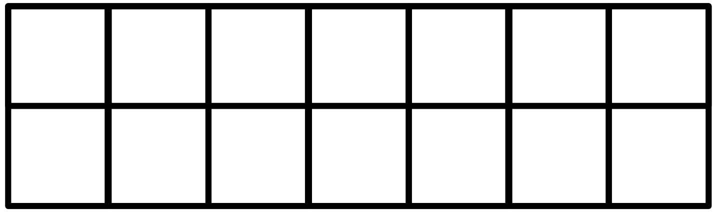
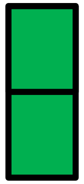
Warm Up

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

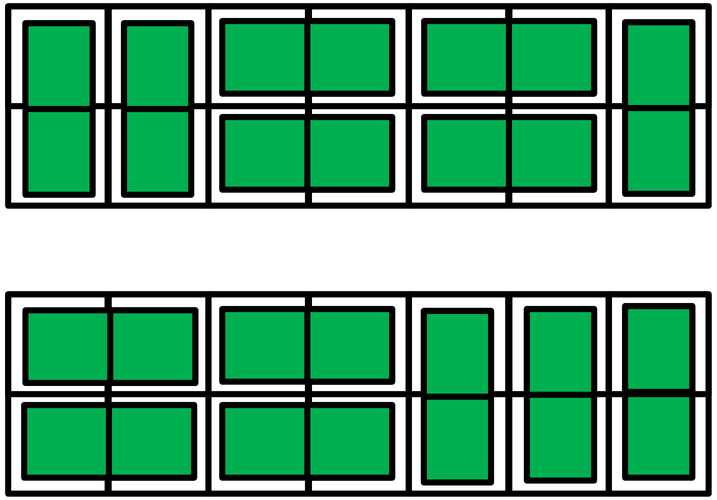
How many ways to tile this:



With these?



For Example:



CS 3100
Data Structures and Algorithms 2

Lecture 16: Dynamic Programming

Co-instructors: Robbie Hott and Tom Horton

Fall 2023

Readings in CLRS 4th edition:

- Chapter 14

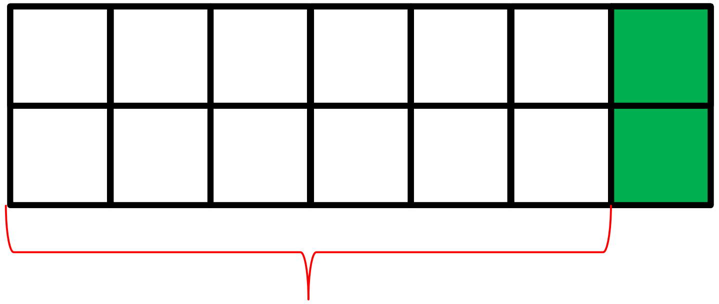
Announcements

- PS6 due yesterday (3/20)
- PA3 due tomorrow (3/22)
- PS7 releasing today, due Wednesday (3/27)
- Grading update
  - Next week, the instructors will meet to make decisions about any general grading changes/modifications for Quiz 1. If there are changes, we'll make an announcement.
  - We will address Quiz 1 regrade requests after next week's meeting.
  - We are currently grading: Quiz 2 Question 1 (the last question to be graded), PS4, PS5.
- Office hours (reminder)
  - Prof Hott Office Hours: Traveling this week
  - Prof Pettit Office Hours: Mondays and Fridays 2:30-4:00p
  - TA office hours posted on our website
  - Office hours are not for "pre-grading"

Warm Up

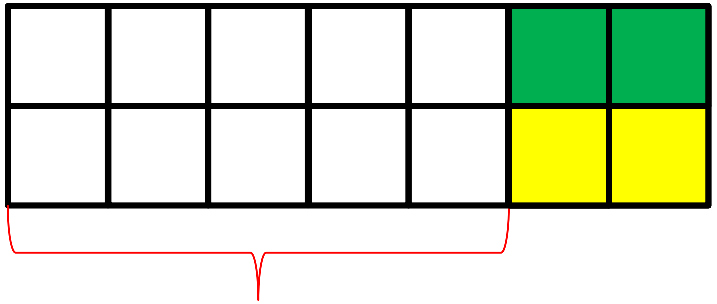
How many ways are there to tile a $2 \times n$ board with dominoes?

Two ways to fill the final column:



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

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

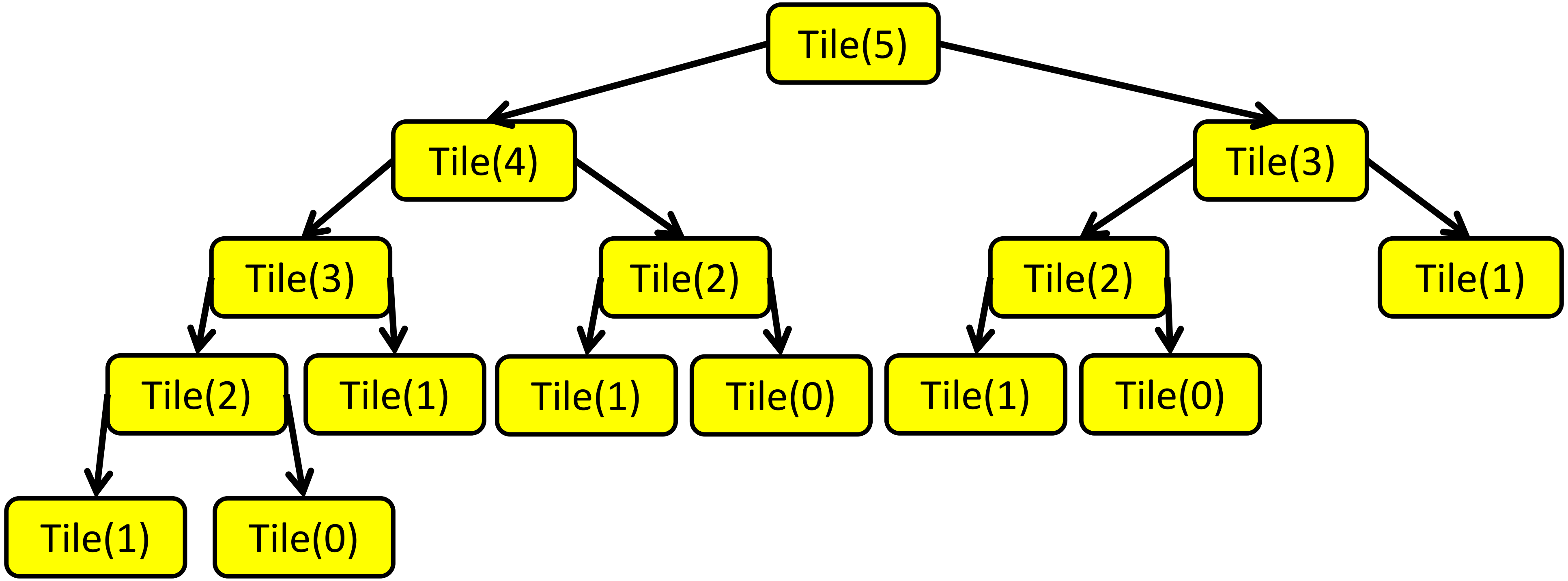


How to compute $Tile(n)$?

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

Problem?

Recursion Tree



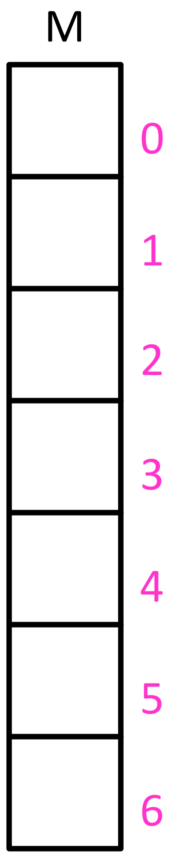
Many redundant calls!

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]
```



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

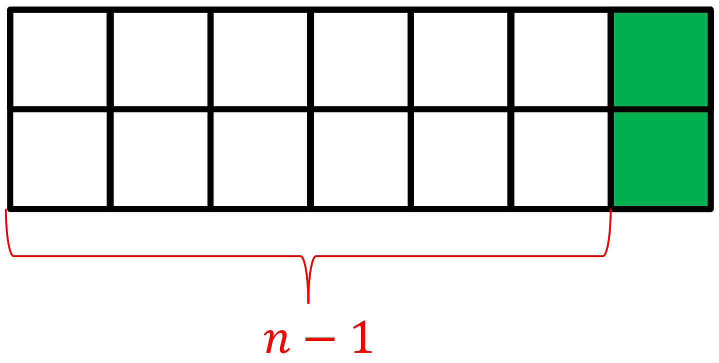
Computing $Tile(n)$ with Memory - “Top Down”

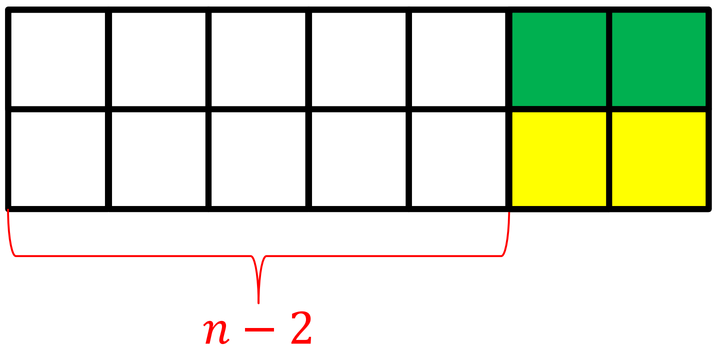
```
Initialize Memory M
Tile(n):
    if n < 2:
        return 1
    if M[n] is filled:
        return M[n]
    M[n] = Tile(n-1)+Tile(n-2)
    return M[n]
```

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 (optimal) solutions to smaller ones
- Idea:
  1. Identify recursive structure of the problem
     - What is the "last thing" done?





Dynamic Programming

- Requires Optimal Substructure
  - Solution to larger problem contains the (optimal) solutions to smaller ones
- Idea:
  1. Identify the recursive structure of the problem
     - What is the "last thing" done?
  2. Save the solution to each subproblem in memory

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 Solution

```
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"

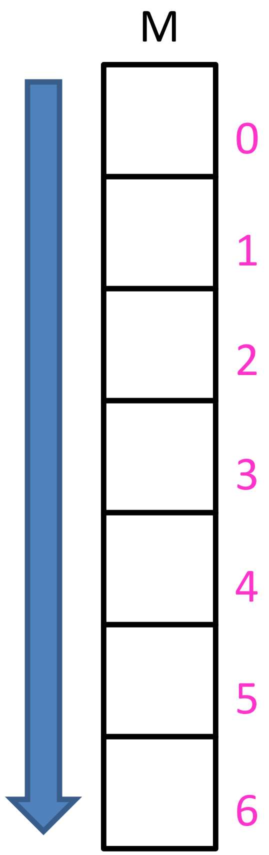
```
Initialize Memory M
Tile(n):
    if n < 2:
        return 1
    if M[n] is filled:
        return M[n]
    M[n] = Tile(n-1)+Tile(n-2)
    return M[n]
```

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

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

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 $\ell_1, \ldots, \ell_k$ such that:
$\sum \ell_i = n$
to maximize $\sum P[\ell_i]$

Brute Force: $O(2^n)$

Greedy Algorithm

- Greedy algorithms 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 Algorithm

- Greedy algorithms 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

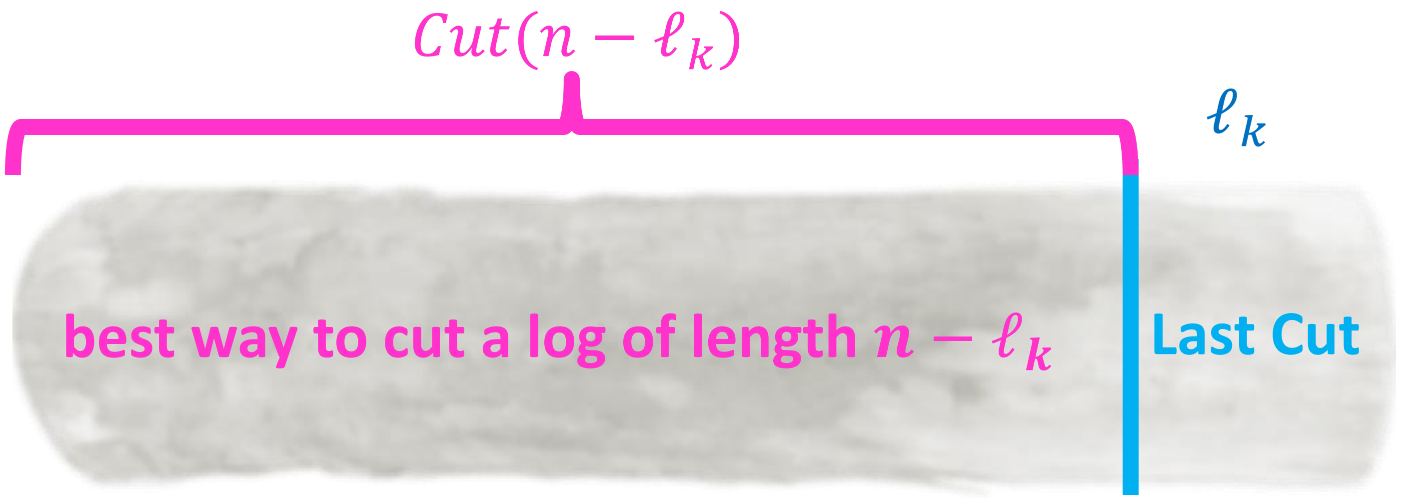
- Requires Optimal Substructure
  - Solution to larger problem contains the solutions to smaller ones
- Idea:
  1. Identify the recursive structure of the problem
     - What is the "last thing" done?
  2. Save the solution to each subproblem in memory
  3. Select a good order for solving subproblems
     - "Top Down": Solve each recursively
     - "Bottom Up": Iteratively solve smallest to largest

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

- Requires Optimal Substructure
  - Solution to larger problem contains the solutions to smaller ones
- Idea:
  1. Identify the recursive structure of the problem
     - What is the "last thing" done?
  2. Save the solution to each subproblem in memory
  3. Select a good order for solving subproblems
     - "Top Down": Solve each recursively
     - "Bottom Up": Iteratively solve smallest to largest

3. 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

3. 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

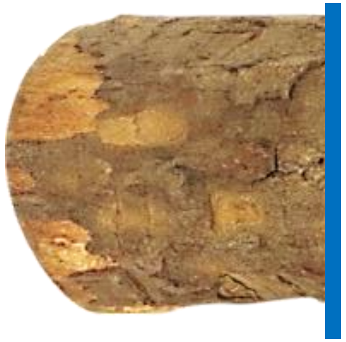
3. Select a Good Order for Solving Subproblems

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



3. 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

3. Select a Good Order for Solving Subproblems

Solve Smallest subproblem first

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

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

4

Log Cutting Pseudocode

```
Initialize Memory C
Cut(n):
    C[0] = 0
    for i=1 to n:  // log size
        best = 0
        for j = 1 to i: // last cut
            best = max(best, C[i-j] + P[j])
        C[i] = best
    return C[n]
```

Run Time: $O(n^2)$

How to find the cuts?

- This procedure told us the profit, but not the cuts themselves
- Idea: remember the choice that you made, then backtrack

Remember the choice made

```
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
        C[i] = best
    return C[n]
```

Gives the size of the last cut

Reconstruct the Cuts

- Backtrack through the choices

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

Example to demo Choices[] only. Profit of 20 is not optimal!

Backtracking Pseudocode

```
i = n
while i > 0:
        print Choices[i]
        i = i – Choices[i]
```

Our Example: Getting Optimal Solution

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

i	0	1	2	3	4	5	6	7	8	9	10
C[i]	0	1	5	8	10	13	17	18	22	25	30
Choice[i]	0	1	2	3	2	2	6	1	2	3	10

- If n were 5
  - Best score is 13
  - Cut at Choice[n]=2, then cut at Choice[n-Choice[n]]= Choice[5-2]= Choice[3]=3
- If n were 7
  - Best score is 18
  - Cut at 1, then cut at 6

Dynamic Programming

- Requires Optimal Substructure
  - – Solution to larger problem contains the solutions to smaller ones
- Idea:
  1. Identify the recursive structure of the problem
     - What is the "last thing" done?
  2. Save the solution to each subproblem in memory
  3. Select a good order for solving subproblems
     - "Top Down": Solve each recursively
     - "Bottom Up": Iteratively solve smallest to largest