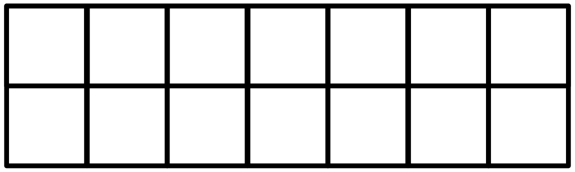
CS4102 Algorithms

Spring 2020 – Horton's Slides

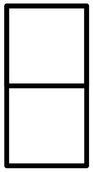
Warm Up

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

How many ways to tile this:

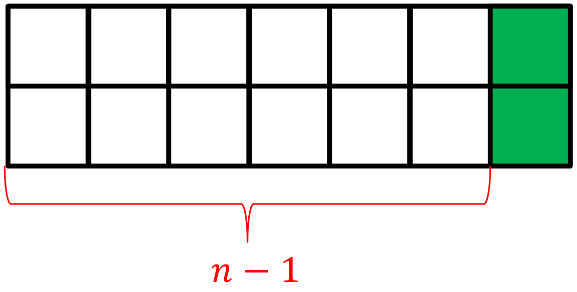


With these?



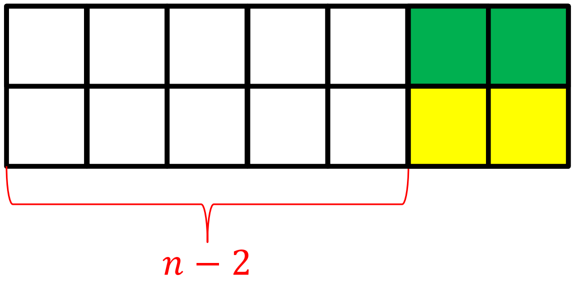
How many ways are there to tile a $2 \times n$ board with dominoes?

Two ways to fill the final column:



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

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



Homeworks

- HW4 due 11pm Thursday, February 27, 2020
  - Divide and Conquer and Sorting
  - Written (use LaTeX!)
  - Submit BOTH a pdf and a zip file (2 separate attachments)
- Midterm: March 4
- Regrade Office Hours
  - Fridays 2:30pm-3:30pm (Rice 210)

Today's Keywords

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

CLRS Readings

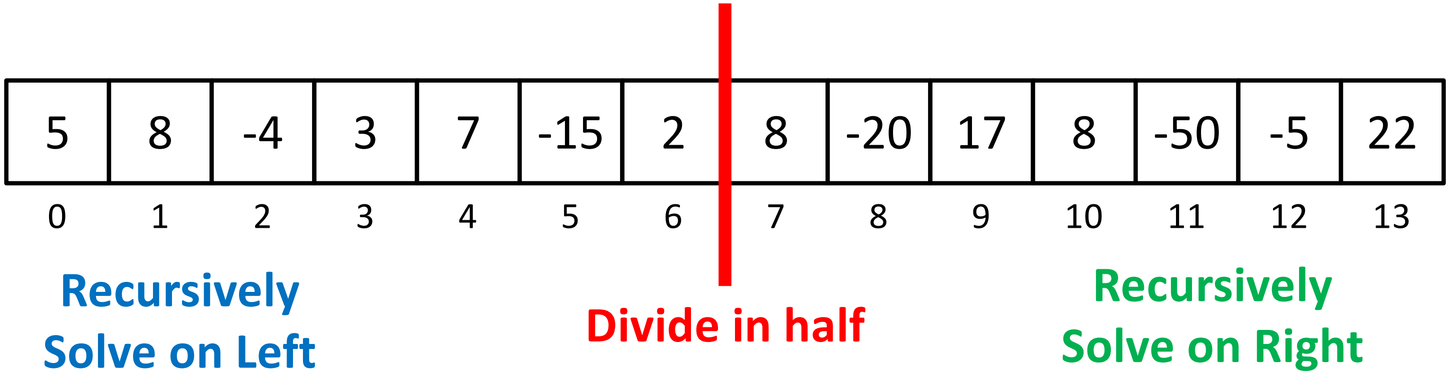
- Chapter 15
  - Section 15.1, Log/Rod cutting, optimal substructure property
    - Note: r_i in book is called Cut() or C[] in our slides. We use their example.
  - Section 15.3, More on elements of DP, including optimal substructure property
  - Section 15.2, matrix-chain multiplication (later example)
  - Section 15.4, longest common subsequence (even later example)

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

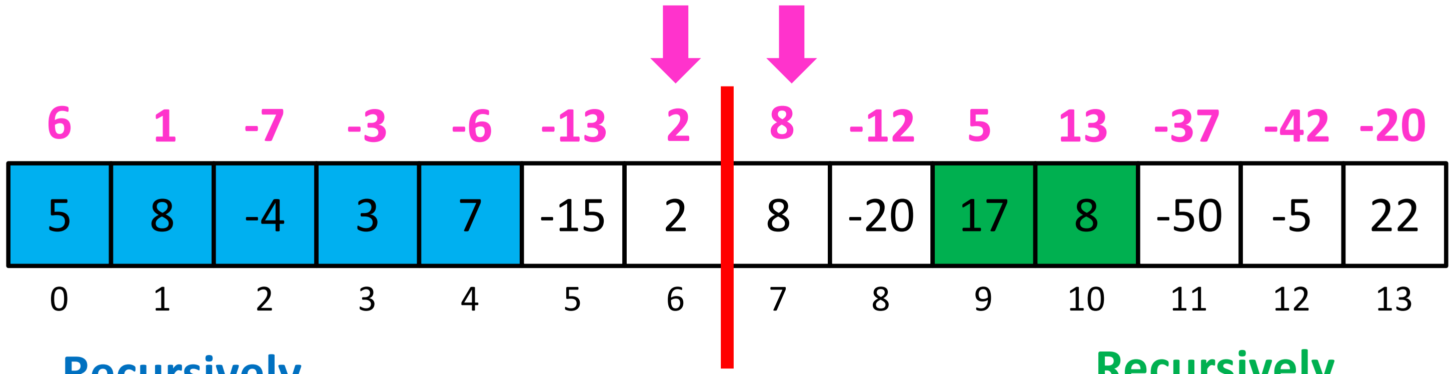
Recursively Solve on Left

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

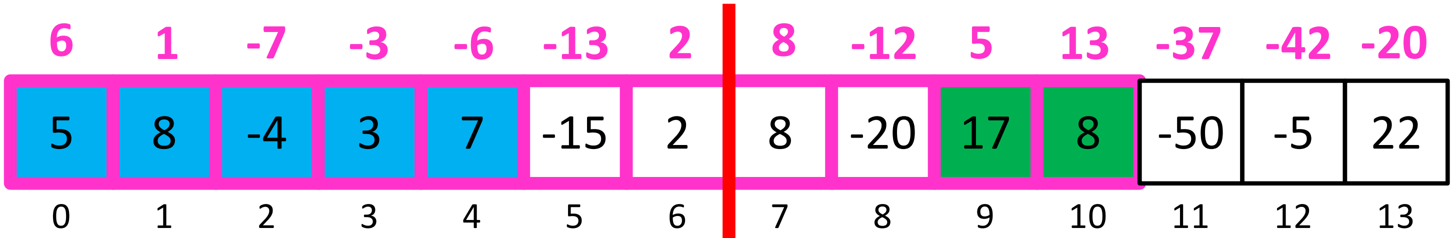
Find Largest sum that spans the cut

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”

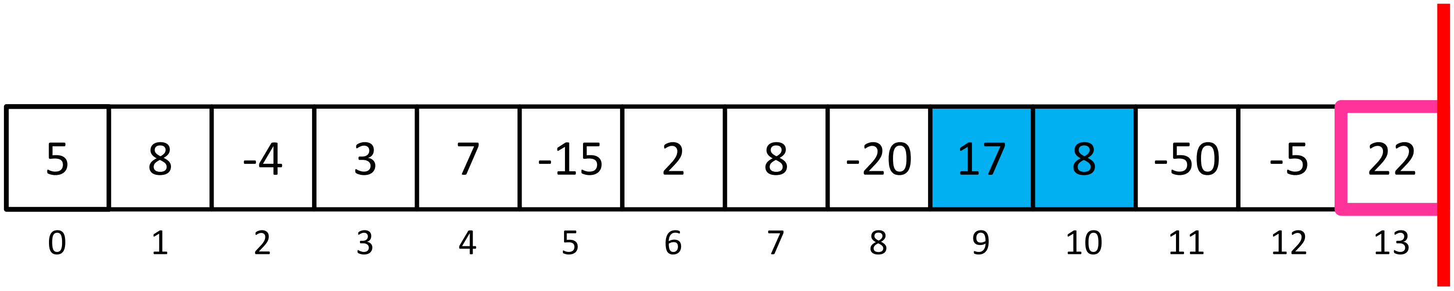
- Divide and Conquer
  - Break the problem up into several subproblems of roughly equal size, recursively solve
  - E.g. Karatsuba, Closest Pair of Points, Mergesort…
- Decrease and Conquer
  - Break the problem into a single smaller subproblem, recursively solve
  - E.g. Impossible Missions Force (Double 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:
    - $BSL(n) = \max\big(BSL(n-1),\ \ BED(n)\big)$

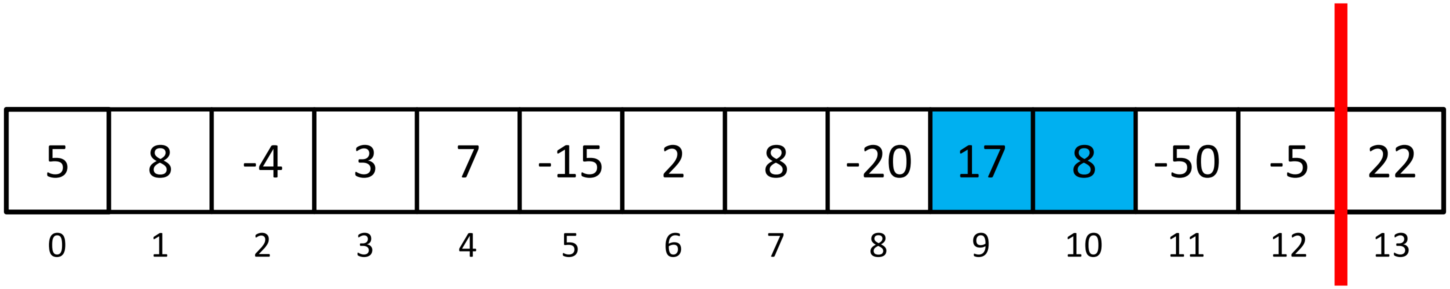


5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Divide

Recursively
Solve on Left
25

Find Largest
sum ending at
the divide
22

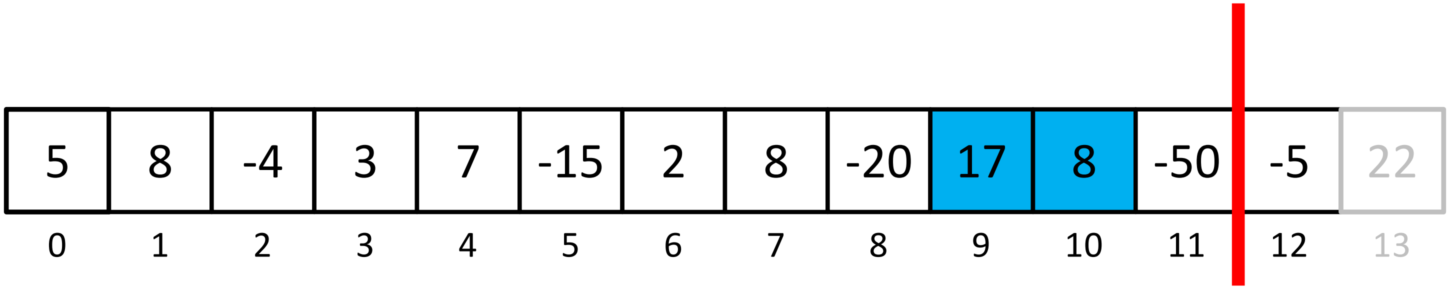


5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Recursively Solve on Left
25

Divide

Find Largest sum ending at the divide
0

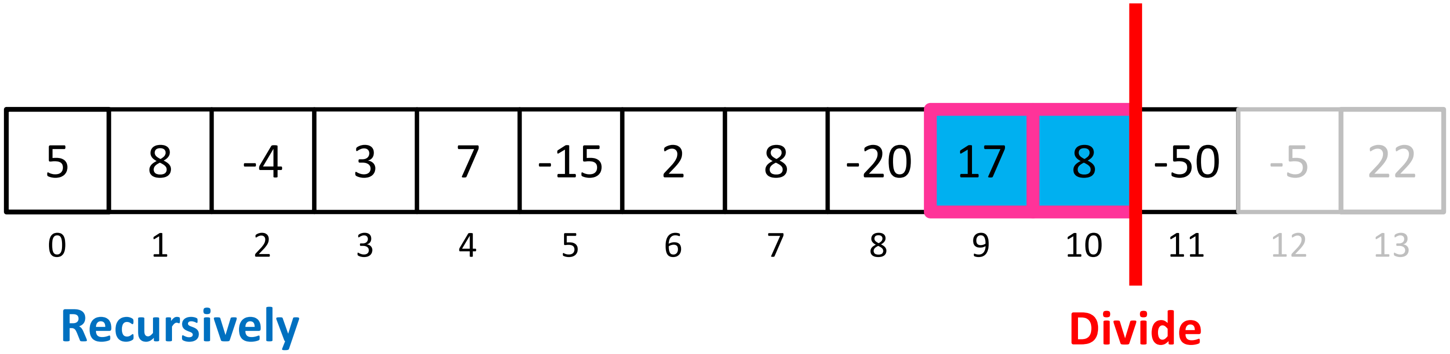


5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Divide

Recursively Solve on Left
25

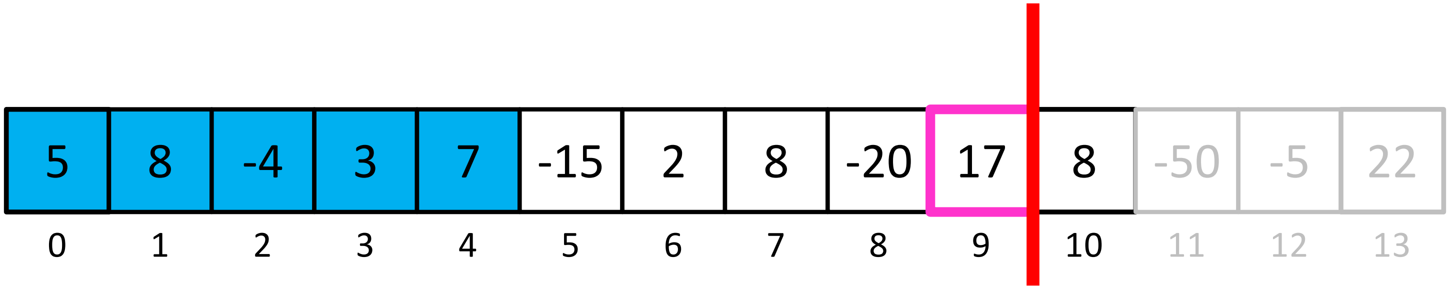
Find Largest sum ending at the divide
0



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

Recursively Solve on Left
25

Find Largest sum ending at the divide
25

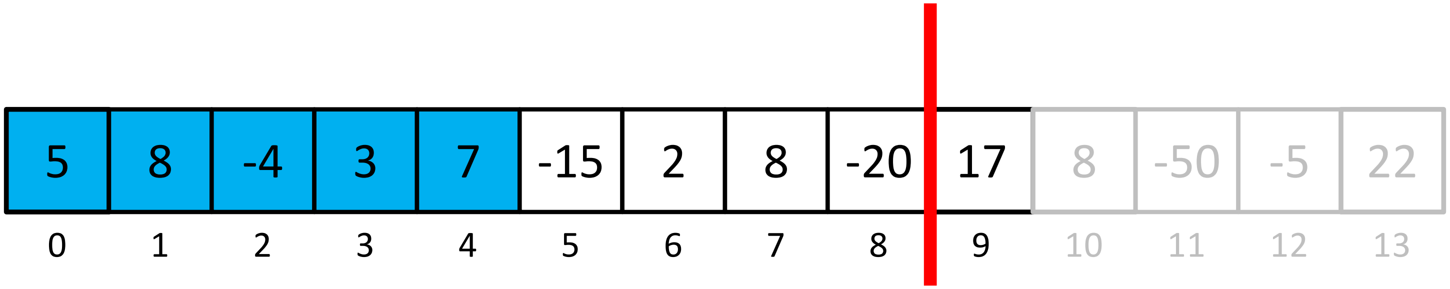


5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Recursively Solve on Left
19

Divide

Find Largest sum ending at the divide
17

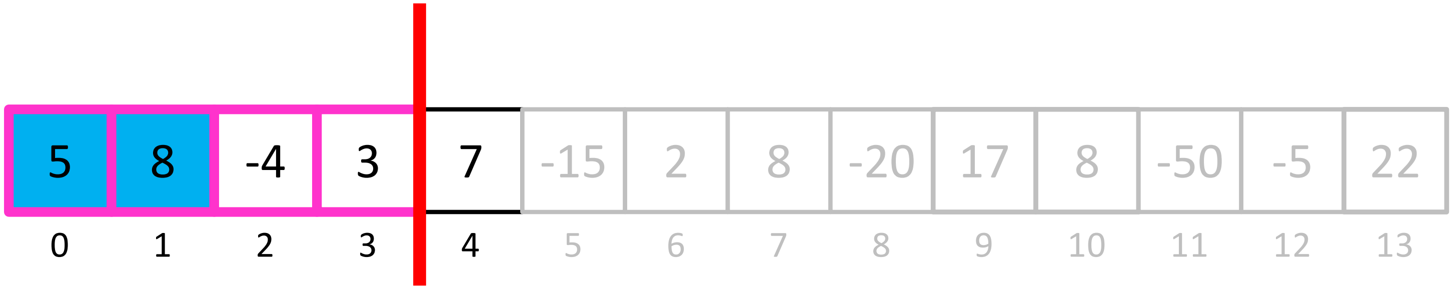


0	1	2	3	4	5	6	7	8	9	10	11	12	13
5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22

Recursively
Solve on Left
19

Divide

Find Largest
sum ending at
the divide
0



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

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

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

- 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

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

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

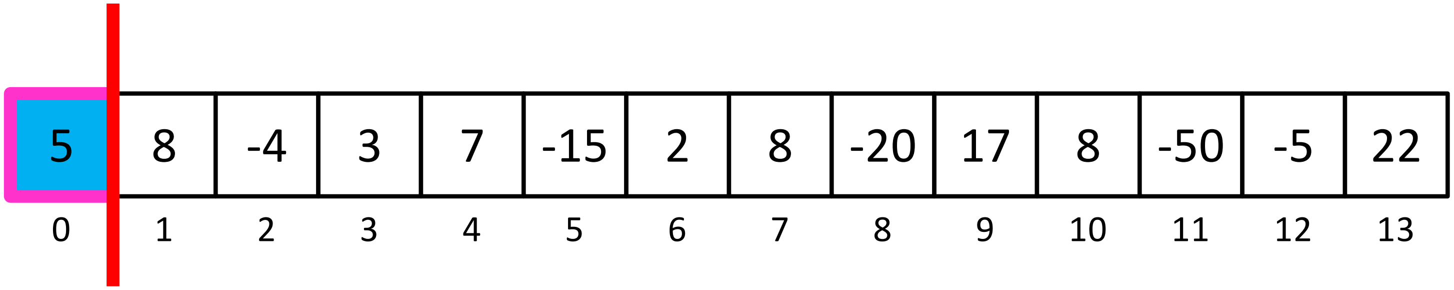
- New:
  - We divide by 1, n-1
  - We solve 2 different problems:
    - Find the best overall on the left ONLY
    - Find the best which ends on the left ONLY
  - Constant time combine

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

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



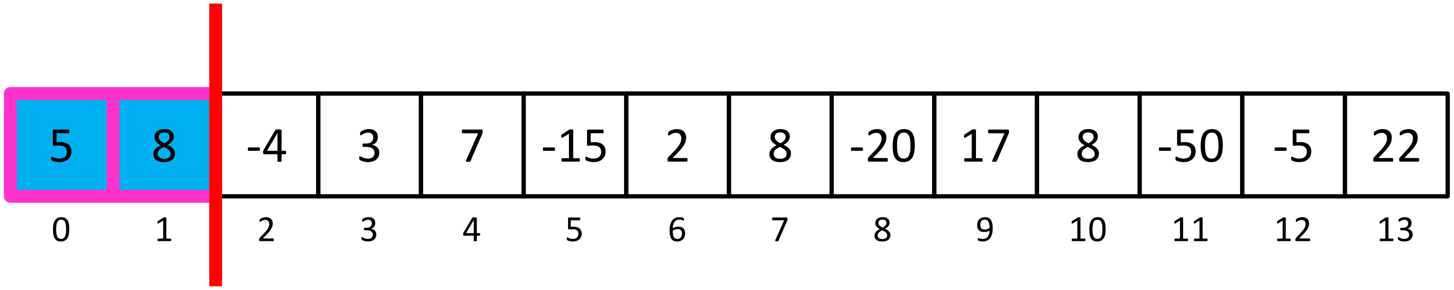
5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Begin here

Remember two values:

Best So Far	Best ending here
5	**5**

$\Theta(n)$ Solution



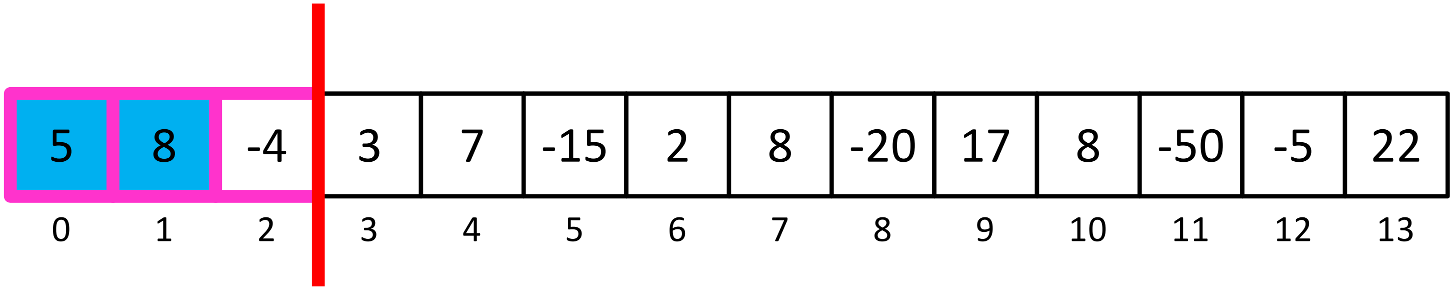
5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Remember two values:

Best So Far
13

Best ending here
13

$\Theta(n)$ Solution



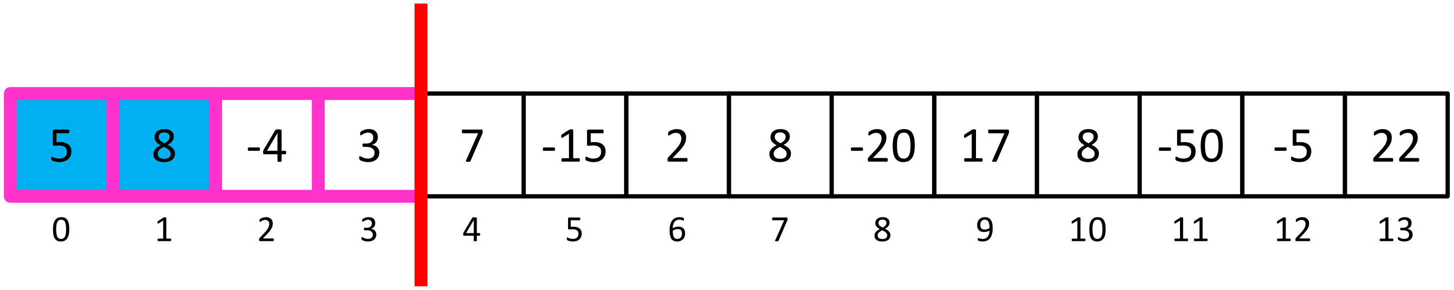
5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Remember two values:

Best So Far
13

Best ending here
9

$\Theta(n)$ Solution



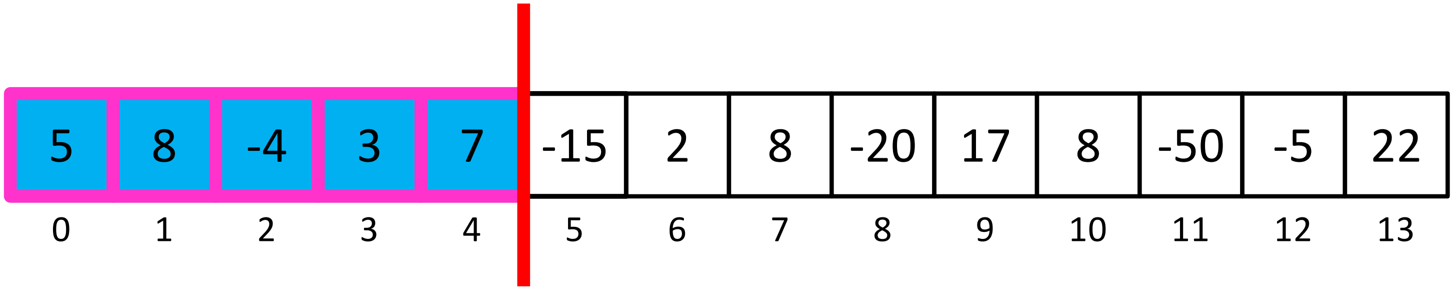
5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Remember two values:

Best So Far
13

Best ending here
12

$\Theta(n)$ Solution



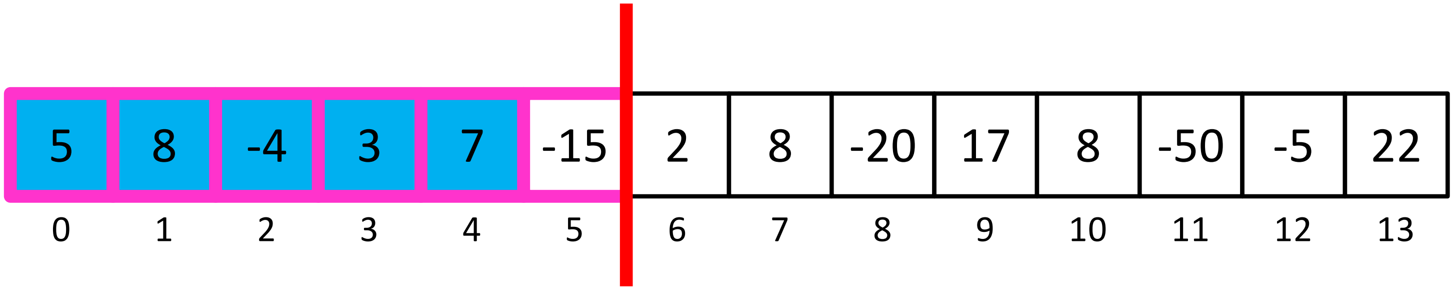
5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Remember two values:

Best So Far
19

Best ending here
19

$\Theta(n)$ Solution



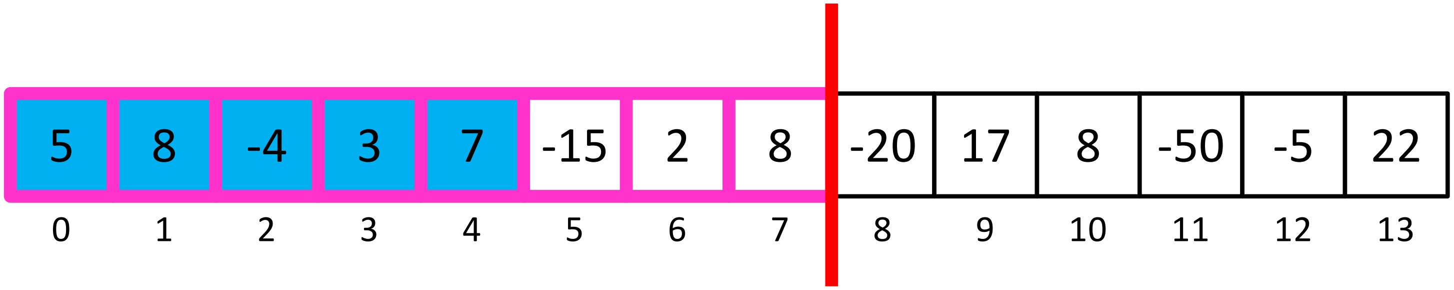
5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Remember two values:

Best So Far
19

Best ending here
4

$\Theta(n)$ Solution

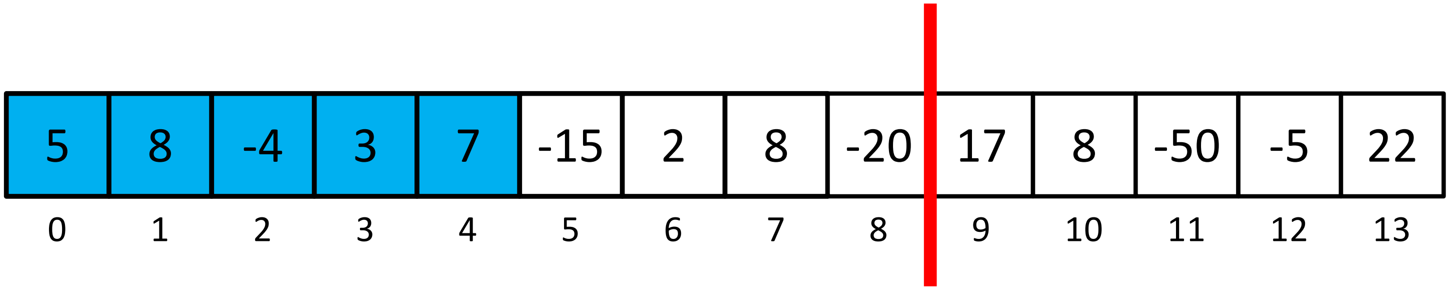


5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

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

19 **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:

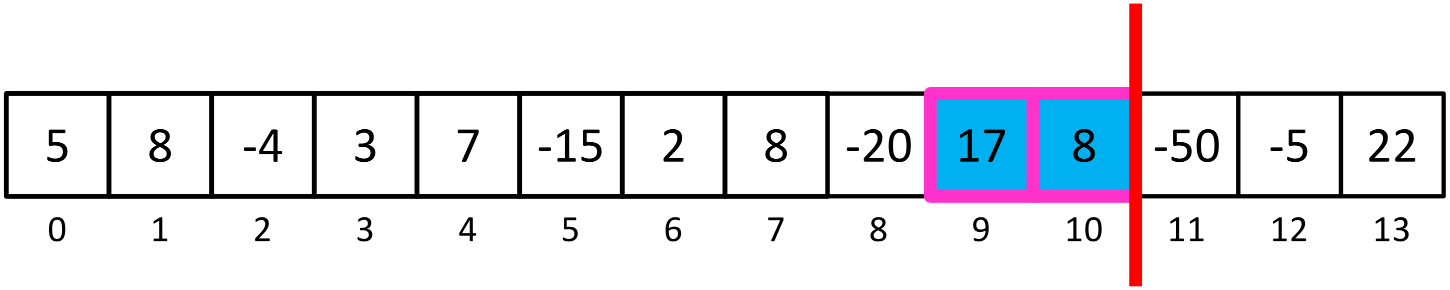
Best So Far

19

Best ending here

17

$\Theta(n)$ Solution



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

Remember two values:

Best So Far
25

Best ending here
25

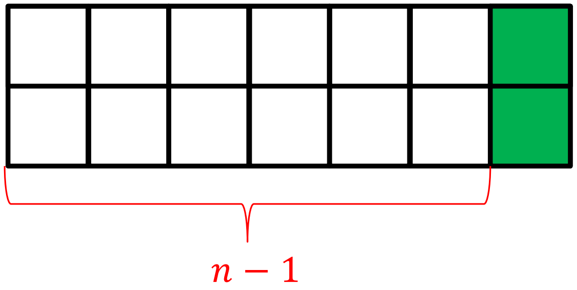
End of Midterm Exam Materials!

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

Back to Tiling

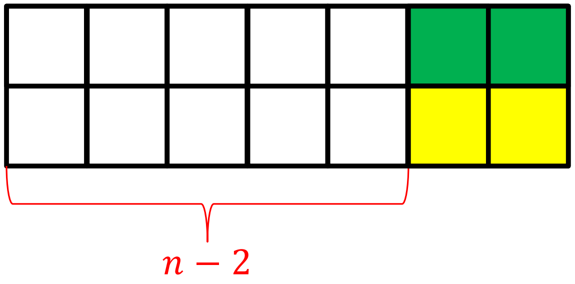
How many ways are there to tile a 2×n board with dominoes?

Two ways to fill the final column:



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

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

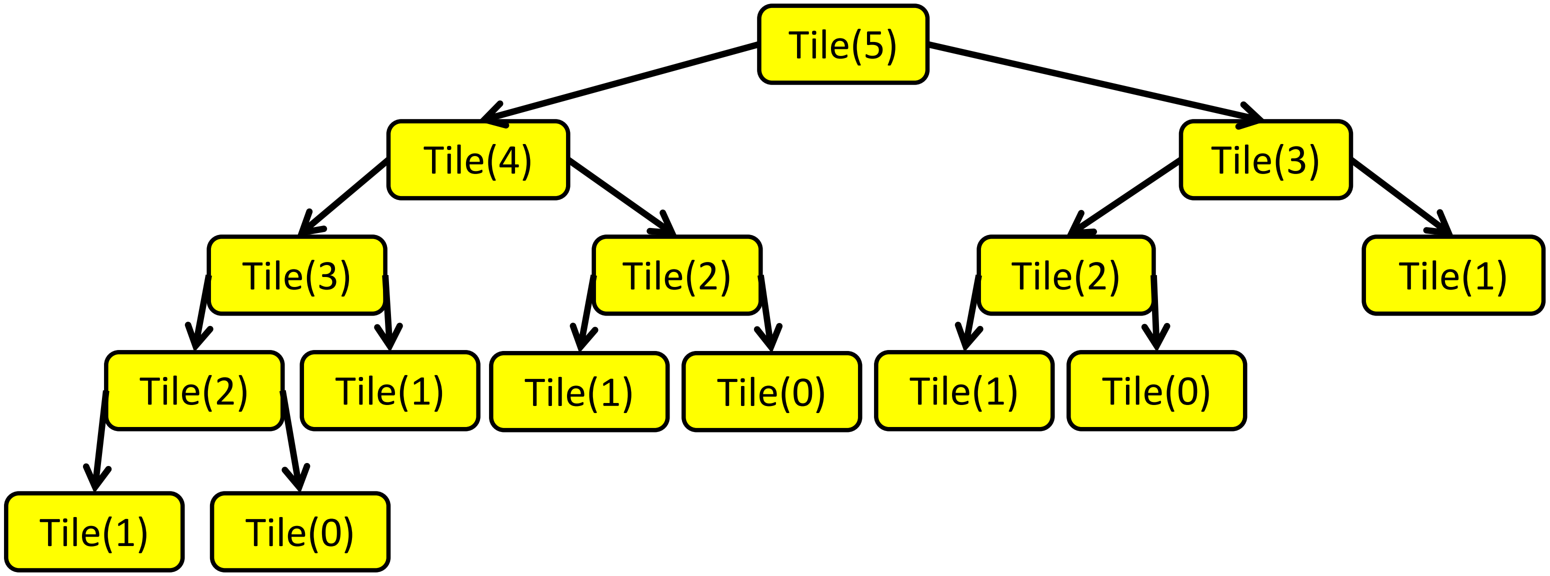


How to compute $Tile(n)$?

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

Problem?

Recursion Tree



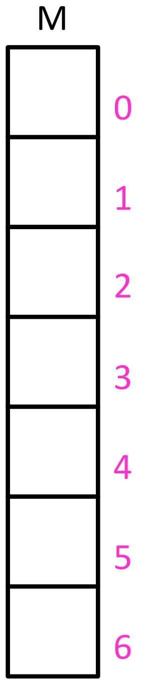
Many redundant calls!

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]
```



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

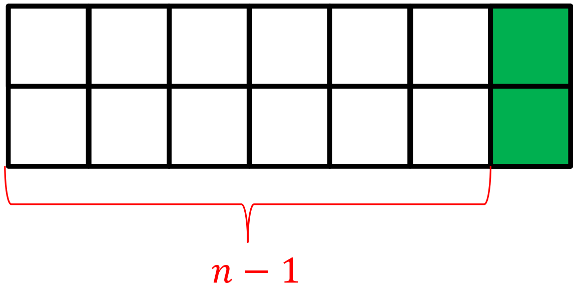
Computing $Tile(n)$ with Memory - “Top Down”

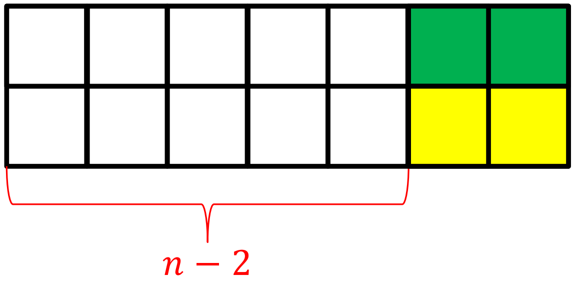
```
Initialize Memory M
Tile(n):
    if n < 2:
        return 1
    if M[n] is filled:
        return M[n]
    M[n] = Tile(n-1)+Tile(n-2)
    return M[n]
```

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 solutions to smaller ones
- Idea:
  1. Identify recursive structure of the problem
     - What is the “last thing” done?





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”

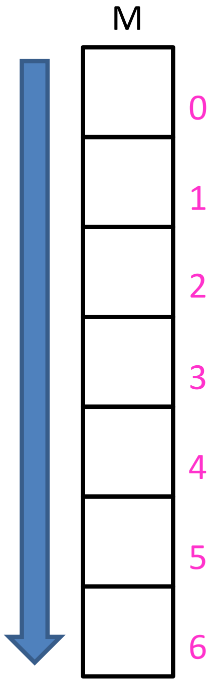
```
Initialize Memory M
Tile(n):
    if n < 2:
        return 1
    if M[n] is filled:
        return M[n]
    M[n] = Tile(n-1)+Tile(n-2)
    return M[n]
```

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
    - Keep in mind that “solution” here means “optimal solution”
- 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

More on Optimal Substructure Property

- Detailed discussion on CLRS p. 379
  - If A is an optimal solution to a problem, then the components of A are optimal solutions to subproblems
- Examples:
  - True for coin-changing
    - Why? Let's discuss
  - True for single-source shortest path (see textbook, p. 381-382)
  - Not true for longest-simple-path (p. 382)
  - True for knapsack

Real World Problems, Real Solutions!

- If 7-year old Tommy bought this at the movies for $1.40
  - Could he sell pieces of it to his young friends and make money?
  - Not if he charges $0.10 per piece
  - Maybe a more complex pricing structure? $0.20 for 1, $0.80 for 7, …

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

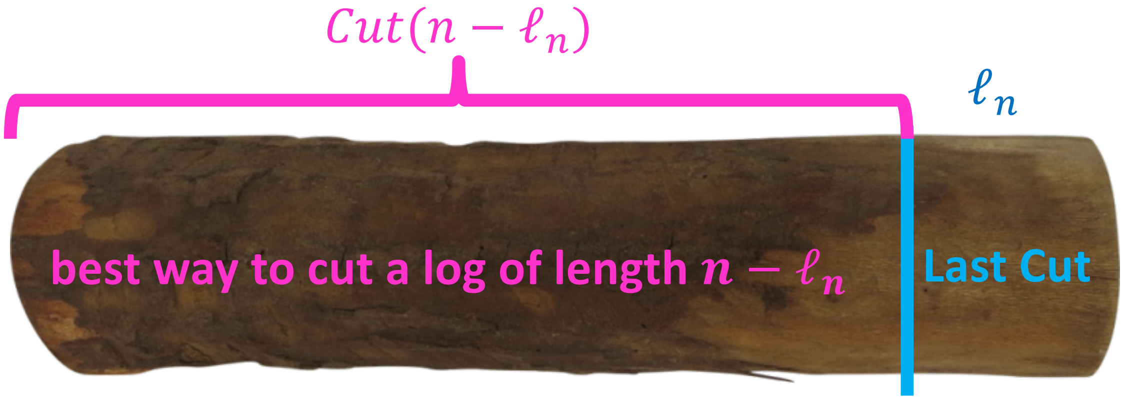
- Requires Optimal Substructure
  - Solution to larger problem contains the solutions to smaller ones
- Idea:
  1. Identify the recursive structure of the problem
     - What is the "last thing" done?
  2. Save the solution to each subproblem in memory
  3. Select a good order for solving subproblems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

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

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. 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

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: $O(n^2)$

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

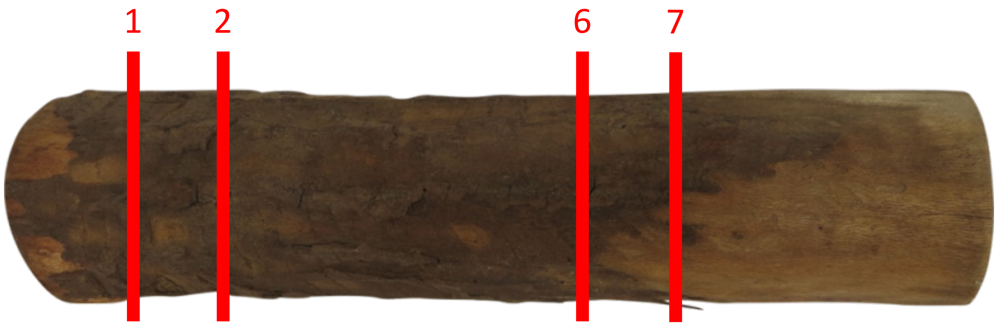
```
Initialize Memory C, Choices
Cut(n):
    C[0] = 0
    for i=1 to n:
        best = 0
        for j = 1 to i:
            if best < C[i-j] + P[j]:
                best = C[i-j] + P[j]
                Choices[i]=j
        C[i] = best
    return C[n]
```

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



Price:	1	5	8	9	10	17	17	20	24	30
Length:	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

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

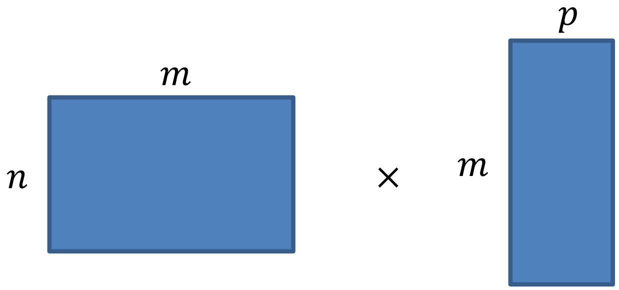
Dynamic Programming

- 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

Mental Stretch

How many arithmetic operations are required to multiply a $n \times m$ Matrix with a $m \times p$ Matrix?

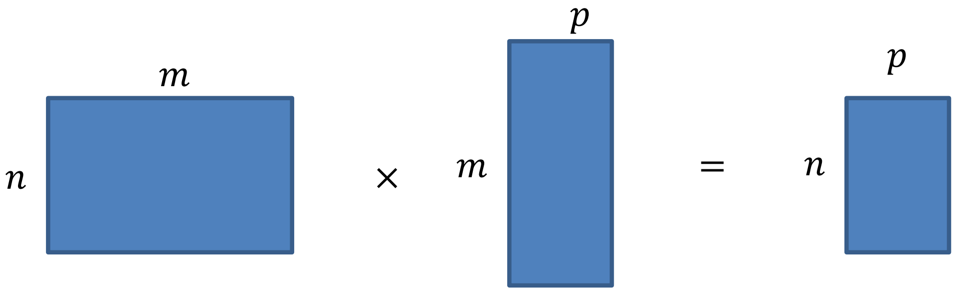
(don’t overthink this)



Mental Stretch

How many arithmetic operations are required to multiply a $n \times m$ Matrix with a $m \times p$ Matrix?

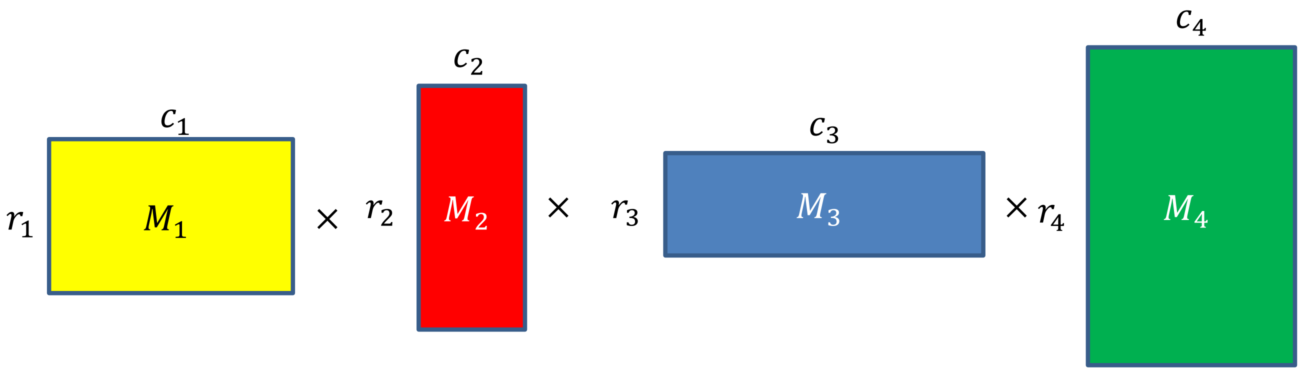
(don't overthink this)



- m multiplications and additions per element
- $n \cdot p$ elements to compute
- Total cost: $m \cdot n \cdot p$

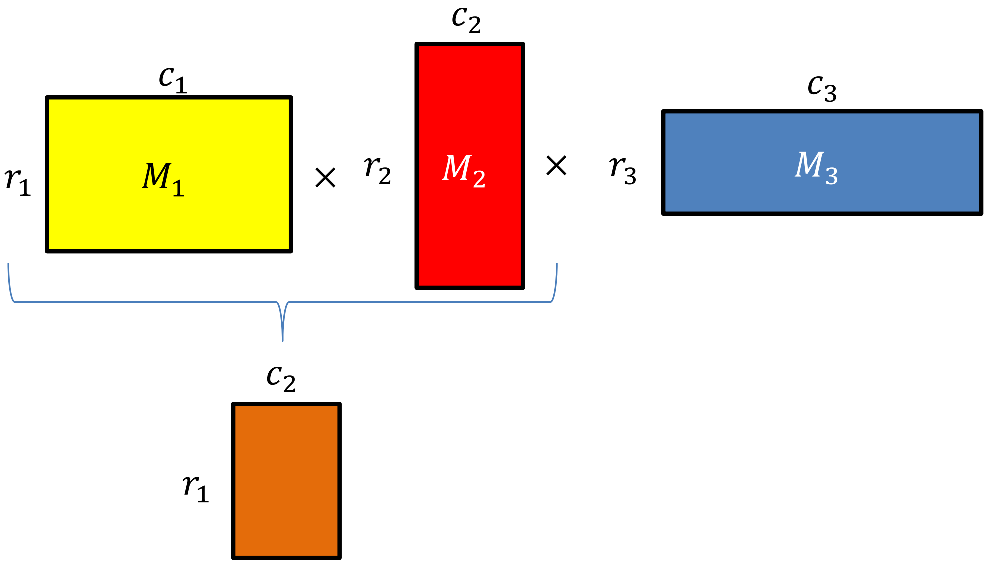
Matrix Chaining

- Given a sequence of Matrices $(M_1, \dots, M_n)$, what is the most efficient way to multiply them?



Order Matters!

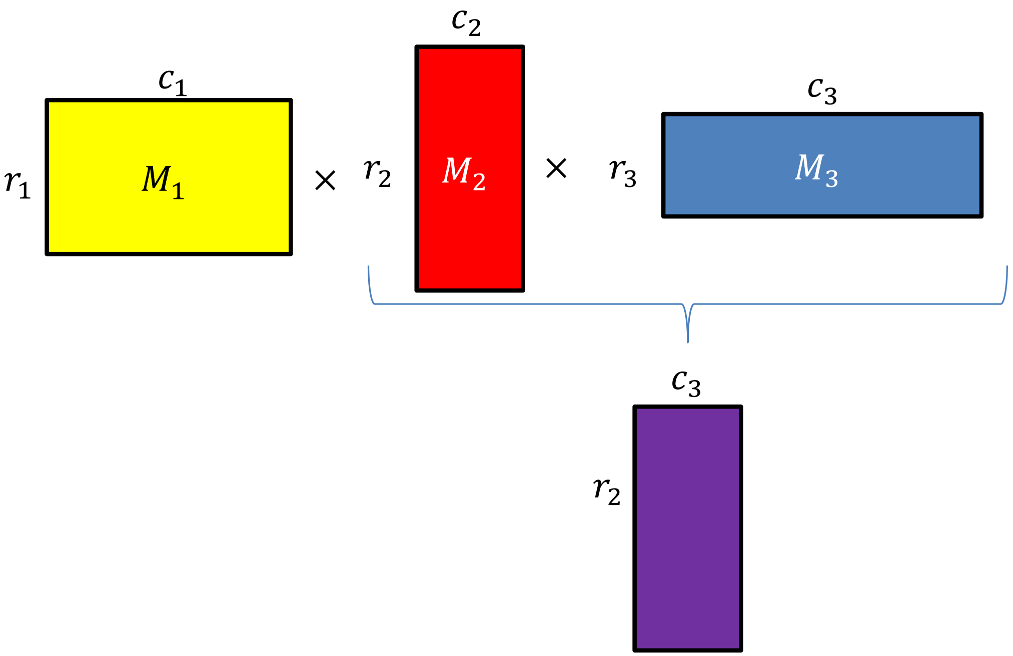
$c_1 = r_2$
$c_2 = r_3$



- $(M_1 \times M_2) \times M_3$
  - uses $(c_1 \cdot r_1 \cdot c_2) + \mathrm{c}_2 \cdot r_1 \cdot c_3$ operations

Order Matters!

$c_1 = r_2$
$c_2 = r_3$



- $M_1 \times (M_2 \times M_3)$
  - uses $\mathrm{c}_1 \cdot \mathrm{r}_1 \cdot c_3 + (\mathrm{c}_2 \cdot r_2 \cdot c_3)$ operations

Order Matters!

$c_1 = r_2$
$c_2 = r_3$

- $(M_1 \times M_2) \times M_3$
  - uses $(c_1 \cdot r_1 \cdot c_2) + \mathrm{c}_2 \cdot r_1 \cdot c_3$ operations
  - $(10 \cdot 7 \cdot 20) + 20 \cdot 7 \cdot 8 = 2520$
- $M_1 \times (M_2 \times M_3)$
  - uses $c_1 \cdot r_1 \cdot c_3 + (\mathrm{c}_2 \cdot r_2 \cdot c_3)$ operations
  - $10 \cdot 7 \cdot 8 + (20 \cdot 10 \cdot 8) = 2160$

$M_1 = 7 \times 10$
$M_2 = 10 \times 20$
$M_3 = 20 \times 8$

$c_1 = 10$
$c_2 = 20$
$c_3 = 8$
$r_1 = 7$
$r_2 = 10$
$r_3 = 20$

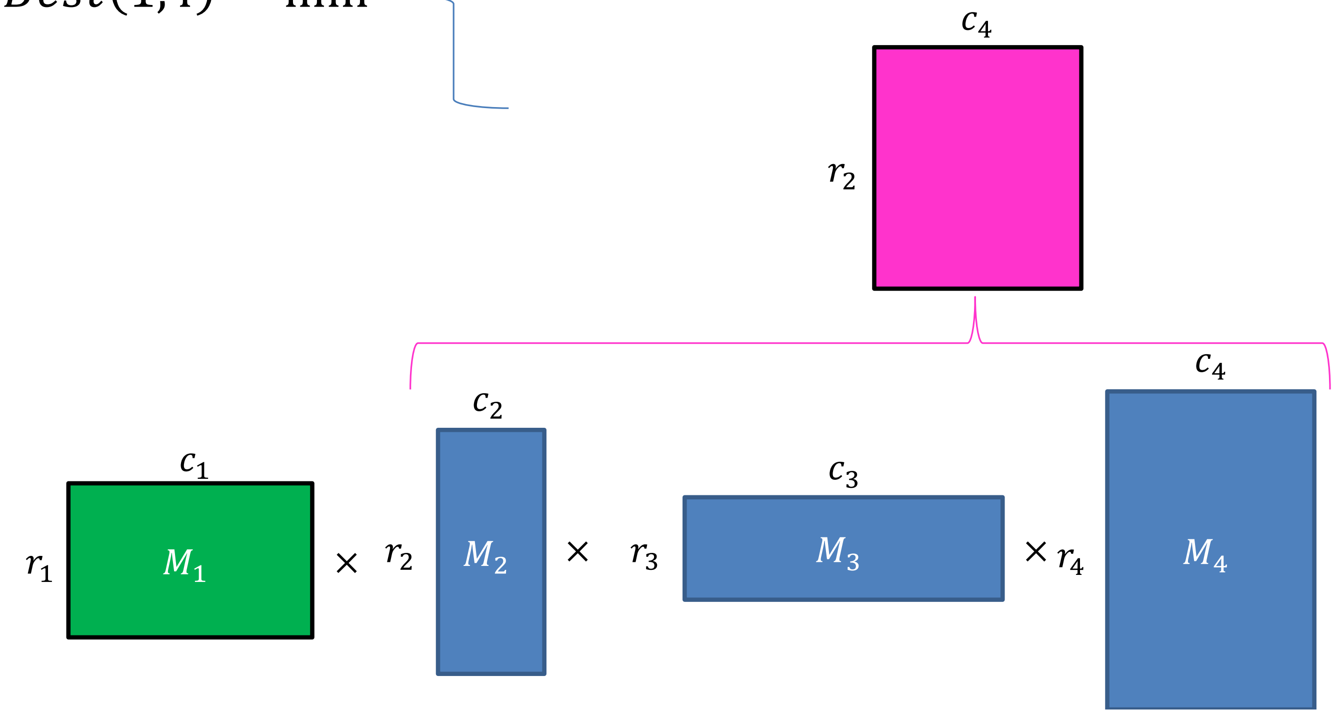
Dynamic Programming

- 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 the Recursive Structure of the Problem

$Best(1, n)$ = cheapest way to multiply together M_1 through M_n

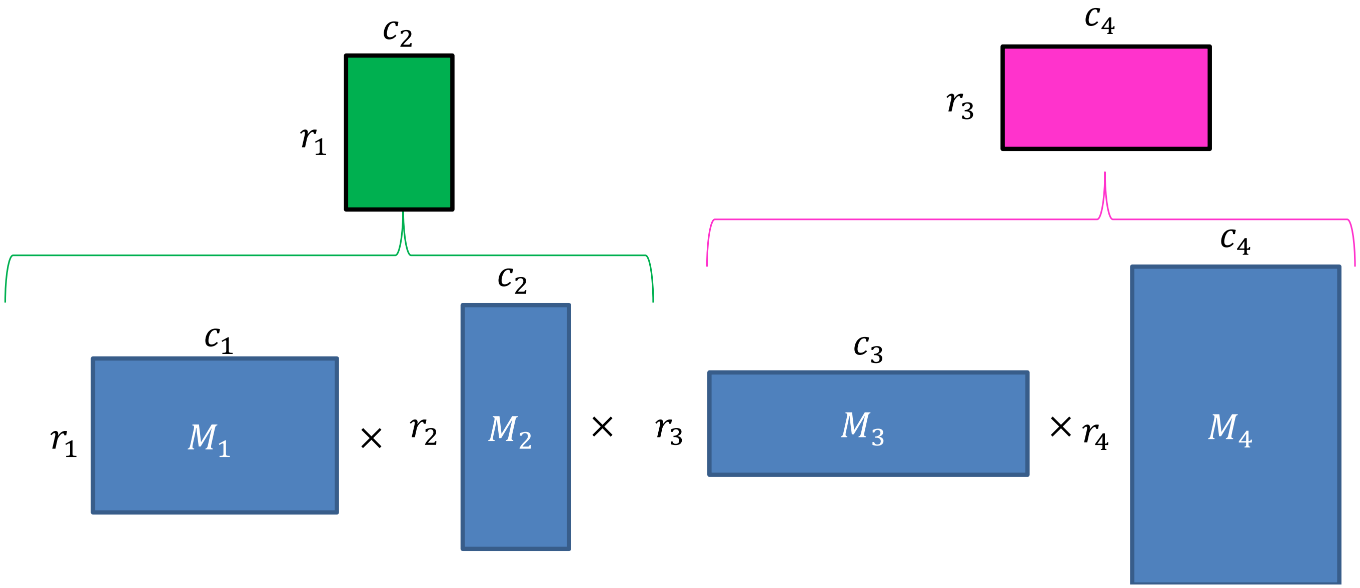
$$Best(1,4) = \min \begin{cases} Best(2,4) + r_1 r_2 c_4 \\ \\ \end{cases}$$



1. Identify the Recursive Structure of the Problem

$Best(1, n)$ = cheapest way to multiply together M_1 through M_n

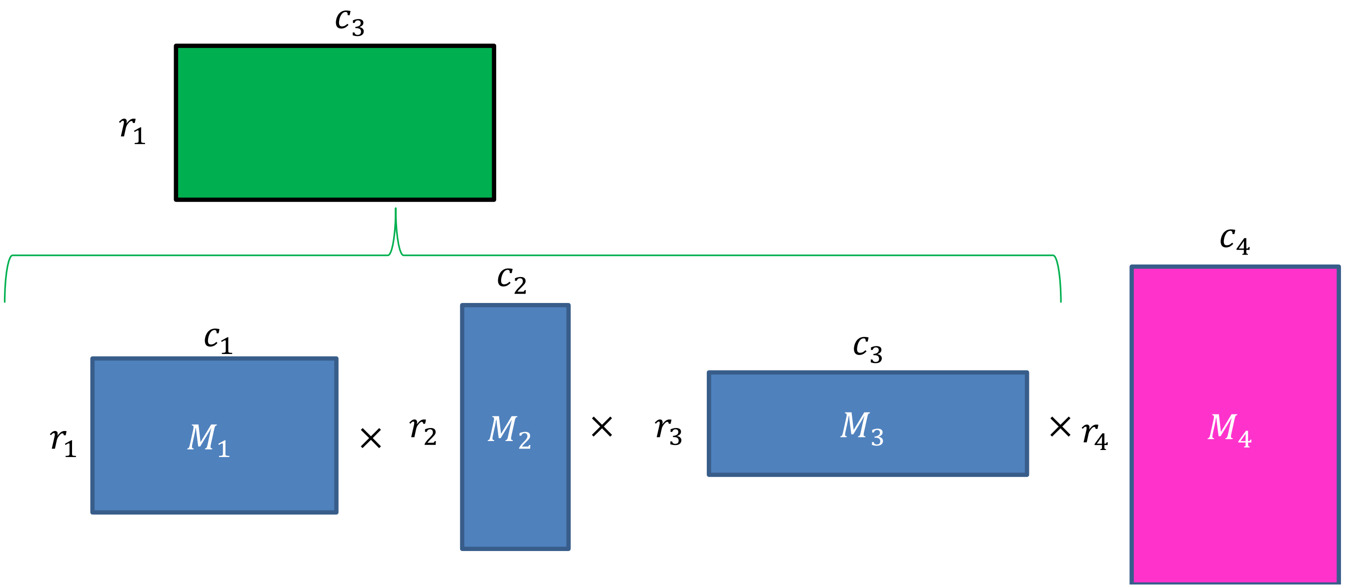
$$Best(1,4) = \min \begin{cases} Best(2,4) + r_1 r_2 c_4 \\ Best(1,2) + Best(3,4) + r_1 r_3 c_4 \\ \\ \end{cases}$$



1. Identify the Recursive Structure of the Problem

$Best(1, n)$ = cheapest way to multiply together M_1 through M_n

$$Best(1,4) = \min \begin{cases} Best(2,4) + r_1 r_2 c_4 \\ Best(1,2) + Best(3,4) + r_1 r_3 c_4 \\ Best(1,3) + r_1 r_4 c_4 \end{cases}$$



1. Identify the Recursive Structure of the Problem

- In general:

$Best(i,j)$ = cheapest way to multiply together M_i through M_j

$$Best(i,j) = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$$

$$Best(i,i) = 0$$

$$Best(1,n) = \min \begin{cases} Best(2,n) + r_1 r_2 c_n \\ Best(1,2) + Best(3,n) + r_1 r_3 c_n \\ Best(1,3) + Best(4,n) + r_1 r_4 c_n \\ Best(1,4) + Best(5,n) + r_1 r_5 c_n \\ \quad \ldots \\ Best(1,n-1) + r_1 r_n c_n \end{cases}$$

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

2. Save Subsolutions in Memory

- In general:

$Best(i,j)$ = cheapest way to multiply together M_i through M_j

$$Best(i,j) = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$$

$$Best(i,i) = 0$$

Read from M[n]
if present

Save to M[n]

$$Best(1,n) = \min \begin{cases} Best(2,n) + r_1 r_2 c_n \\ Best(1,2) + Best(3,n) + r_1 r_3 c_n \\ Best(1,3) + Best(4,n) + r_1 r_4 c_n \\ Best(1,4) + Best(5,n) + r_1 r_5 c_n \\ \quad \dots \\ Best(1,n-1) + r_1 r_n c_n \end{cases}$$

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

- In general:

$Best(i,j)$ = cheapest way to multiply together M_i through M_j

$$Best(i,j) = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$$

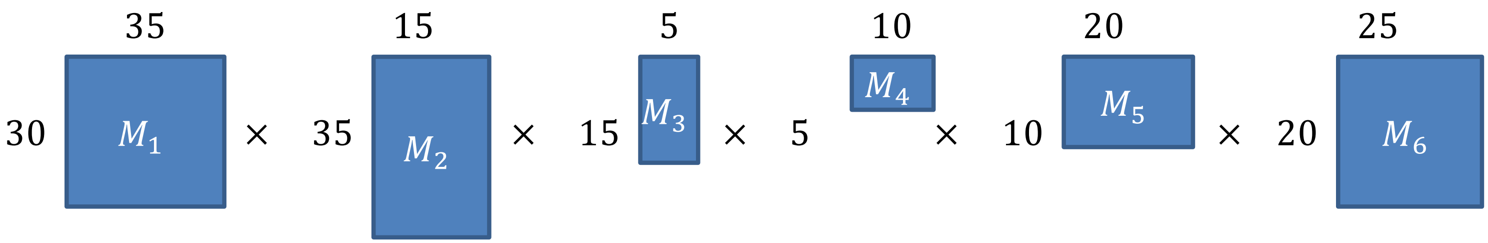
$$Best(i,i) = 0$$

Read from M[n]
if present

Save to M[n]

$$Best(1,n) = \min \begin{cases} Best(2,n) + r_1 r_2 c_n \\ Best(1,2) + Best(3,n) + r_1 r_3 c_n \\ Best(1,3) + Best(4,n) + r_1 r_4 c_n \\ Best(1,4) + Best(5,n) + r_1 r_5 c_n \\ \quad\dots \\ Best(1,n-1) + r_1 r_n c_n \end{cases}$$

3. Select a good order for solving subproblems

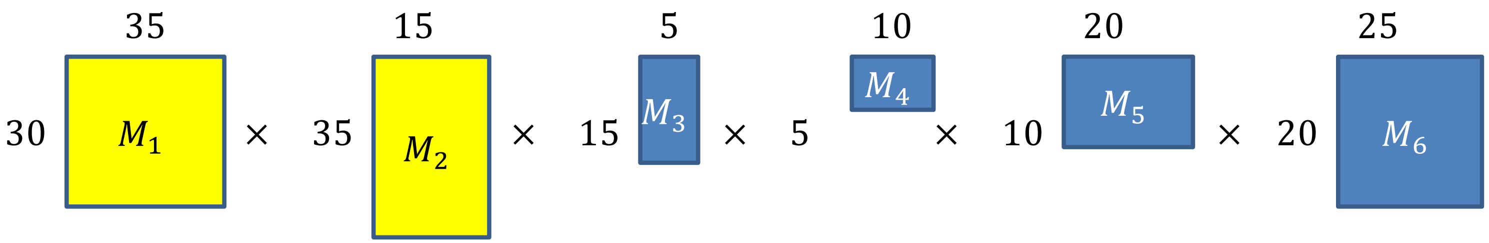


$$Best(i,j) = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$$

$$Best(i,i) = 0$$

$j =$ 1	2	3	4	5	6	$= i$
0						1
	0					2
		0				3
			0			4
				0		5
					0	6

3. Select a good order for solving subproblems



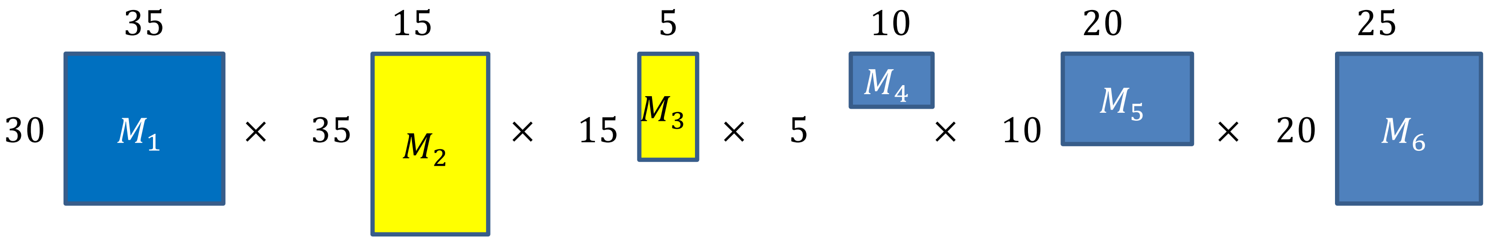
$$Best(i,j) = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$$

$$Best(i,i) = 0$$

$j =$ 1	2	3	4	5	6	$= i$
0	15750					1
	0					2
		0				3
			0			4
				0		5
					0	6

$$Best(1,2) = \min\begin{cases} Best(1,1) + Best(2,2) + r_1 r_2 c_2 \end{cases}$$

3. Select a good order for solving subproblems



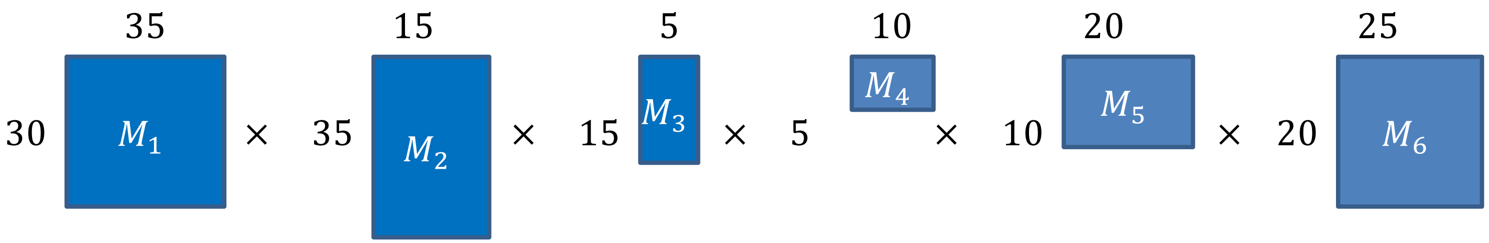
$$Best(i,j) = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$$

$$Best(i,i) = 0$$

$j =$	1	2	3	4	5	6	$= i$
	0	15750					1
		0	2625				2
			0				3
				0			4
					0		5
						0	6

$$Best(2,3) = \min\left\{ Best(2,2) + Best(3,3) + r_2 r_3 c_3 \right.$$

3. Select a good order for solving subproblems

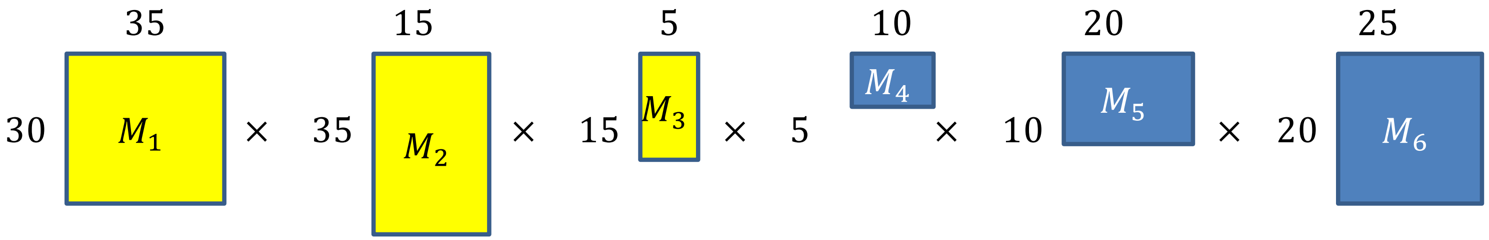


$$Best(i,j) = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$$

$$Best(i,i) = 0$$

$j =$ 1	2	3	4	5	6	$= i$
0	15750					1
	0	2625				2
		0	750			3
			0	1000		4
				0	5000	5
					0	6

3. Select a good order for solving subproblems



$$Best(i,j) = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$$

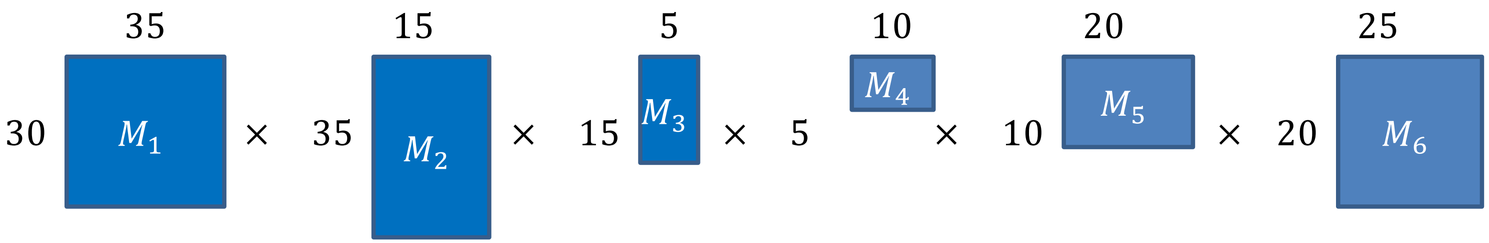
$$Best(i,i) = 0$$

$j =$ 1	2	3	4	5	6	$= i$
0	15750	7875				1
	0	2625				2
		0	750			3
			0	1000		4
				0	5000	5
					0	6

$$r_1 r_2 c_3 = 30 \cdot 35 \cdot 5 = 5250$$
$$r_1 r_3 c_3 = 30 \cdot 15 \cdot 5 = 2250$$

$$Best(1,3) = \min\begin{cases} \overset{0}{Best(1,1)} + \overset{2625}{Best(2,3)} + r_1 r_2 c_3 \\ \underset{15750}{Best(1,2)} + \underset{0}{Best(3,3)} + r_1 r_3 c_3 \end{cases}$$

3. Select a good order for solving subproblems



$$Best(i,j) = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$$

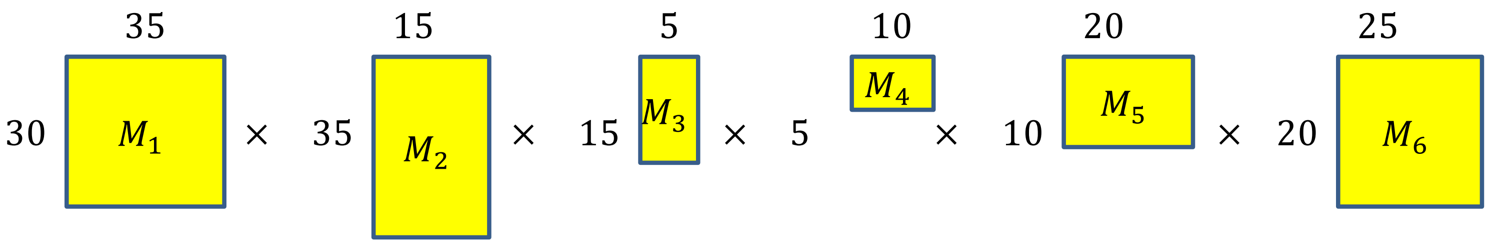
$$Best(i,i) = 0$$

$j =$ 1	2	3	4	5	6	$= i$
0	15750	7875	🟩	🟦	🟪	1
	0	2625	🟨	🟩	🟦	2
		0	750	🟨	🟩	3
			0	1000	🟨	4
				0	5000	5
					0	6

To find $Best(i,j)$: Need all preceding terms of row i and column j

Conclusion: solve in order of diagonal

Matrix Chaining



$$Best(i,j) = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$$

$$Best(i,i) = 0$$

$j =$	1	2	3	4	5	6	$= i$
	0	15750	7875	9375	11875	15125	1
		0	2625	4375	7125	10500	2
			0	750	2500	5375	3
				0	1000	3500	4
					0	5000	5
						0	6

$$Best(1,6) = \min \begin{cases} Best(1,1) + Best(2,6) + r_1 r_2 c_6 \\ Best(1,2) + Best(3,6) + r_1 r_3 c_6 \\ Best(1,3) + Best(4,6) + r_1 r_4 c_6 \\ Best(1,4) + Best(5,6) + r_1 r_5 c_6 \\ Best(1,5) + Best(6,6) + r_1 r_6 c_6 \end{cases}$$

Run Time

1. Initialize $Best[i,i]$ to be all 0s

   $\Theta(n^2)$ cells in the Array

2. Starting at the main diagonal, working to the upper-right, fill in each cell using:
   1. $Best[i,i] = 0$
   2. $Best[i,j] = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$

   $\Theta(n)$ options for each cell

   Each "call" to Best() is a O(1) memory lookup

$\Theta(n^3)$ overall run time

Backtrack to find the best order

“remember” which choice of k was the minimum at each cell

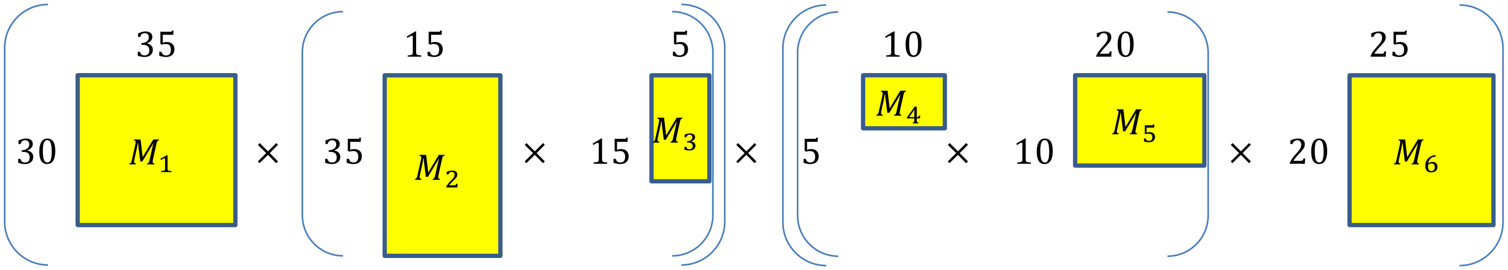
$$Best(i,j) = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$$

$$Best(i,i) = 0$$

$j =$ 1	2	3	4	5	6	$= i$
0	15750	7875 1	9375	11875	15125 3	1
	0	2625	4375	7125	10500	2
		0	750	2500	5375	3
			0	1000	3500 5	4
				0	5000	5
					0	6

$$Best(1,6) = \min \begin{cases} Best(1,1) + Best(2,6) + r_1 r_2 c_6 \\ Best(1,2) + Best(3,6) + r_1 r_3 c_6 \\ \boxed{Best(1,3) + Best(4,6) + r_1 r_4 c_6} \\ Best(1,4) + Best(5,6) + r_1 r_5 c_6 \\ Best(1,5) + Best(6,6) + r_1 r_6 c_6 \end{cases}$$

Matrix Chaining



$$Best(i,j) = \min_{k=i}^{j-1}\Big(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\Big)$$

$$Best(i,i) = 0$$

j =	1	2	3	4	5	6	= i
	0	15750	7875 1	9375	11875	15125 3	1
		0	2625	4375	7125	10500	2
			0	750	2500	5375	3
				0	1000	3500 5	4
					0	5000	5
						0	6

$$Best(1,6) = \min \begin{cases} Best(1,1) + Best(2,6) + r_1 r_2 c_6 \\ Best(1,2) + Best(3,6) + r_1 r_3 c_6 \\ \boxed{Best(1,3) + Best(4,6) + r_1 r_4 c_6} \\ Best(1,4) + Best(5,6) + r_1 r_5 c_6 \\ Best(1,5) + Best(6,6) + r_1 r_6 c_6 \end{cases}$$

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

Movie Time!

In Season 9 Episode 7 “The Slicer” of the hit 90s TV show *Seinfeld*, George discovers that, years prior, he had a heated argument with his new boss, Mr. Kruger. This argument ended in George throwing Mr. Kruger’s boombox into the ocean. How did George make this discovery?

https://www.youtube.com/watch?v=pSB3HdmLcY4

Seam Carving

- Method for image resizing that doesn’t scale/crop the image

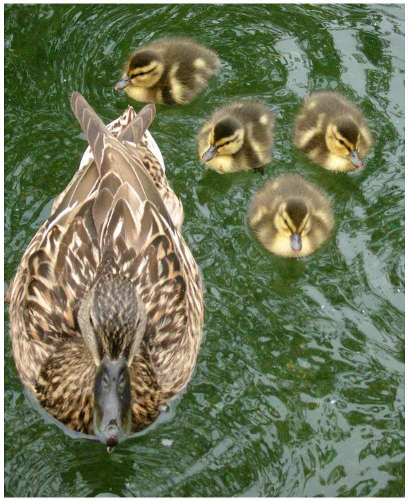
Seam Carving

- Method for image resizing that doesn’t scale/crop the image

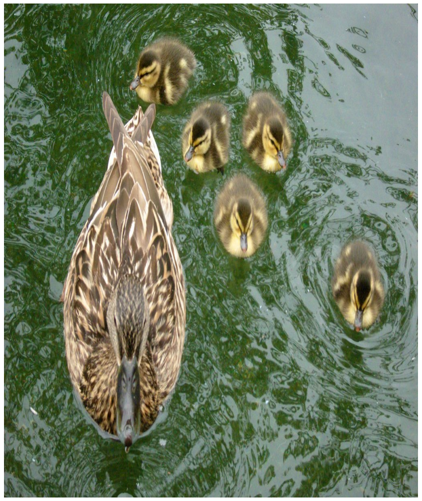
Seam Carving

- Method for image resizing that doesn’t scale/crop the image

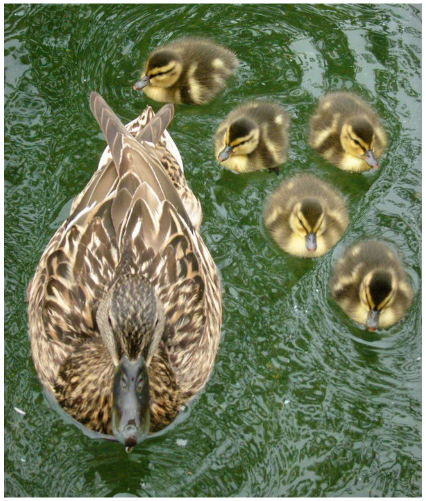
Cropped



Scaled

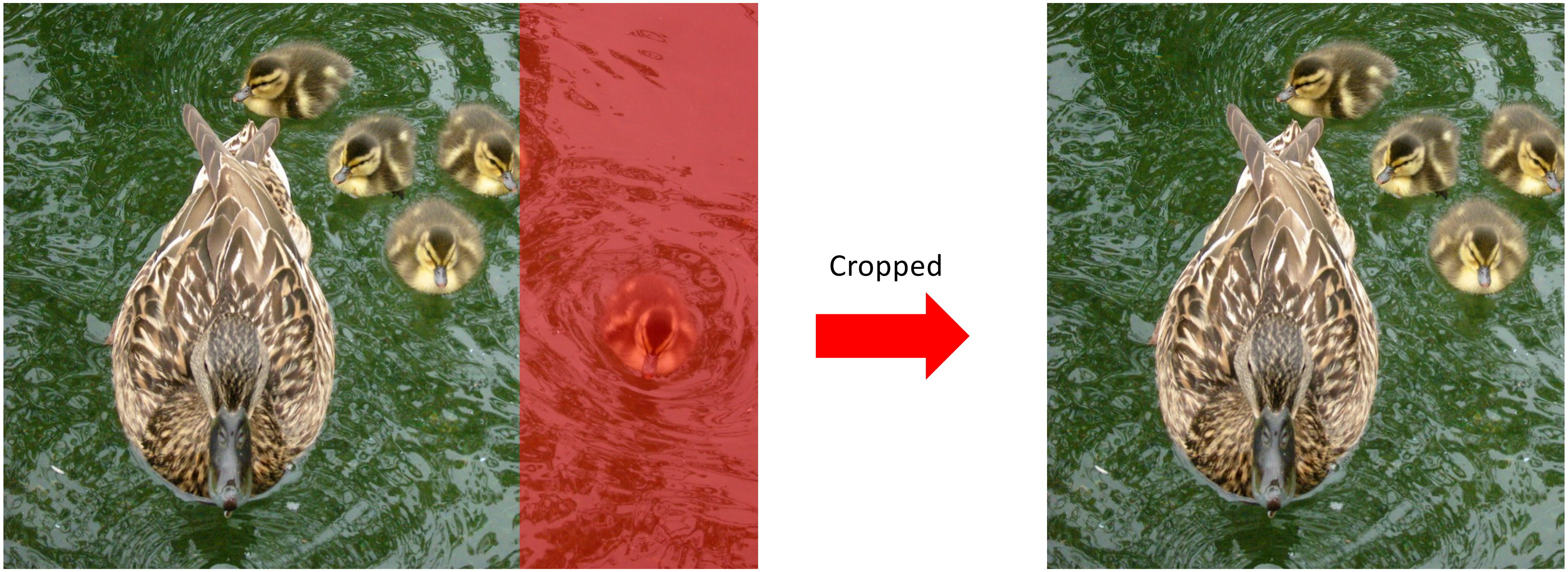


Carved



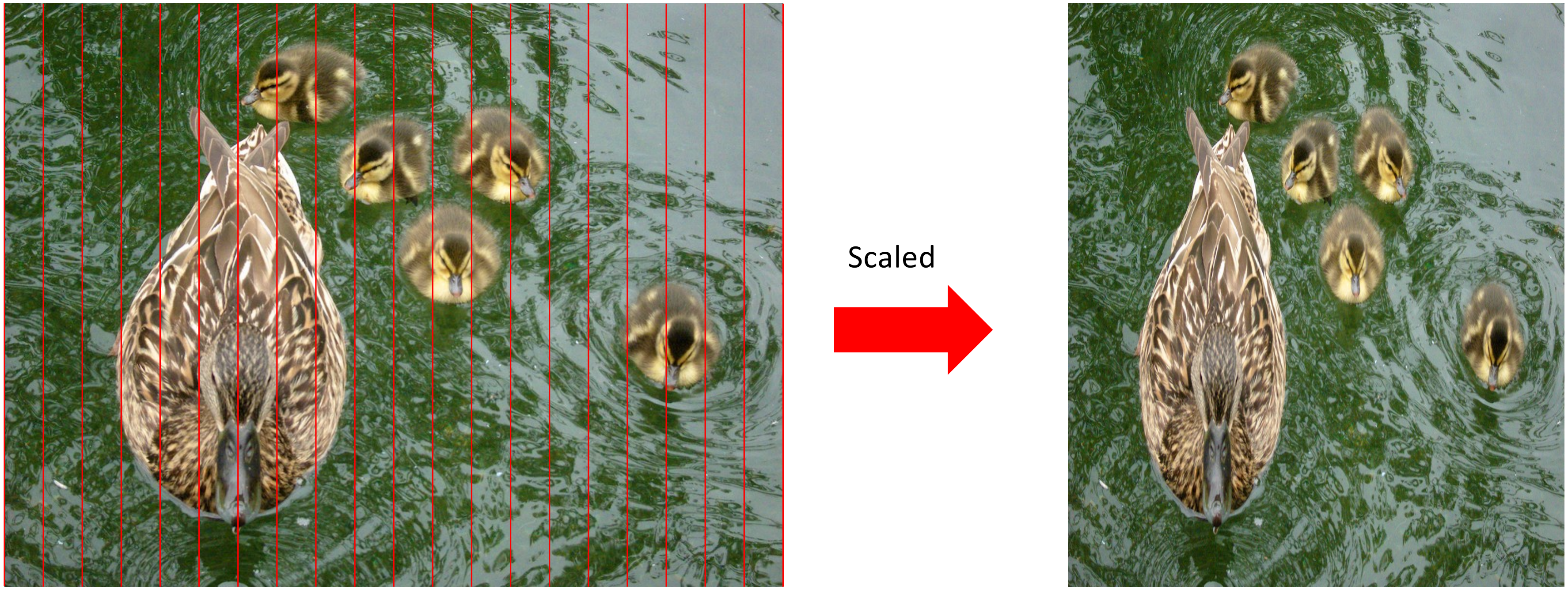
Cropping

- Removes a “block” of pixels



Scaling

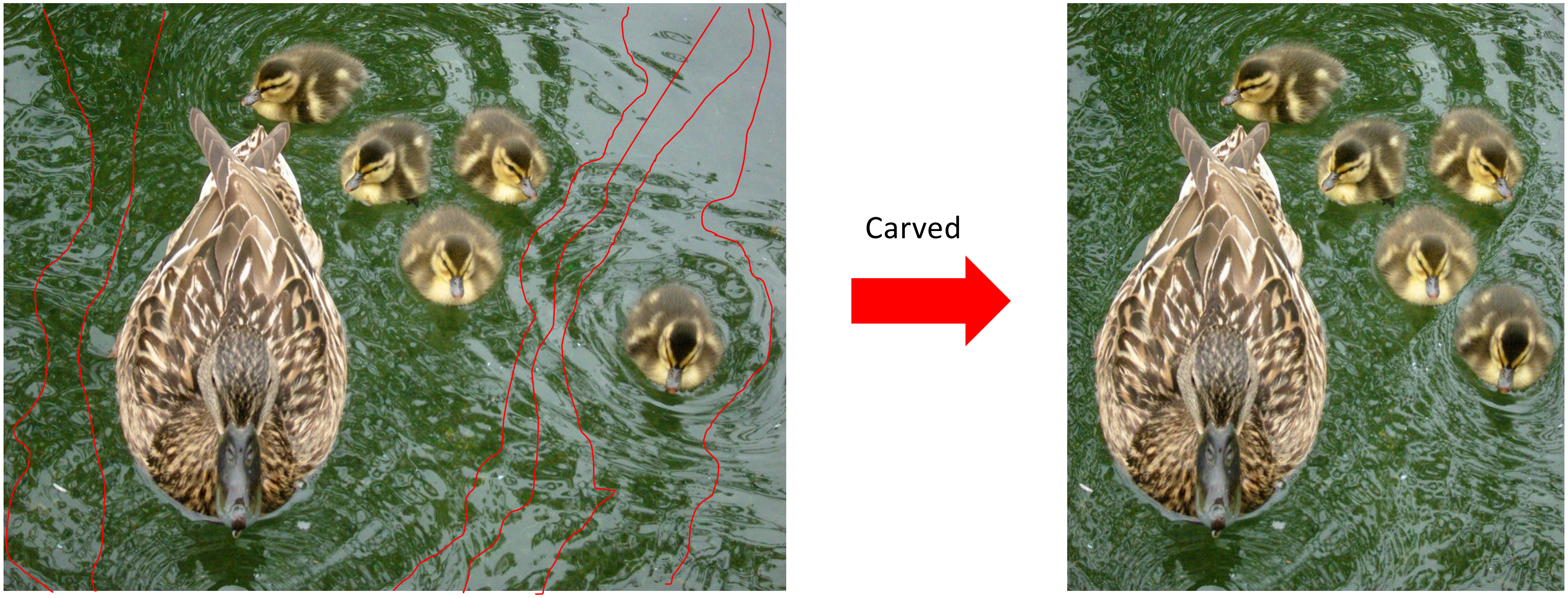
- Removes “stripes” of pixels



Scaled

Seam Carving

- Removes “least energy seam” of pixels
- http://rsizr.com/



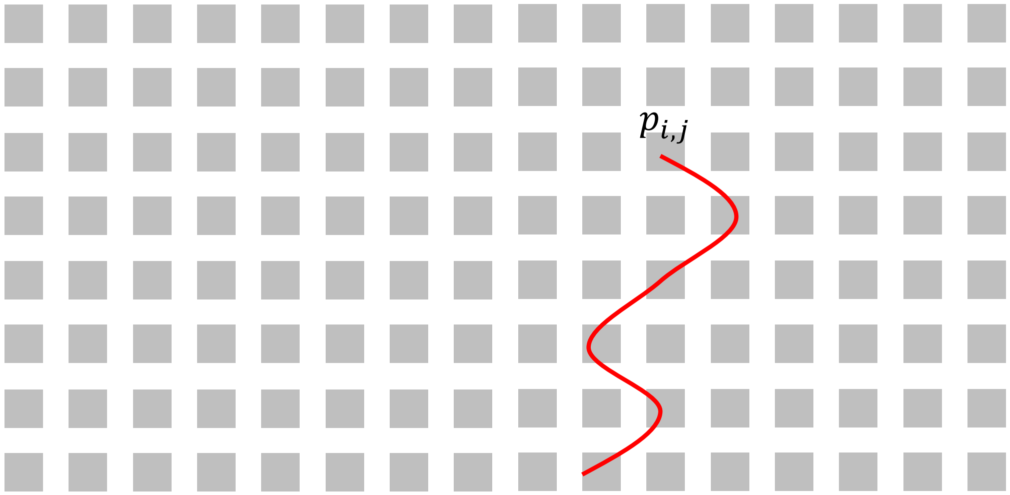
Seattle Skyline

Energy of a Seam

- Sum of the energies of each pixel
  - $e(p) =$ energy of pixel p
- Many choices
  - E.g.: change of gradient (how much the color of this pixel differs from its neighbors)
  - Particular choice doesn't matter, we use it as a "black box"

Identify Recursive Structure

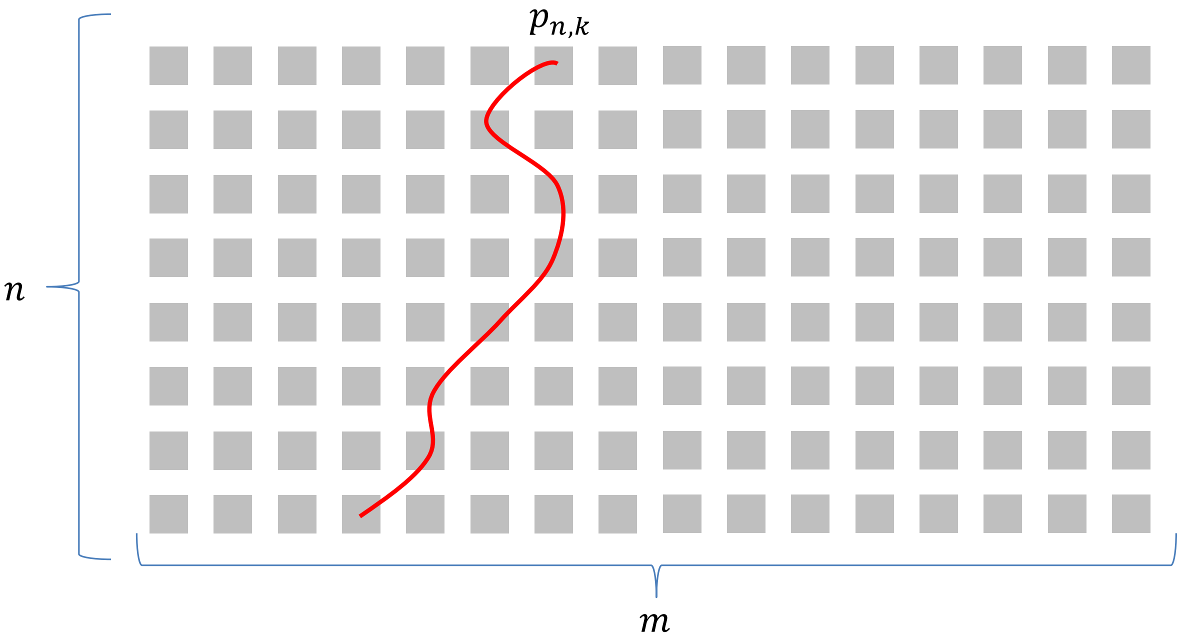
Let $S(i,j)$ = least energy seam from the bottom of the image up to pixel $p_{i,j}$



Finding the Least Energy Seam

Want the least energy seam going from bottom to top, so delete:

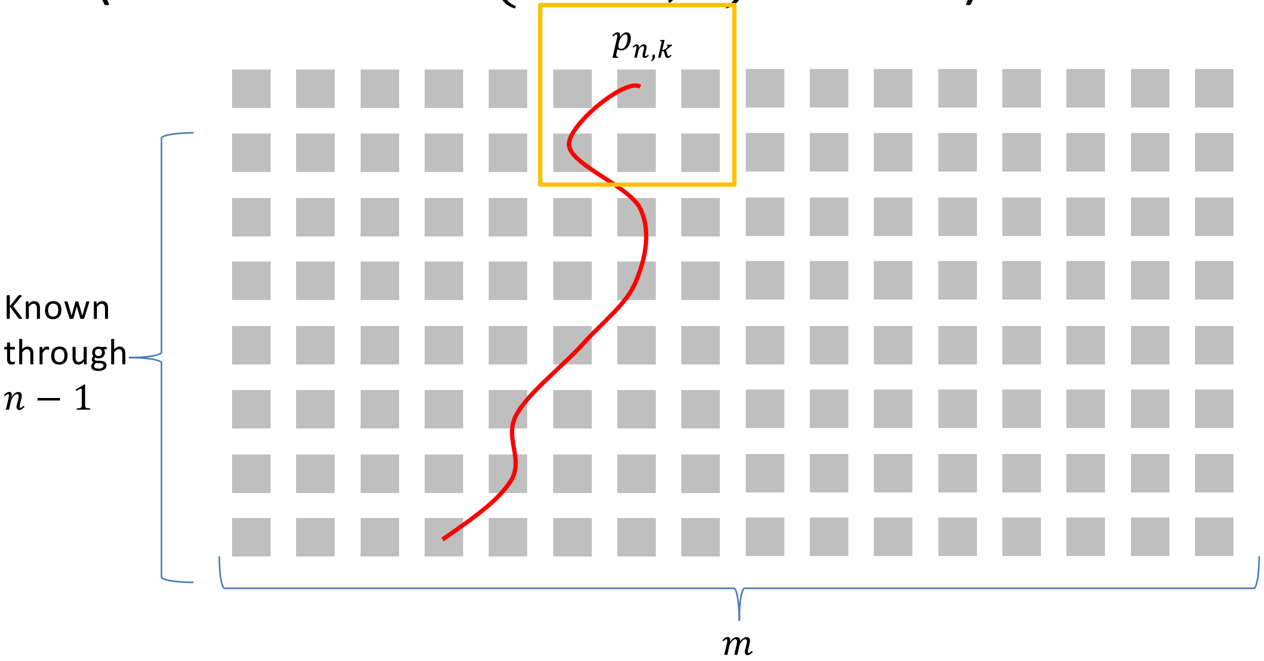
$$\min_{k=1}^{m}(S(n,k))$$



Computing $S(n,k)$

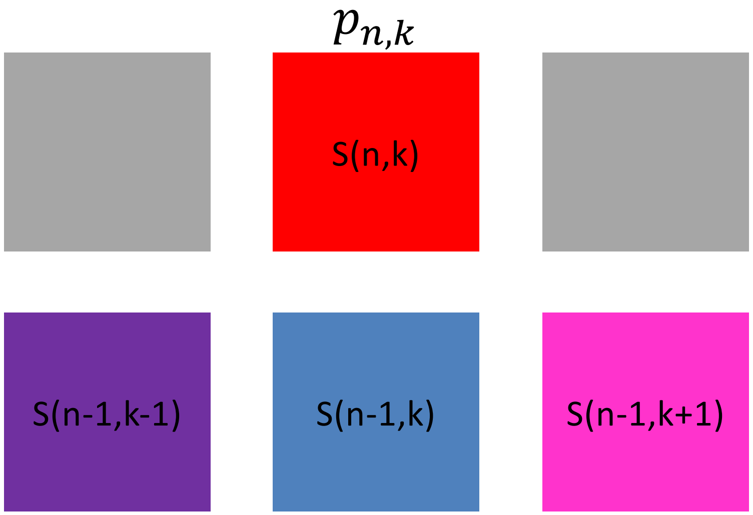
Assume we know the least energy seams for all of row $n-1$

(i.e. we know $S(n-1,\ell)$ for all ℓ)



Computing $S(n,k)$

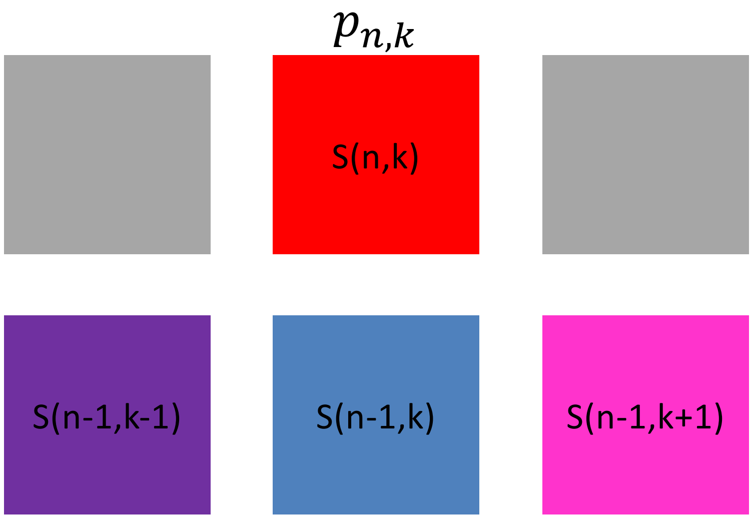
Assume we know the least energy seams for all of row $n-1$ (i.e. we know $S(n-1,\ell)$ for all ℓ)



Computing $S(n,k)$

Assume we know the least energy seams for all of row $n-1$ (i.e. we know $S(n-1,\ell)$ for all ℓ)

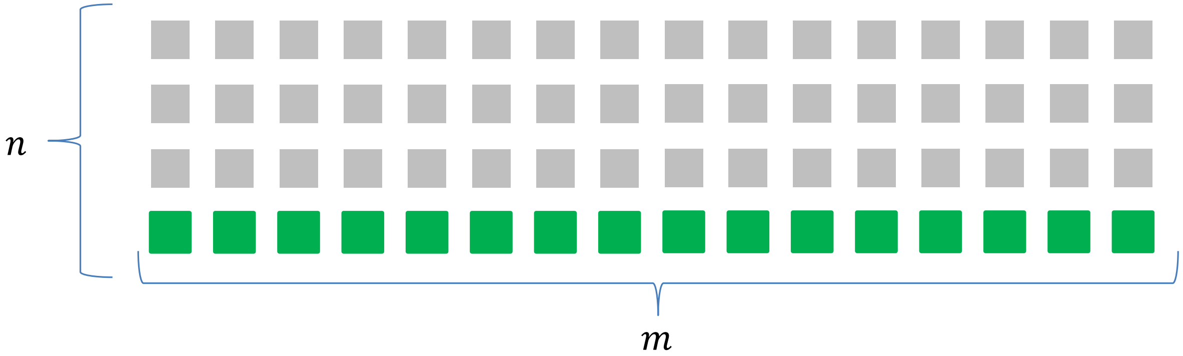
$$S(n,k) = min\begin{cases} S(n-1,k-1) + e(p_{n,k}) \\ S(n-1,k) + e(p_{n,k}) \\ S(n-1,k+1) + e(p_{n,k}) \end{cases}$$



Bring It All Together

Start from bottom of image (row 1), solve up to top

Initialize $S(1,k) = e(p_{1,k})$ for each pixel in row 1



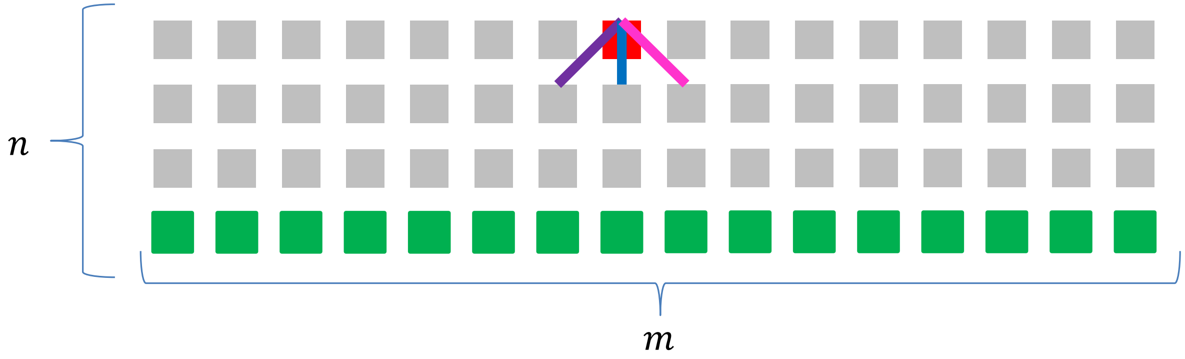
Energy of the seam initialized to the energy of that pixel

Bring It All Together

Start from bottom of image (row 1), solve up to top

Initialize $S(1,k) = e(p_{1,k})$ for each pixel $p_{1,k}$

For $i > 2$ find $S(i,k) = \min \begin{cases} S(n-1,k-1) + e(p_{n,k}) \\ S(n-1,k) + e(p_{n,k}) \\ S(n-1,k+1) + e(p_{n,k}) \end{cases}$



Energy of the seam initialized to the energy of that pixel

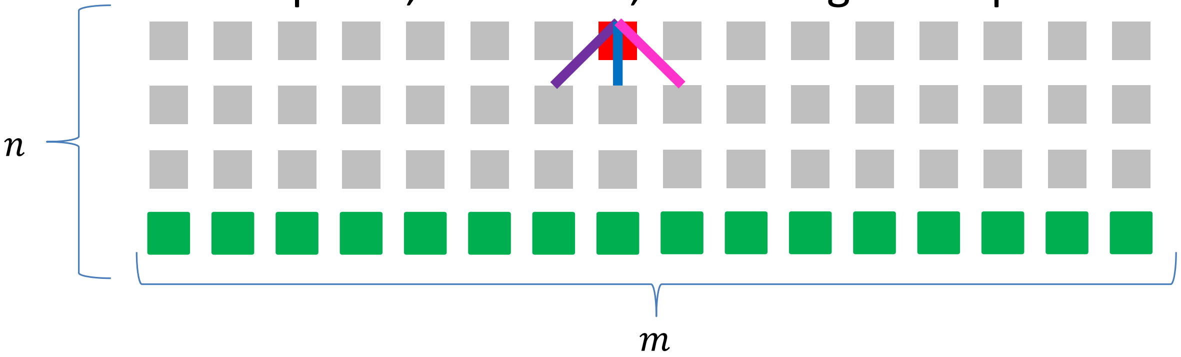
Bring It All Together

Start from bottom of image (row 1), solve up to top

Initialize $S(1,k) = e(p_{1,k})$ for each pixel $p_{1,k}$

For $i > 2$ find $S(i,k) = \min \begin{cases} S(n-1,k-1) + e(p_{n,k}) \\ S(n-1,k) + e(p_{n,k}) \\ S(n-1,k+1) + e(p_{n,k}) \end{cases}$

Pick smallest from top row, backtrack, removing those pixels



Energy of the seam initialized to the energy of that pixel

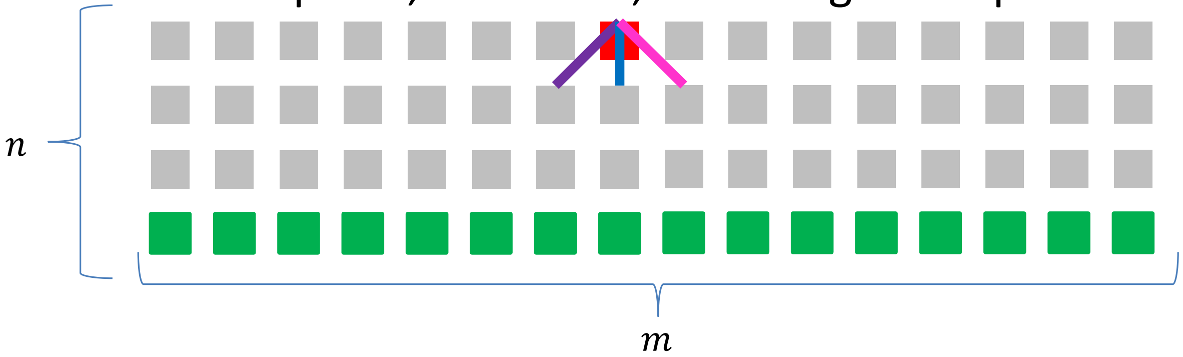
Run Time?

Start from bottom of image (row 1), solve up to top

Initialize $S(1,k) = e(p_{1,k})$ for each pixel $p_{1,k}$ $\qquad \Theta(m)$

$$\text{For } i \geq 2 \text{ find } S(i,k) = \min \begin{cases} S(n-1,k-1) + e(p_{i,k}) \\ S(n-1,k) + e(p_{i,k}) \\ S(n-1,k+1) + e(p_{i,k}) \end{cases} \qquad \Theta(n \cdot m)$$

Pick smallest from top row, backtrack, removing those pixels $\qquad \Theta(n+m)$



Energy of the seam initialized to the energy of that pixel

Repeated Seam Removal

Only need to update pixels dependent on the removed seam

$2n$ pixels change

$\Theta(2n)$ time to update pixels

$\Theta(n+m)$ time to find min+backtrack