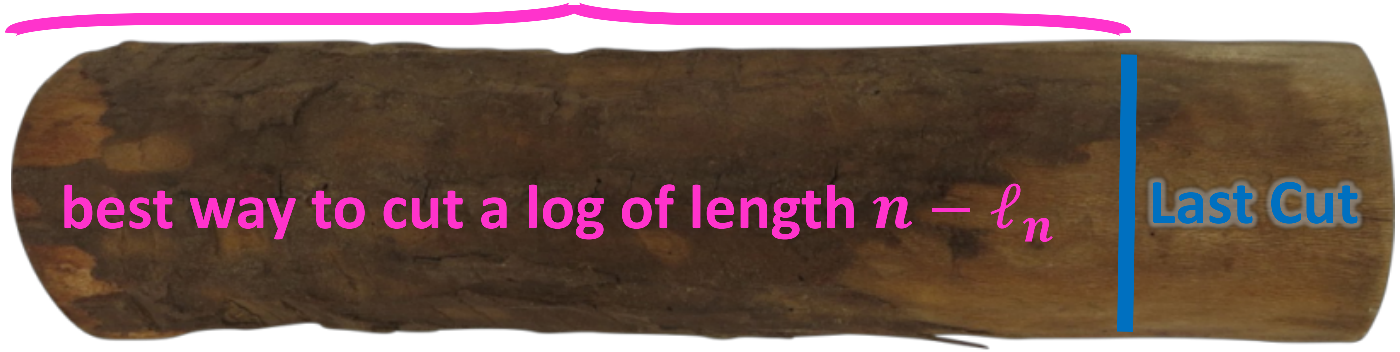
Step 1: Identify Recursive Structure

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

$\text{Cut}(n)$ = value of best way to cut a log of length n

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

$Cut(n - \ell_n)$ $\qquad$ ℓ_n



Dynamic Programming

Requires optimal substructure

- Solution to larger problem contains the solutions to smaller ones

General Blueprint:

1. Identify recursive structure of the problem
   - What is the “last thing” done?
2. Select a good order for solving subproblems
   - “Top Down:” Solve each problem recursively
   - “Bottom Up:” Iteratively solve each problem from smallest to largest

2. Select a Good Order for Solving Subproblems

Solve smallest subproblem first

$$\text{Cut}(0) = 0$$

$\text{Cut}(i)$:	0						
Length:	**0**	**1**	**2**	**3**	**4**	**5**	**6**

0

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

2. Select a Good Order for Solving Subproblems

Solve smallest subproblem first

$$\text{Cut}(1) = \text{Cut}(0) + P[1]$$

$\text{Cut}(i)$:	0	1					
Length:	**0**	**1**	**2**	**3**	**4**	**5**	**6**

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

2. Select a Good Order for Solving Subproblems

Solve smallest subproblem first

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

$\text{Cut}(i)$:	0	1	18				
Length:	**0**	**1**	**2**	**3**	**4**	**5**	**6**

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

2. Select a Good Order for Solving Subproblems

Solve smallest subproblem first

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

$\mathrm{Cut}(i)$:	0	1	18	24			
Length:	**0**	**1**	**2**	**3**	**4**	**5**	**6**

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

2. Select a Good Order for Solving Subproblems

Solve smallest subproblem first

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

$\text{Cut}(i)$:	0	1	18	24	36	50	54
Length:	**0**	**1**	**2**	**3**	**4**	**5**	**6**

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

Log Cutting Pseudocode

Run Time: $O(n^2)$

```
initialize memory C
cut(n):
    C[0] = 0
    for i = 1 to n:
        best = 0
        for j = 1 to i:
            best = max(best, C[i-j] + P[j])
        C[i] = best
    return C[n]
```

Finding the Cuts

This procedure told us the profit, but not the cuts themselves

Idea: remember the choice that you made, then backtrack

Remembering the Choices

```
initialize memory C, choices
cut(n):
    C[0] = 0
    for i = 1 to n:
        best = 0
        for j = 1 to i:
            if best < C[i-j] + P[j]:
                best = C[i-j] + P[j]
                choices[i] = j      ← size of the last cut
        C[i] = best
    return C[n], choices
```

Reconstruct the Cuts

Backtrack through the choices:

Optimal cut for log of length 10 is to first cut segment of length 3

Choices:	0	1	1	2	4	3	4	1	2	4	3
Length:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

Backtracking Pseudocode

```
i = n
while i > 0:
    print choices[i]
    i = i - choices[i]
```

Dynamic Programming

Requires optimal substructure

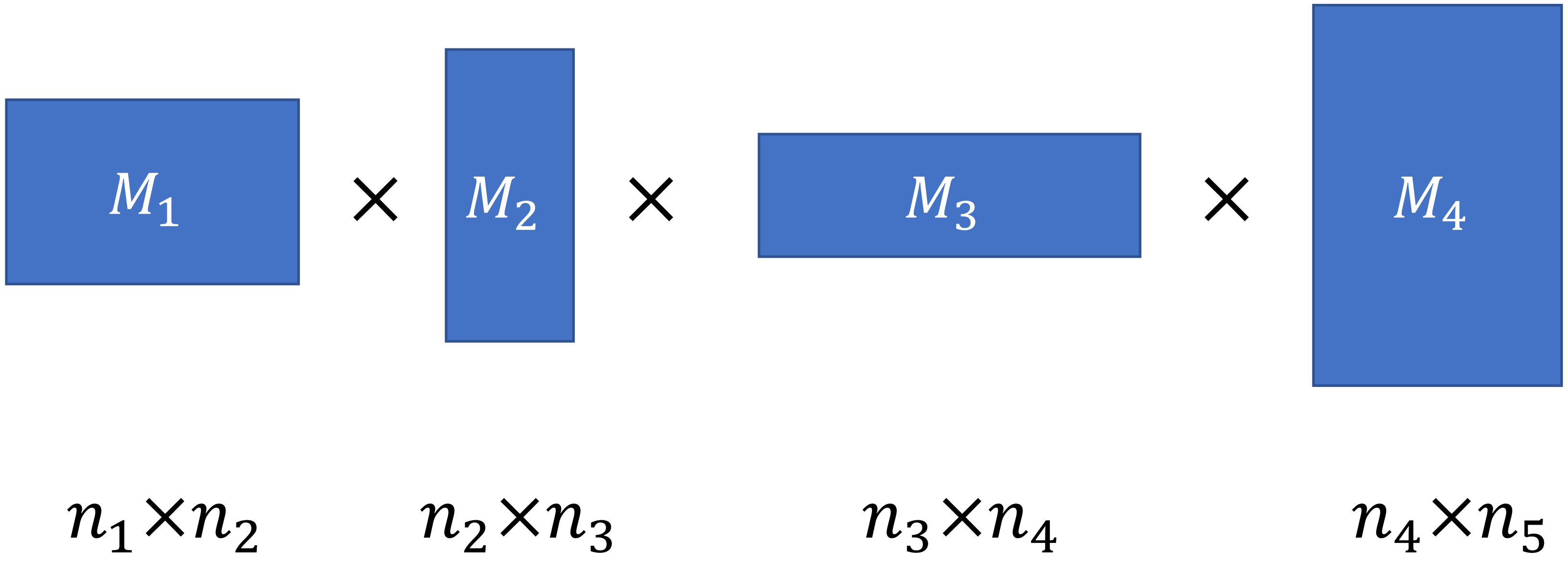
- Solution to larger problem contains the solutions to smaller ones

General Blueprint:

1. Identify recursive structure of the problem
   - What is the “last thing” done?
2. Select a good order for solving subproblems
   - “Top Down:” Solve each problem recursively
   - “Bottom Up:” Iteratively solve each problem from smallest to largest
3. Save solution to each subproblem in memory

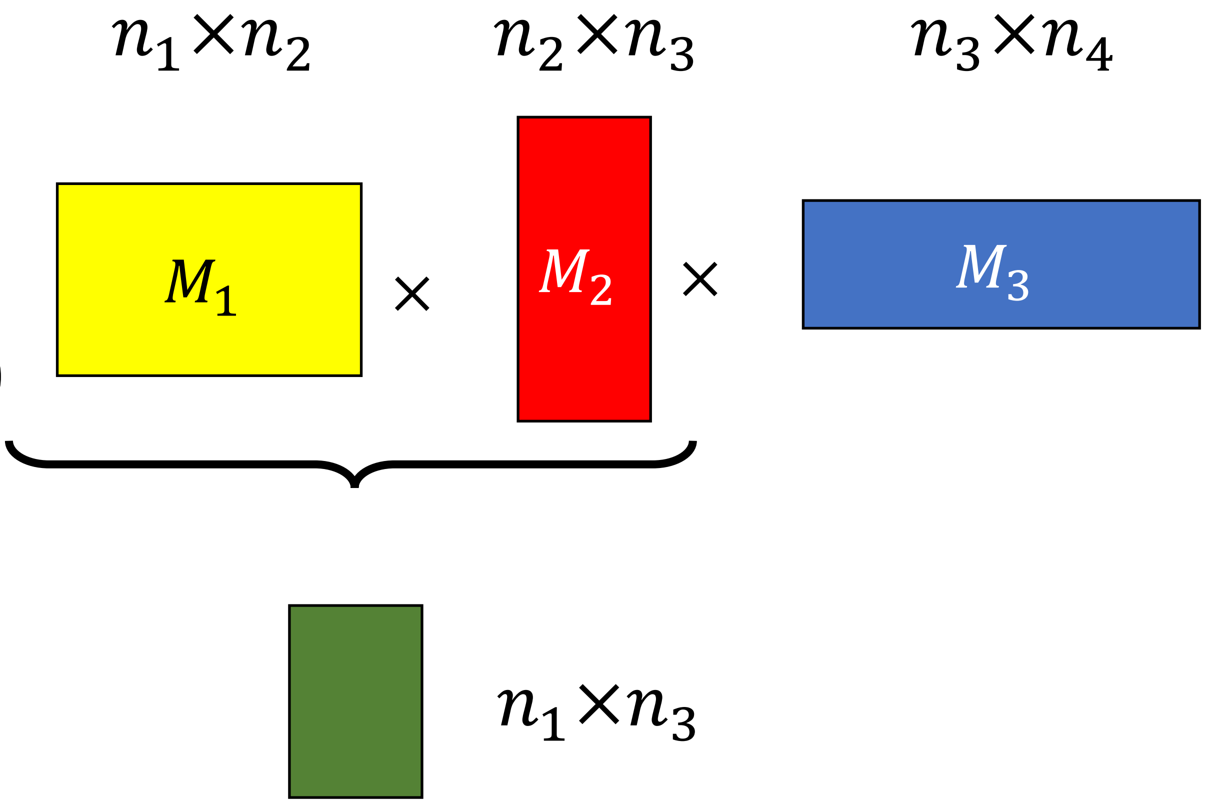
Matrix Chaining

Problem: Given a sequence of matrices $M_1, \ldots, M_n$, what is the most efficient way to multiply them?



Remember: matrix multiplication is associative

Order Matters!



$n_1 = 7$ $n_2 = 10$

$n_3 = 20$ $n_4 = 8$

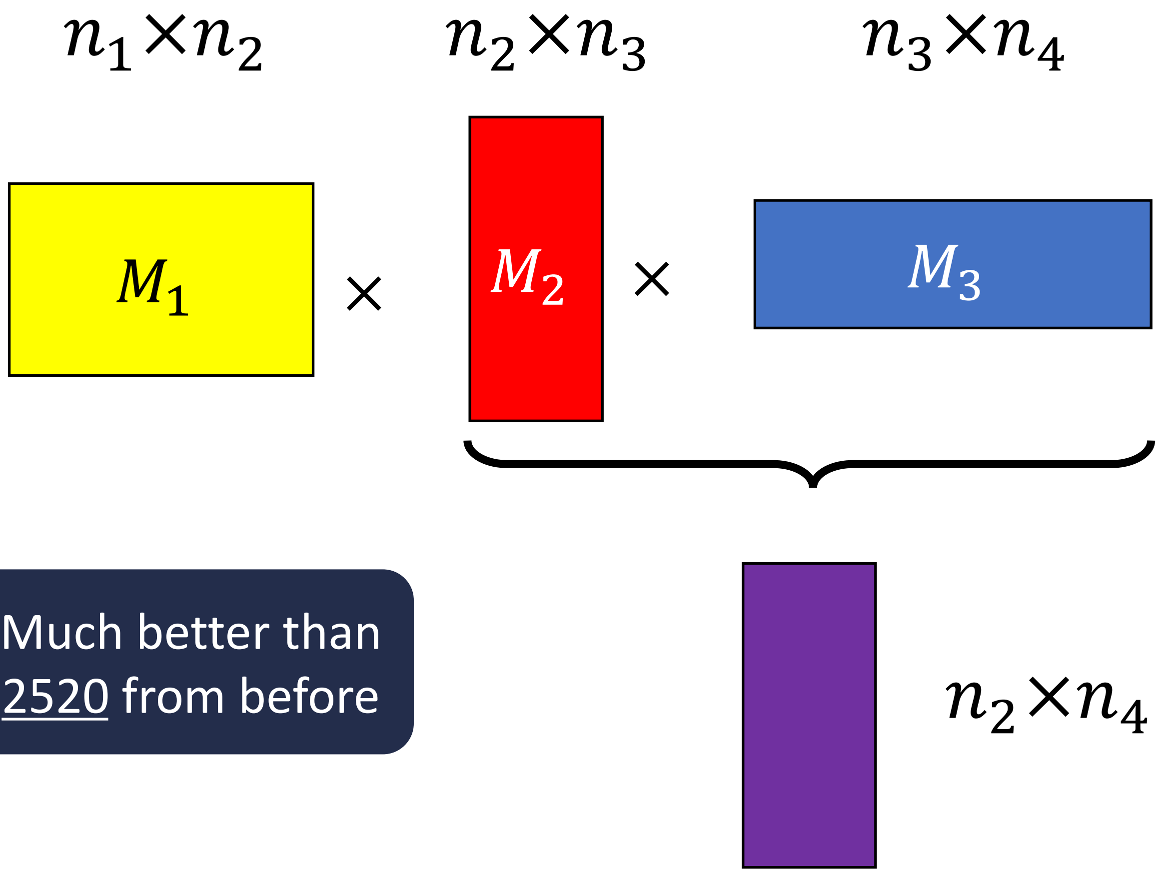
Total operations:

2520

$(M_1 \times M_2) \times M_3$

- requires $n_1 n_2 n_3 + n_1 n_3 n_4$ operations

Order Matters!



$n_1 = 7$ $n_2 = 10$

$n_3 = 20$ $n_4 = 8$

Total operations:

2160

Much better than 2520 from before

$M_1 \times (M_2 \times M_3)$

- requires $n_1 n_2 n_4 + n_2 n_3 n_4$ operations

Dynamic Programming

Requires optimal substructure

- Solution to larger problem contains the solutions to smaller ones

General Blueprint:

1. Identify recursive structure of the problem
    - What is the “last thing” done?
2. Select a good order for solving subproblems
    - “Top Down:” Solve each problem recursively
    - “Bottom Up:” Iteratively solve each problem from smallest to largest
3. Save solution to each subproblem in memory

Dynamic Programming

Requires optimal substructure

- Solution to larger problem contains the solutions to smaller ones

General Blueprint:

1. Identify recursive structure of the problem
   - What is the “last thing” done?
2. Select a good order for solving subproblems
   - “Top Down:” Solve each problem recursively
   - “Bottom Up:” Iteratively solve each problem from smallest to largest
3. Save solution to each subproblem in memory