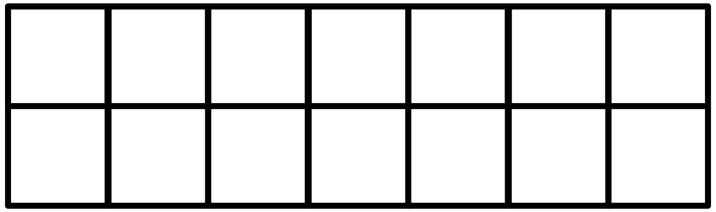
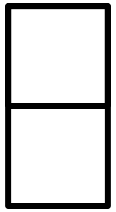
Warm Up

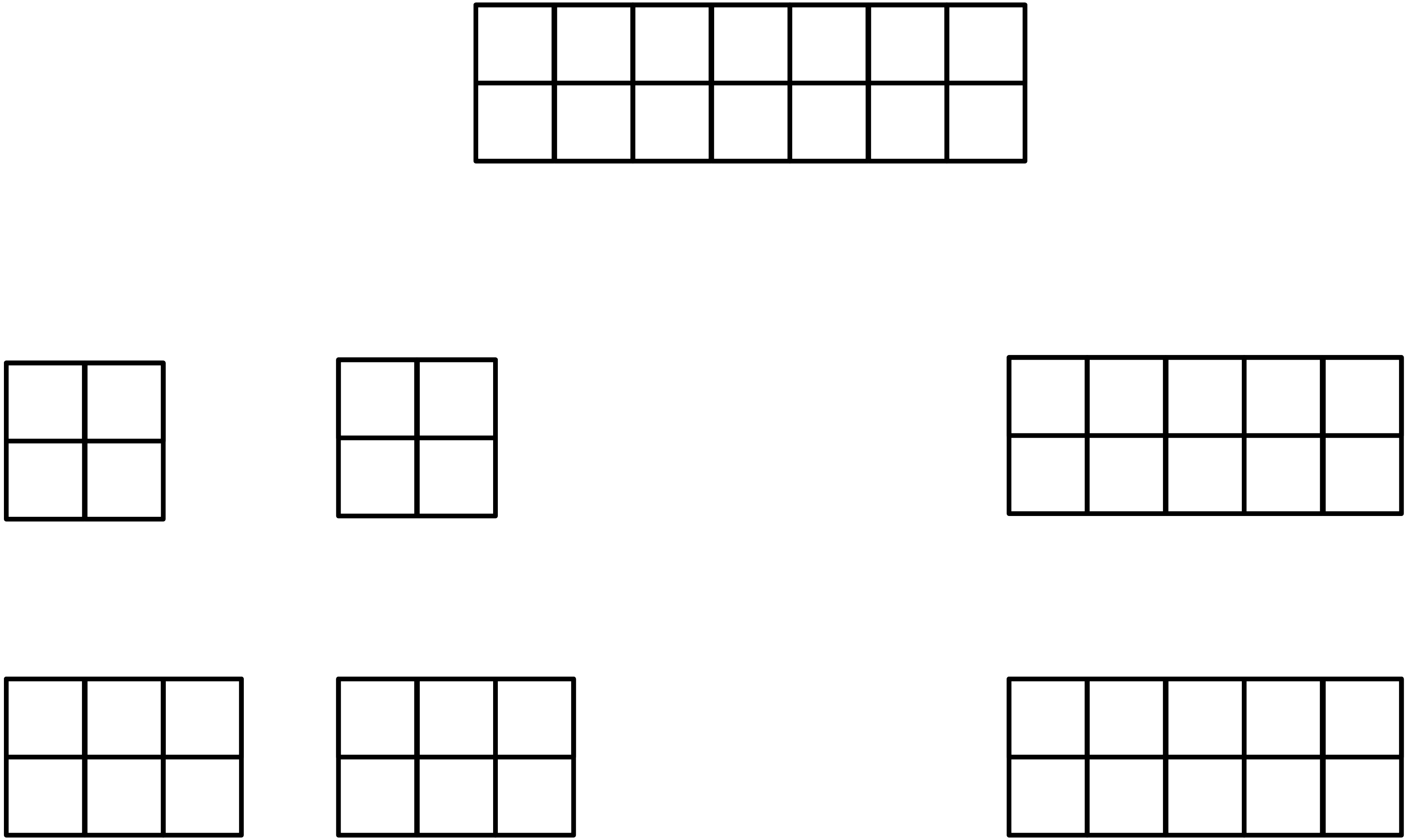
How many ways are there to tile a $2 \times n$ board with dominoes?

How many ways to tile this:



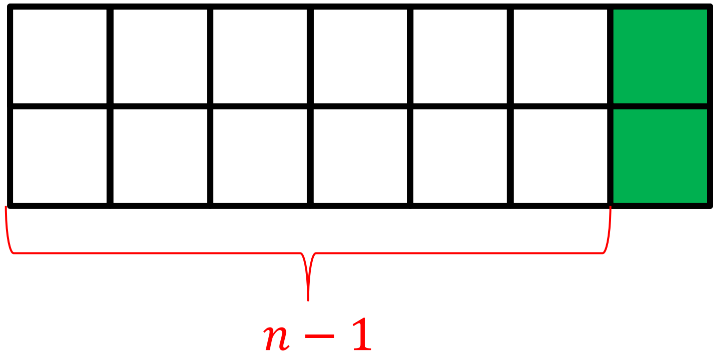
With these?





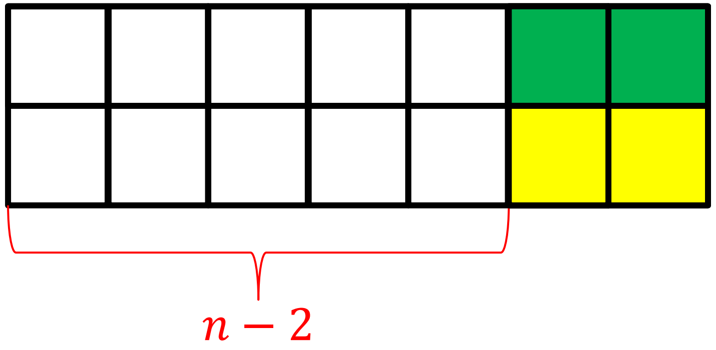
How many ways are there to tile a $2\times n$ board with dominoes?

Two ways to fill the final column:



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

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



Homeworks

- HW3 due 11pm Wednesday, October 2
  - Divide and Conquer
  - Written (use LaTeX!)
  - Submit **BOTH** a pdf and a zip file (2 separate attachments)
- HW4 out tonight!
  - Sorting, Divide and Conquer, Dynamic Programming
  - Written (use LaTeX!)
- Regrade Office Hours
  - Thursdays 11am-12pm @ Rice 210
  - Thursdays 4pm-5pm @ Rice 501

Today’s Keywords

- Maximum Sum Continuous Subarray
- Domino Tiling
- Dynamic Programming
- Log Cutting

CLRS Readings

- Chapter 8

Maximum Sum Contiguous Subarray Problem

The maximum-sum subarray of a given array of integers A is the interval $[a, b]$ such that the sum of all values in the array between a and b inclusive is maximal.

Given an array of n integers (may include both positive and negative values), give a $O(n \log n)$ algorithm for finding the maximum-sum subarray.

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

Divide in half

Recursively Solve on Right

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

Find Largest sum that spans the cut

Recursively Solve on Right
25

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

Return the Max of
Left, Right, Center

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

Find Largest sum that spans the cut
19

Recursively Solve on Right
25

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Divide and Conquer Summary

Typically multiple subproblems.
Typically all roughly the same size.

- Divide
  - Break the list in half
- Conquer
  - Find the best subarrays on the left and right
- Combine
  - Find the best subarray that "spans the divide"
  - I.e. the best subarray that ends at the divide concatenated with the best that starts at the divide

Generic Divide and Conquer Solution

```
def myDCalgo(problem):
    if baseCase(problem):
        solution = solve(problem) #brute force if necessary
        return solution
    subproblems = Divide(problem)
    for sub in subproblems:
        subsolutions.append(myDCalgo(sub))
    solution = Combine(subsolutions)
    return solution
```

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

```
def MSCS(list):
    if list.length < 2:
        return list[0]      #list of size 1 the sum is maximal
    {listL, listR} = Divide (list)
    for list in {listL, listR}:
        subSolutions.append(MSCS(list))
    solution = max(solnL, solnR, span(listL, listR))
    return solution
```

Types of “Divide and Conquer”

- Divide and Conquer
  - Break the problem up into several subproblems of roughly equal size, recursively solve
  - E.g. Karatsuba, Closest Pair of Points, Mergesort…
- Decrease and Conquer
  - Break the problem into a single smaller subproblem, recursively solve
  - E.g. Impossible Missions Force (Double Agents), Quickselect, Binary Search

Pattern So Far

- Typically looking to divide the problem by some fraction (½, ¼ the size)
- Not necessarily always the best!
  - Sometimes, we can write faster algorithms by finding unbalanced divides.
  - Chip and Conquer

Chip (Unbalanced Divide) and Conquer

- Divide
  - Make a subproblem of all but the last element
- Conquer
  - Find **B**est **S**ubarray (sum) on the **L**eft ($BSL(n-1)$)
  - Find the **B**est subarray **E**nding at the **D**ivide ($BED(n-1)$)
- Combine
  - New **B**est **E**nding at the **D**ivide:
    - $BED(n) = \max(BED(n-1) + arr[n],\ 0)$
  - New **B**est **S**ubarray (sum) on the **L**eft:
    - $BSL(n) = \max\left(BSL(n-1),\ BED(n)\right)$

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

Recursively
Solve on Left
25

Find Largest
sum ending at
the divide
22

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
25

Divide

Find Largest sum ending at the divide
0

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

Recursively Solve on Left
25

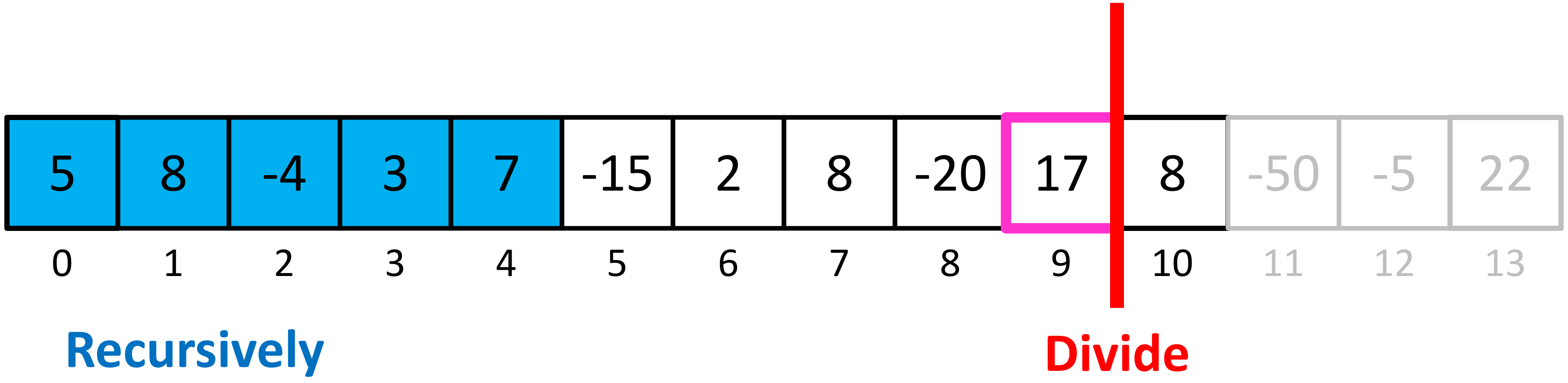
Find Largest sum ending at the divide
0

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

Recursively Solve on Left
25

Find Largest sum ending at the divide
25



Recursively
Solve on Left
19

Find Largest
sum ending at
the divide
17

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

Recursively
Solve on Left
19

Find Largest
sum ending at
the divide
0

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
13

Divide

Find Largest sum ending at the divide
12

Chip (Unbalanced Divide) and Conquer

- Divide
  - Make a subproblem of all but the last element
- Conquer
  - Find **B**est **S**ubarray (sum) on the **L**eft ($BSL(n-1)$)
  - Find the **B**est subarray **E**nding at the **D**ivide ($BED(n-1)$)
- Combine
  - New **B**est **E**nding at the **D**ivide:
    - $BED(n) = \max(BED(n-1) + arr[n],\ 0)$
  - New **B**est **S**ubarray (sum) on the **L**eft:
    - $BSL(n) = \max\big(BSL(n-1),\ BED(n)\big)$

Was unbalanced better? YES

- Old:
  - We divided in Half
  - We solved 2 different problems:
    - Find the best overall on BOTH the left/right
    - Find the best which end/start on BOTH the left/right respectively
  - Linear time combine

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$T(n) = \Theta(n \log n)$$

- New:
  - We divide by 1, n-1
  - We solve 2 different problems:
    - Find the best overall on the left ONLY
    - Find the best which ends on the left ONLY
  - Constant time combine

$$T(n) = 1T(n-1) + 1$$

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

MSCS Problem - Redux

- Solve in $O(n)$ by increasing the problem size by 1 each time.
- Idea: Only include negative values if the positives on both sides of it are "worth it"

$\Theta(n)$ Solution

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

Begin here

Remember two values:

Best So Far
5

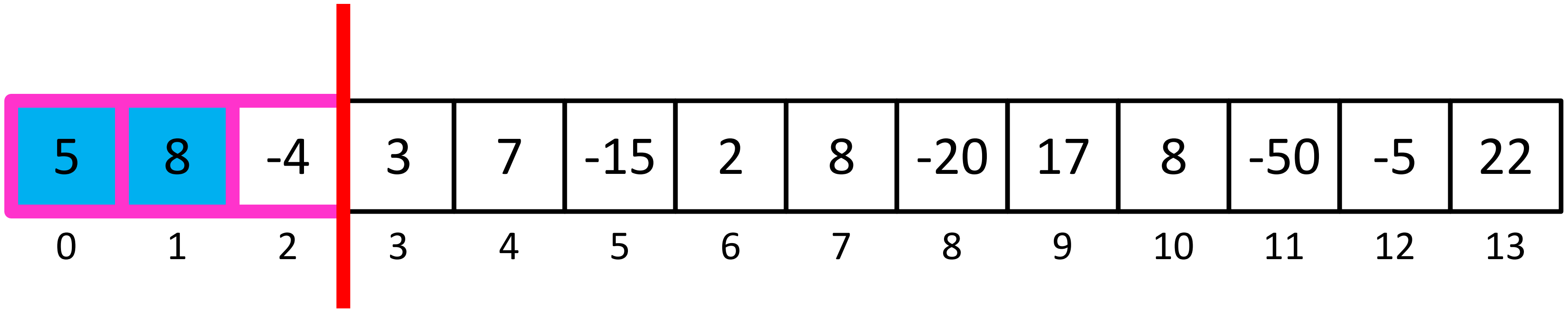
Best ending here
5

$\Theta(n)$ Solution

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

Remember two values: **Best So Far** **13** **Best ending here** **13**

$\Theta(n)$ Solution



Remember two values:

Best So Far
13

Best ending here
9

$\Theta(n)$ Solution

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

Remember two values:

Best So Far
13

Best ending here
12

$\Theta(n)$ Solution

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

Remember two values:

Best So Far
19

Best ending here
19

$\Theta(n)$ Solution

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

Remember two values:

Best So Far
19

Best ending here
4

$\Theta(n)$ Solution

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

Remember two values:

Best So Far
19

Best ending here
14

$\Theta(n)$ Solution

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

Remember two values:

Best So Far
19

Best ending here
0

$\Theta(n)$ Solution

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

Remember two values:

Best So Far
19

Best ending here
17

$\Theta(n)$ Solution

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

Remember two values:

Best So Far
25

Best ending here
25

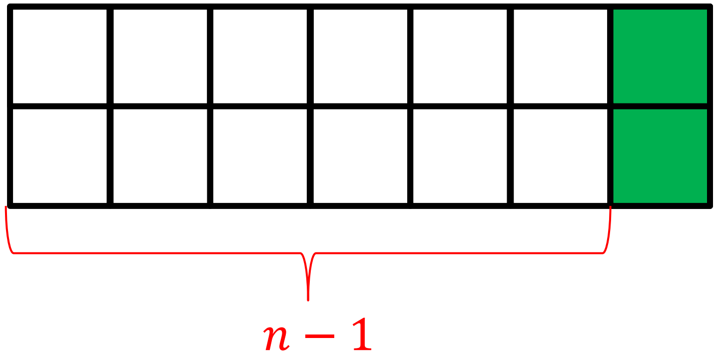
End of Midterm Exam Materials!

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

Back to Tiling

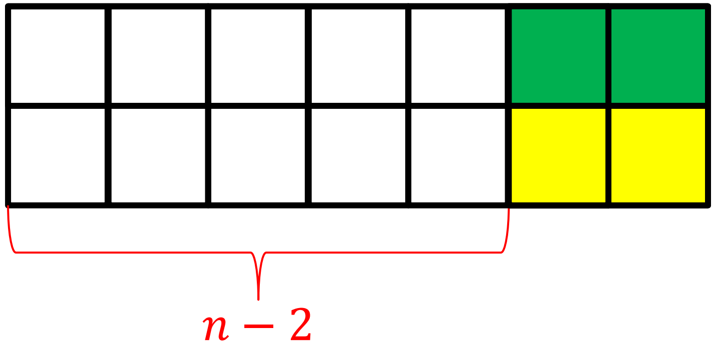
How many ways are there to tile a $2 \times n$ board with dominoes?

Two ways to fill the final column:



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

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

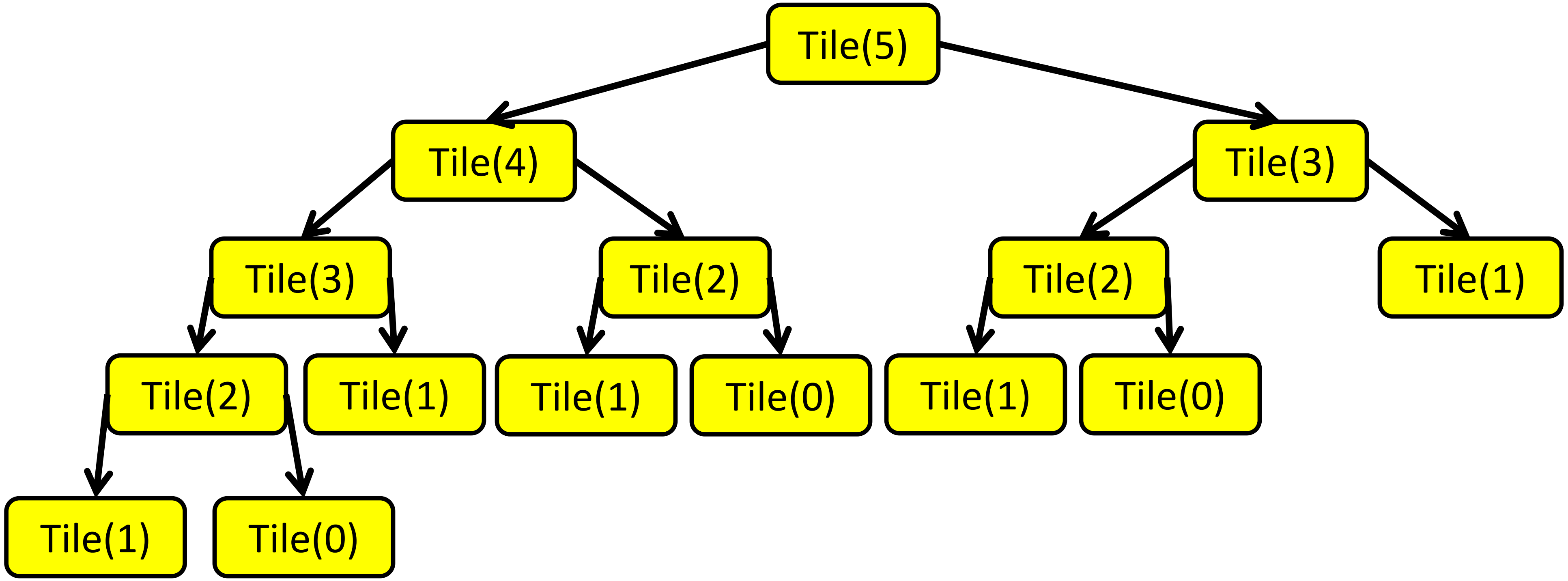


How to compute $Tile(n)$?

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

Problem?

Recursion Tree



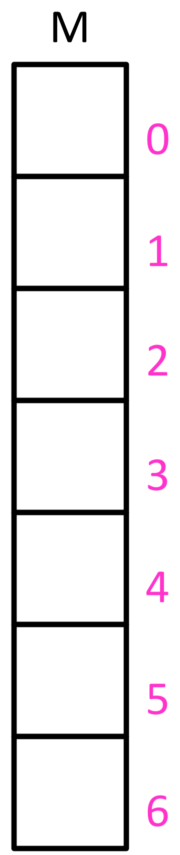
Many redundant calls!

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]
```



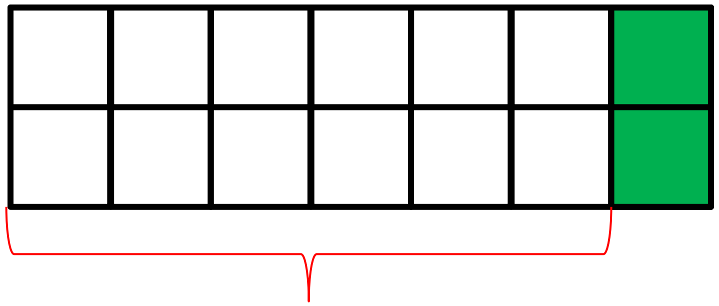
Computing $Tile(n)$ with Memory - "Top Down"

```
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]
```

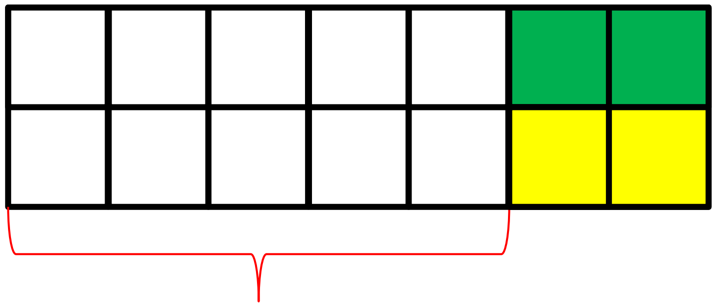
M	
1	0
1	1
2	2
3	3
5	4
8	5
13	6

Dynamic Programming

- Requires Optimal Substructure
  - – Solution to larger problem contains the solutions to smaller ones
- Idea:
  1. Identify recursive structure of the problem
     - What is the “last thing” done?



$n-1$



$n-2$

Generic Divide and Conquer Solution

```
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 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
```

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

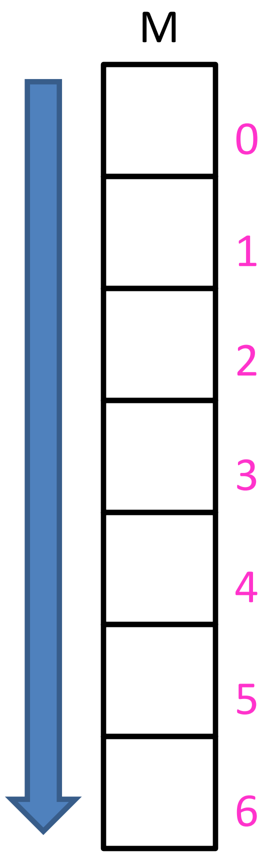
```
Initialize Memory M
Tile(n):
        if n < 2:
                return 1
        if M[n] is filled:
                return M[n]
        M[n] = Tile(n-1)+Tile(n-2)
        return M[n]
```

M	
1	0
1	1
2	2
3	3
5	4
8	5
13	6

Recursive calls happen in a predictable order

Better $Tile(n)$ 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]
```



Dynamic Programming

- Requires Optimal Substructure
  - Solution to larger problem contains the solutions to smaller ones
- Idea:
  1. Identify recursive structure of the problem
     - What is the “last thing” done?
  2. Select a good order for solving subproblems
     - Usually smallest problem first
     - “Bottom up”

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

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

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

Brute Force: $O(2^n)$

Greedy won't work

- 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

- 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

Dynamic Programming

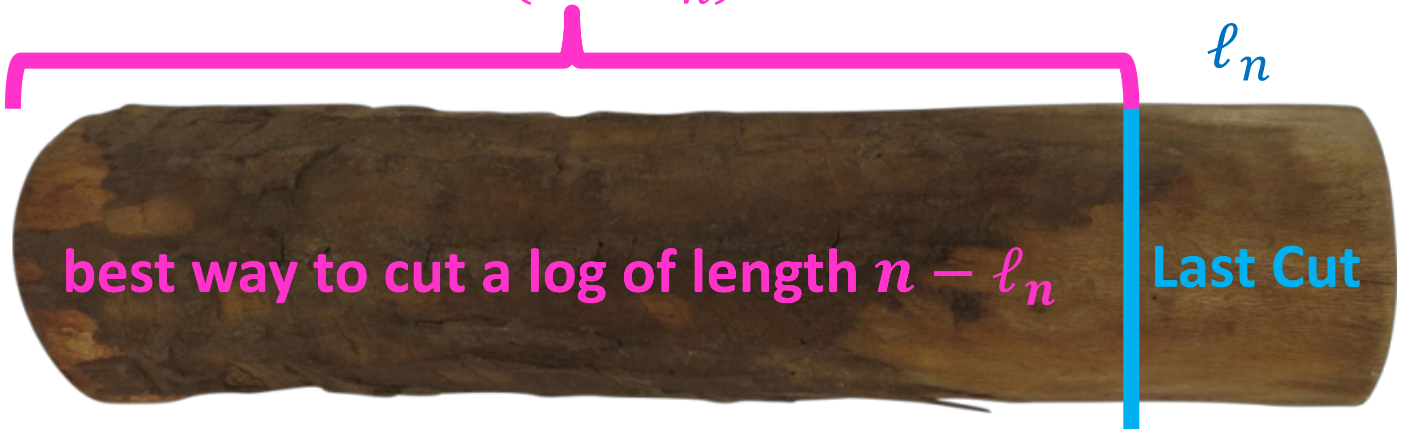
- Idea:
  1. Identify recursive structure of the problem
     - What is the "last thing" done?
  2. Select a good order for solving subproblems
     - Usually smallest problem first
     - "Bottom up"

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

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



Dynamic Programming

- Idea:
  1. Identify recursive structure of the problem
     - What is the "last thing" done?
  2. Select a good order for solving subproblems
     - Usually smallest problem first
     - "Bottom up"

2. Select a Good Order for Solving Subproblems

Solve Smallest subproblem first

$$Cut(0) = 0$$

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

0

2. Select a Good Order for Solving Subproblems

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

2. Select a Good Order for Solving Subproblems

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

2. Select a Good Order for Solving Subproblems

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

2. Select a Good Order for Solving Subproblems

Solve Smallest subproblem first

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

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

4

Log Cutting Pseudocode

```
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]
```

Run Time: $O(n^2)$