CS 4102: Algorithms – Unit C

Dynamic Programming

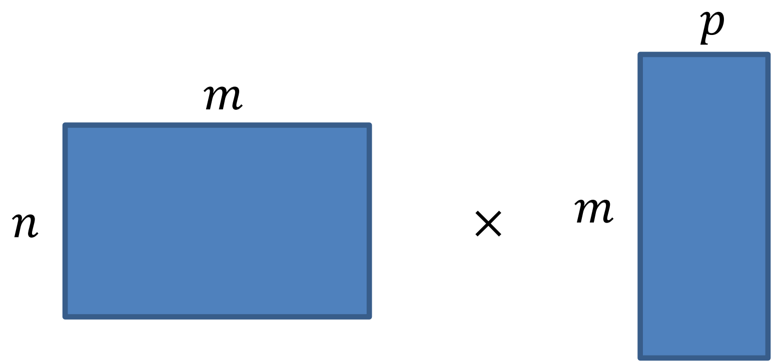
Co-instructors: Robbie Hott and Tom Horton

Spring 2022

Warm Up

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

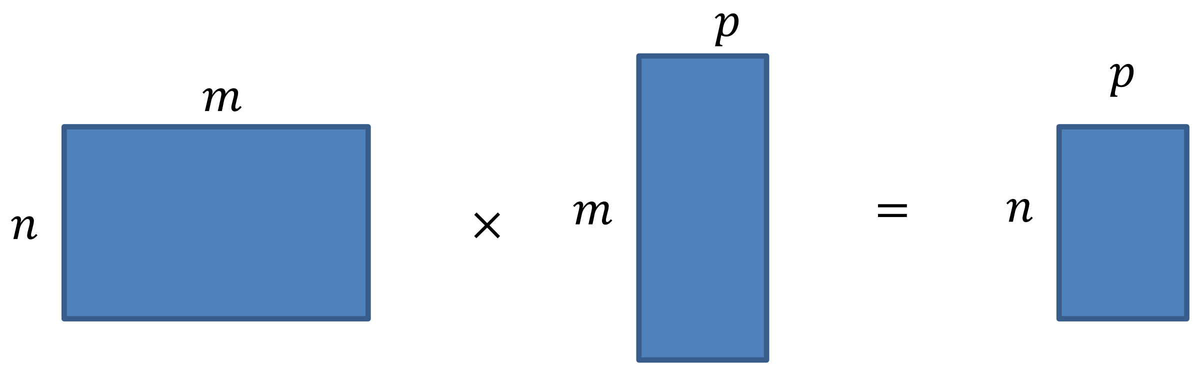
(don't overthink this)



Warm Up

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 $m-1$ additions per element
- $n \cdot p$ elements to compute
- Total cost: $O(m \cdot n \cdot p)$

Today's Keywords

- Dynamic Programming
- Matrix Chaining

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)

Announcements

- Updated Deadlines for Unit B
  - Exam rescheduled for March 29
  - Encouraged to submit early, Unit C assignments are coming soon

Dynamic Programming

- Requires Optimal Substructure
  – Solution to larger problem contains the solutions to smaller ones
  – Or: If S is an optimal solution to a problem, then the components of S are optimal solutions to sub-problems
- 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

Generic Top-Down Dynamic Programming Soln

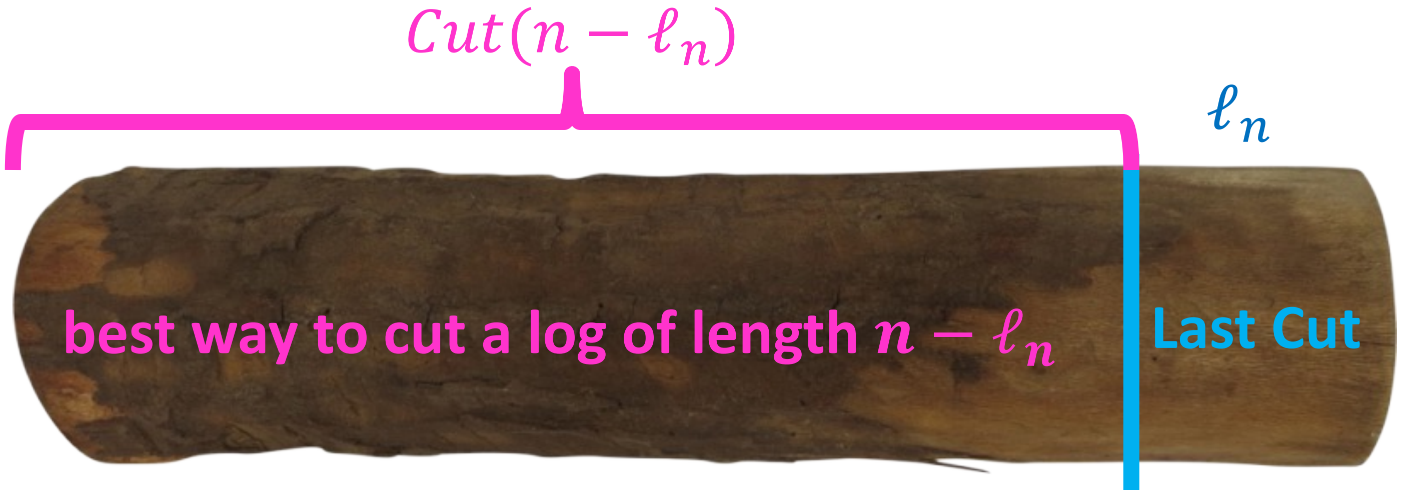
```
mem = {}
def myDPalgo(problem):
        if mem[problem] not blank:
                return mem[problem]
        if baseCase(problem):
                solution = solve(problem)
                mem[problem] = solution
                return solution
        for subproblem of problem:
              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

Log Cutting Recursive Structure

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

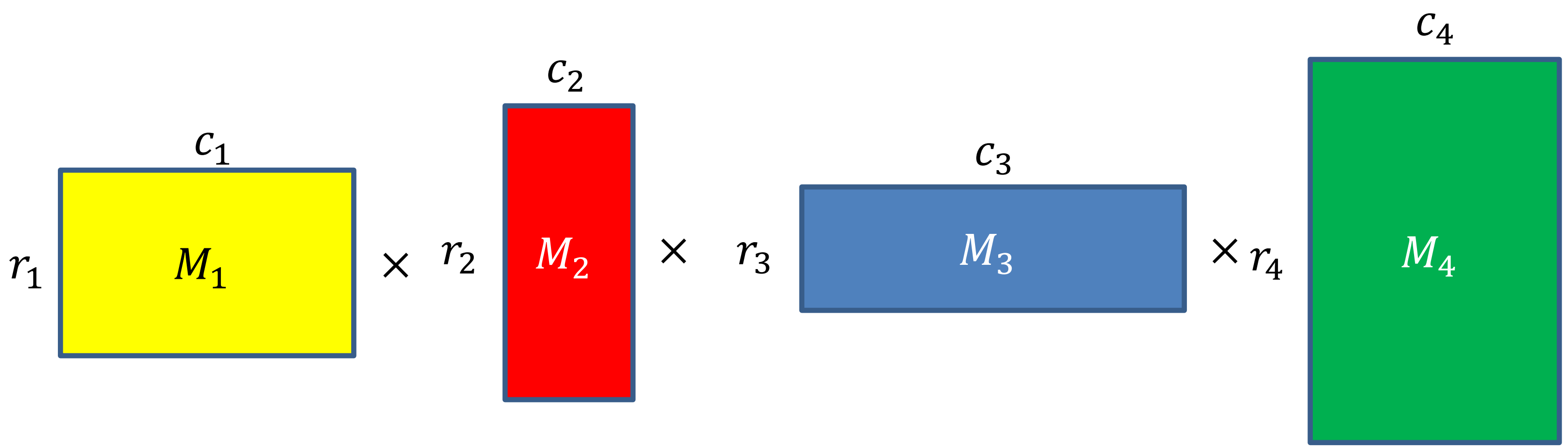
$Cut(n)$ = value of best way to cut a log of length n

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



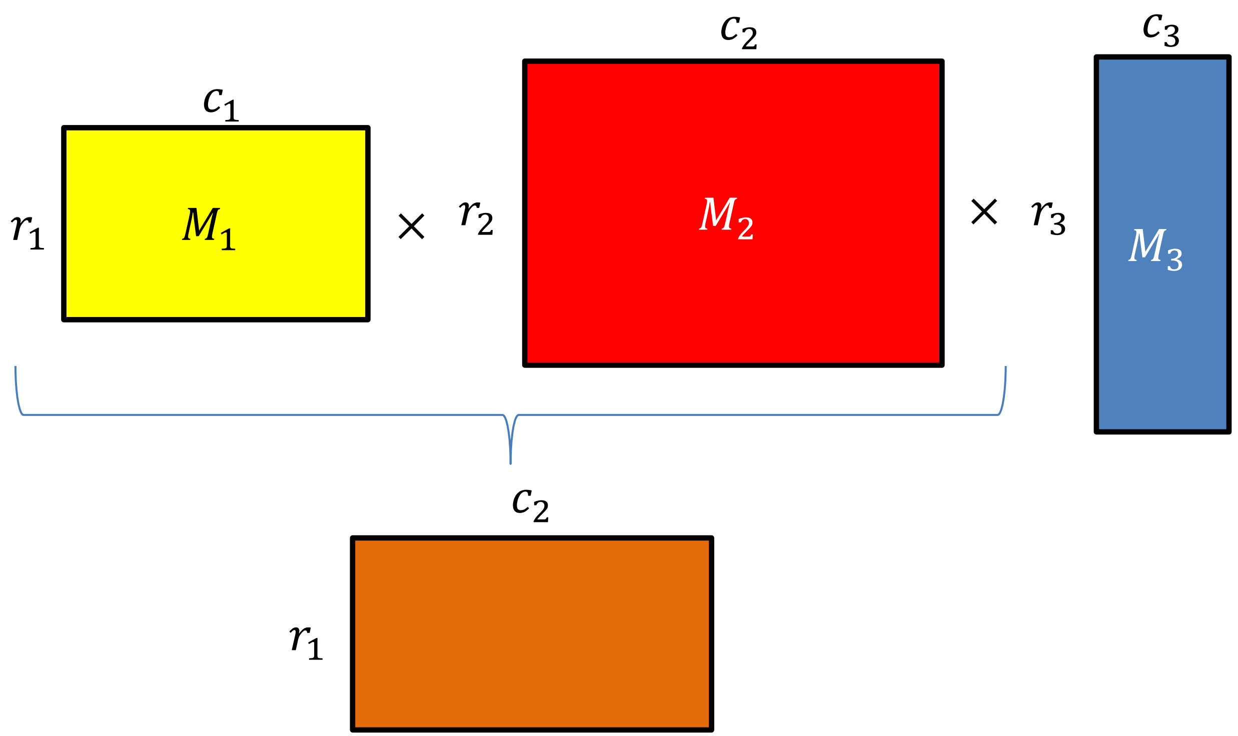
Matrix Chaining

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



Order Matters!

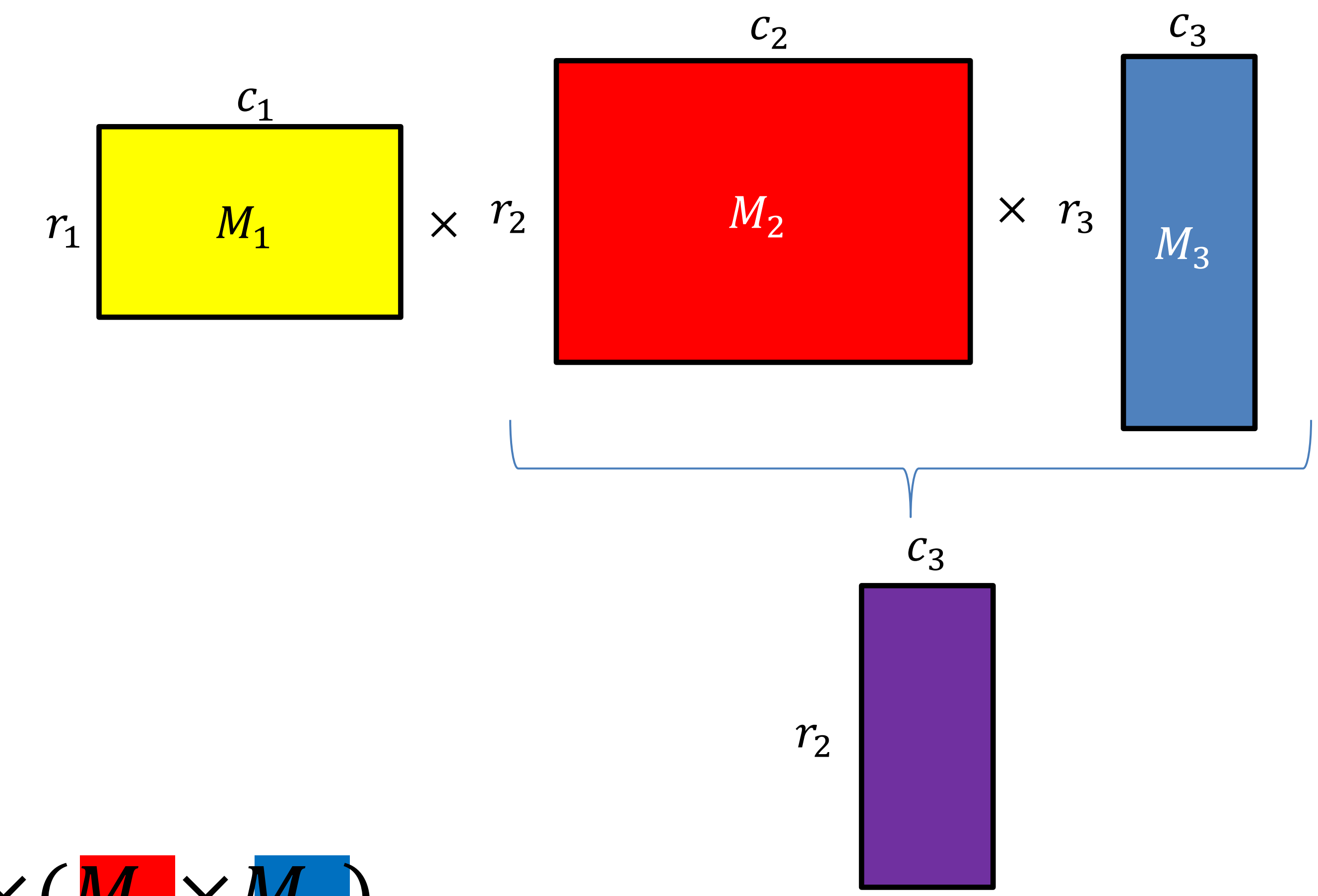
$c_1 = r_2$
$c_2 = r_3$



- $(M_1 \times M_2) \times M_3$
  - uses $(c_1 \cdot r_1 \cdot c_2) + c_2 \cdot r_1 \cdot c_3$ 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
  - $10 \cdot 7 \cdot 8 + (20 \cdot 10 \cdot 8) = 2160$

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

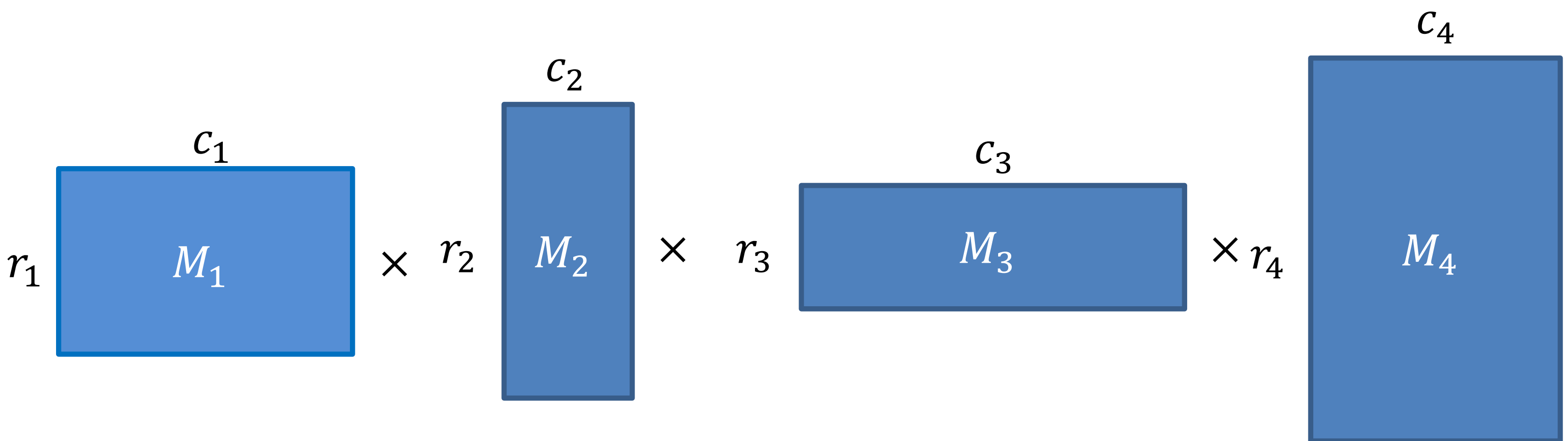
$c_1 = 10$
$c_2 = 20$
$c_3 = 8$
$r_1 = 7$
$r_2 = 10$
$r_3 = 20$

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

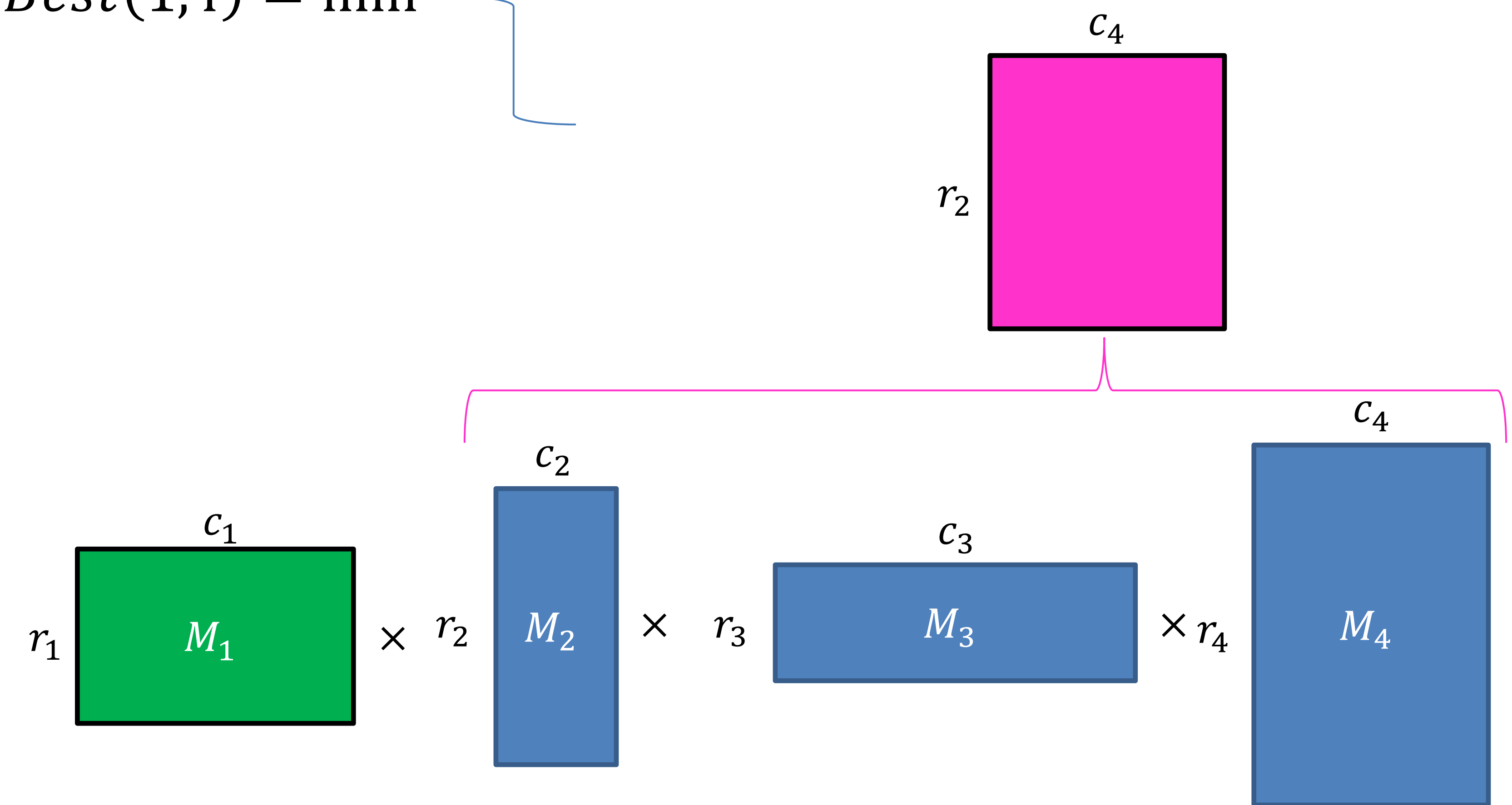
$Best(1, n)$ = cheapest way to multiply together M_1 through M_n



1. Identify the Recursive Structure of the Problem

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

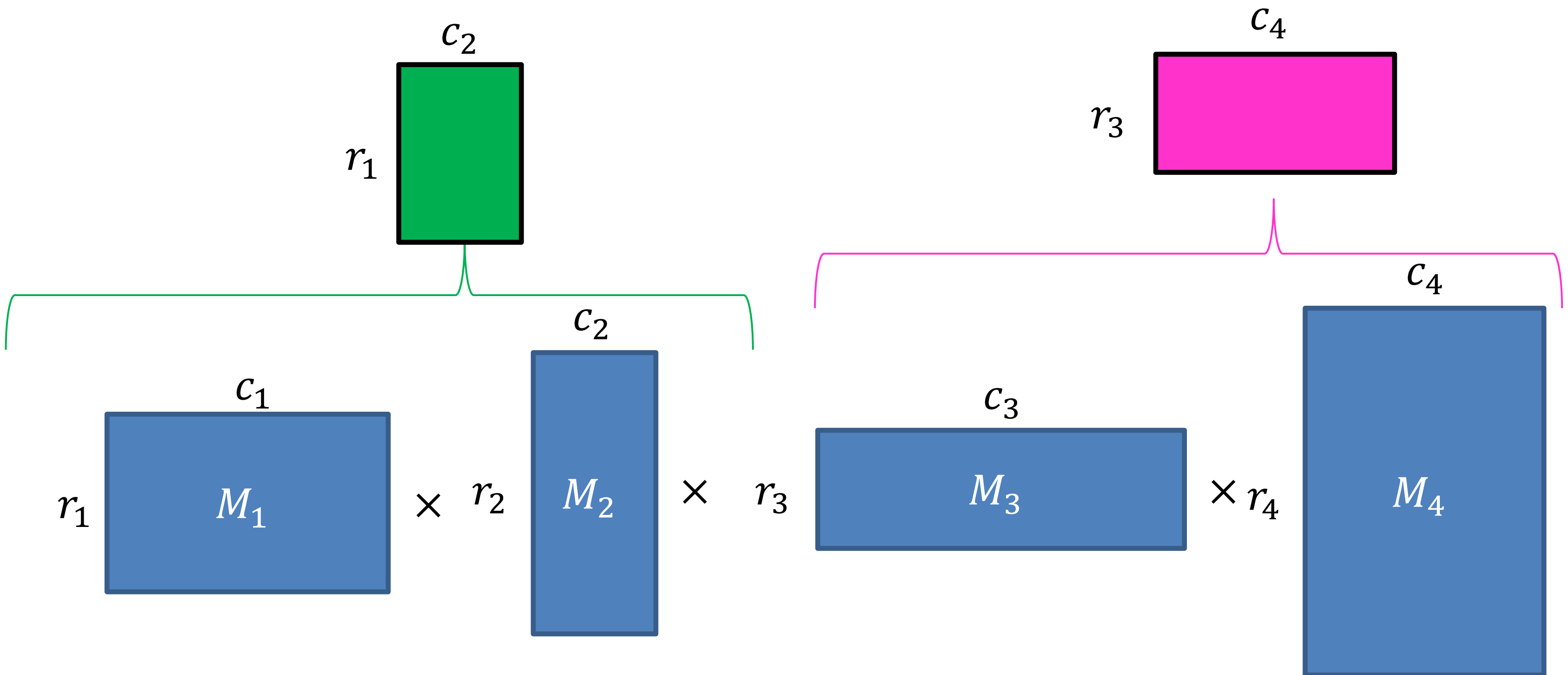
$$Best(1,4) = \min \begin{cases} Best(2,4) + r_1 r_2 c_4 \\ \\ \end{cases}$$



1. Identify the Recursive Structure of the Problem

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

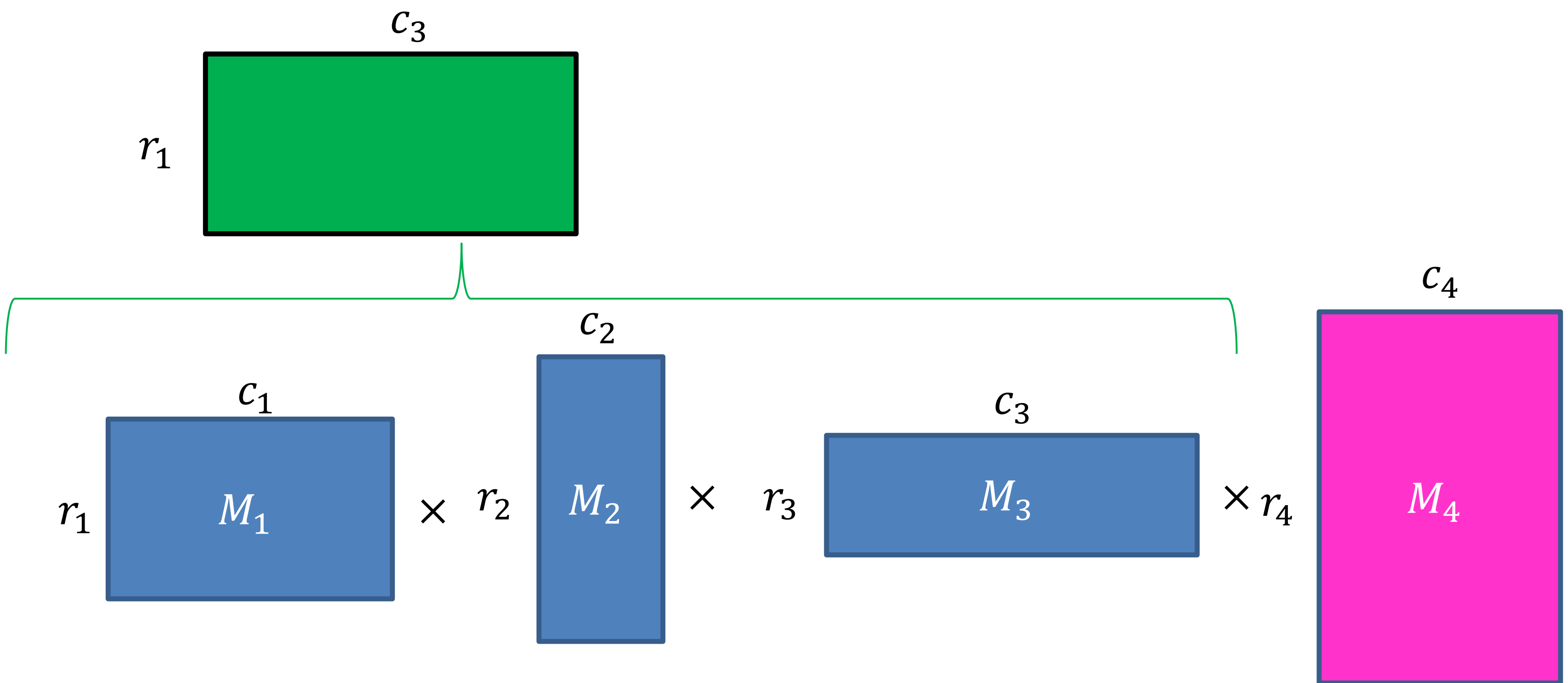
$$Best(1,4) = \min \begin{cases} Best(2,4) + r_1 r_2 c_4 \\ Best(1,2) + Best(3,4) + r_1 r_3 c_4 \\ \end{cases}$$



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:

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

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

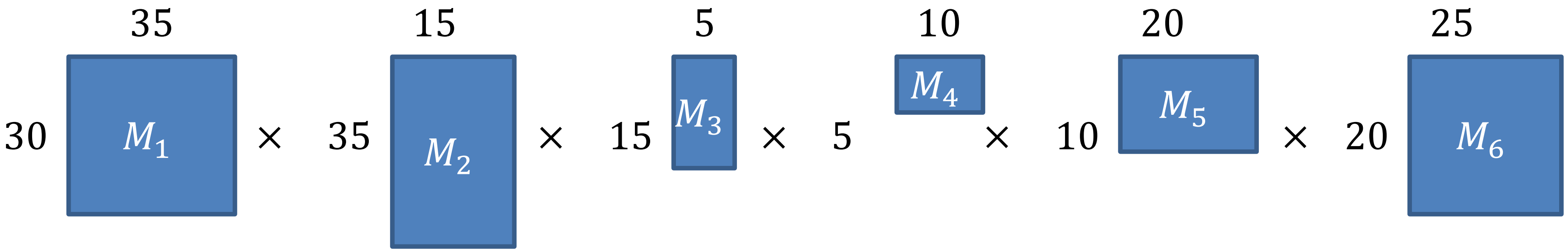
$$Best(i,i) = 0$$

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

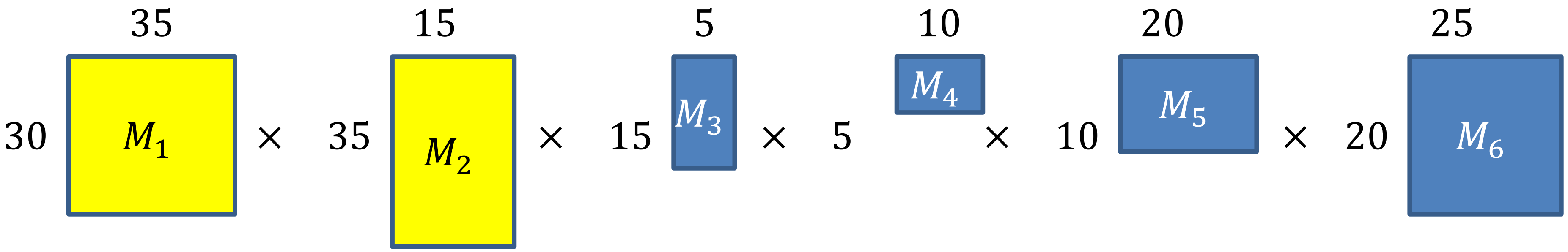


$$Best(i,j) = \min_{k=i}^{j-1}\big(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\big)$$

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

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

3. Select a good order for solving subproblems



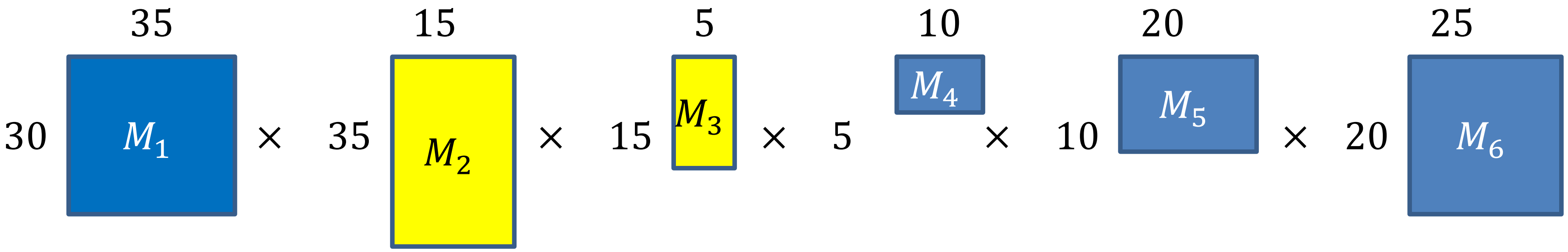
$$Best(i,j) = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$$

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

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

$$Best(1,2) = \min\left\{ Best(1,1) + Best(2,2) + r_1 r_2 c_2 \right.$$

3. Select a good order for solving subproblems



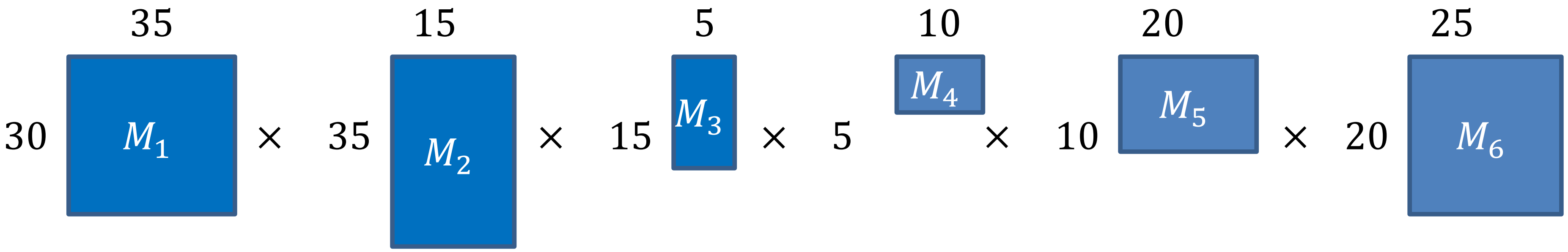
$$Best(i,j) = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$$

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

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

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

3. Select a good order for solving subproblems

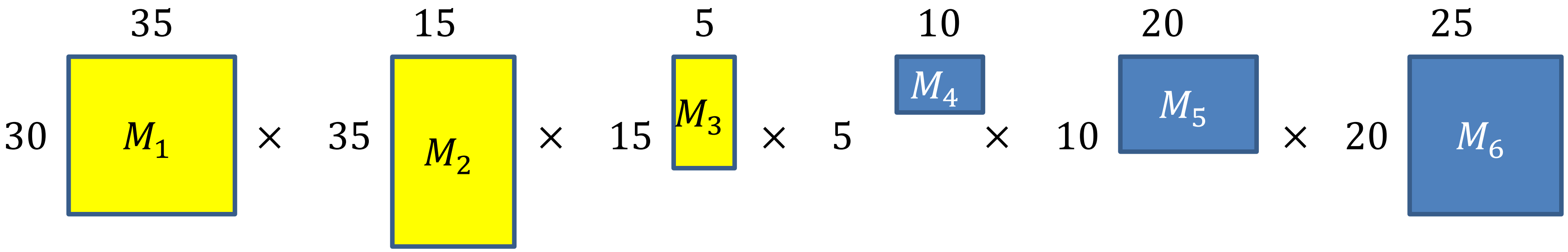


$$Best(i,j) = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$$

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

$j =$	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



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

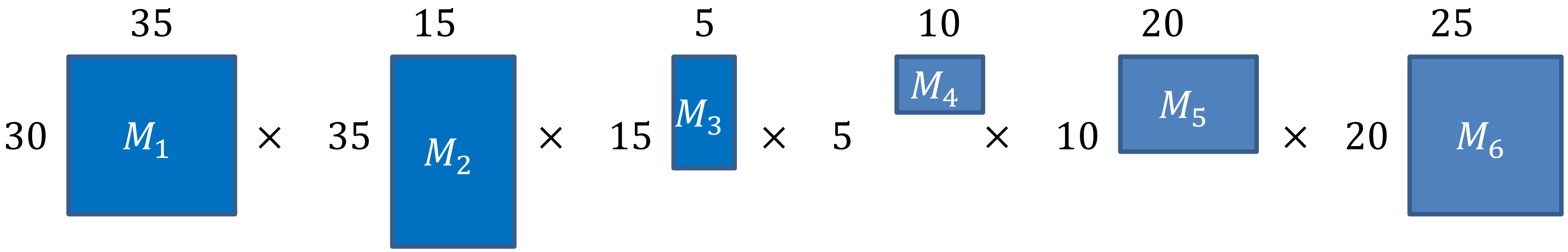
$$Best(i,i) = 0$$

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

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

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

3. Select a good order for solving subproblems



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

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

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

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

Conclusion: solve in order of diagonal

Matrix Chaining

$$\underset{30\times 35}{M_1} \times \underset{35\times 15}{M_2} \times \underset{15\times 5}{M_3} \times \underset{5\times 10}{M_4} \times \underset{10\times 20}{M_5} \times \underset{20\times 25}{M_6}$$

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

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

$j =$ 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

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

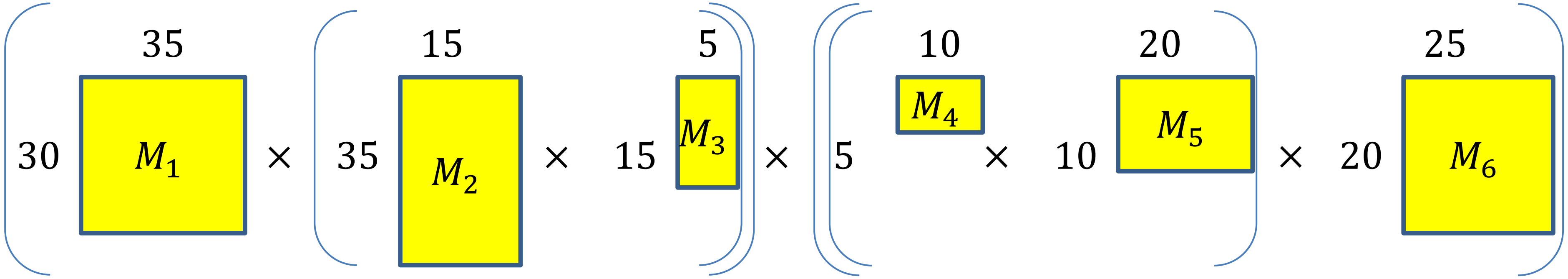
$$Best(i,i) = 0$$

$j =$	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

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

32

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 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

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

Storing and Recovering Optimal Solution

- Maintain table Choice[i,j] in addition to Best table
  - Choice[i,j] = k means the best "split" was right after M_k
  - Work backwards from value for whole problem, Choice[1,n]
  - Note: Choice[i,i+1] = i because there are just 2 matrices
- From our example:
  - Choice[1,6] = 3. So $[M_1\ M_2\ M_3]\ [M_4\ M_5\ M_6]$
  - We then need Choice[1,3] = 1. So $[(M_1)\ (M_2\ M_3)]$
  - Also need Choice[4,6] = 5. So $[(M_4\ M_5)\ M_6]$
  - Overall: $[(M_1)\ (M_2\ M_3)]\ [(M_4\ M_5)\ M_6]$

Dynamic Programming

- Requires Optimal Substructure
  – Solution to larger problem contains the solutions to smaller ones
  – Or: If S is an optimal solution to a problem, then the components of S are optimal solutions to sub-problems
- 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

Longest Common Subsequence

Given two sequences X and Y, find the length of their longest common subsequence

Example:

$X = ATCTGAT$

$Y = TGCATA$

$LCS = TCTA$

Brute force: Compare every subsequence of X with Y

$\Omega(2^n)$

Applications other than bioinformatics?
Of course, Including version control!
http://cbx33.github.io/gitt/afterhours3-1.html

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

Let $LCS(i, j)$ = length of the LCS for the first i characters of X, first j character of Y

Find $LCS(i, j)$:

Case 1: $X[i] = Y[j]$

$$X = ATCTGCGT$$
$$Y = TGCATAT$$
$$LCS(i, j) = LCS(i-1, j-1) + 1$$

Case 2: $X[i] \neq Y[j]$

$$X = ATCTGCGA$$
$$Y = TGCATAC$$
$$LCS(i, j) = LCS(i, j-1)$$

$$X = ATCTGCGA$$
$$Y = TGCATAC$$
$$LCS(i, j) = LCS(i-1, j)$$

$$LCS(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i-1, j-1) + 1 & \text{if } X[i] = Y[j] \\ \max(LCS(i, j-1), LCS(i-1, j)) & \text{otherwise} \end{cases}$$

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

1. Identify Recursive Structure

Let $LCS(i, j)$ = length of the LCS for the first i characters of X, first j character of Y

Find $LCS(i, j)$:

Case 1: $X[i] = Y[j]$

$$X = ATCTGCGT$$
$$Y = TGCATAT$$

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

Case 2: $X[i] \neq Y[j]$

$$X = ATCTGCGA$$
$$Y = TGCATAC$$
$$LCS(i, j) = LCS(i, j-1)$$

$$X = ATCTGCGA$$
$$Y = TGCATAC$$
$$LCS(i, j) = LCS(i-1, j)$$

$$LCS(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i-1, j-1) + 1 & \text{if } X[i] = Y[j] \\ \max(LCS(i, j-1), LCS(i-1, j)) & \text{otherwise} \end{cases}$$

Read from M[i,j] if present

Save to M[i,j]

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. Solve in a Good Order

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i-1, j-1) + 1 & \text{if } X[i] = Y[j] \\ \max(LCS(i, j-1), LCS(i-1, j)) & \text{otherwise} \end{cases}$$

$Y =$ \ $X =$	0	A 1	T 2	C 3	T 4	G 5	A 6	T 7
0	0	0	0	0	0	0	0	0
T 1	0	0	1	1	1	1	1	1
G 2	0	0	1	1	1	2	2	2
C 3	0	0	1	2	2	2	2	2
A 4	0	1	1	2	2	2	3	3
T 5	0	1	2	2	3	3	3	4
A 6	0	1	2	2	3	3	4	4

To fill in cell (i,j) we need cells $(i-1, j-1), (i-1, j), (i, j-1)$
Fill from Top->Bottom, Left->Right (with any preference)

LCS Length Algorithm

LCS-Length(X, Y) // Y for M's rows, X for its columns

1. n = length(X) // get the # of symbols in X

2. m = length(Y) // get the # of symbols in Y

3. for i = 1 to m   M[i,0] = 0   // special case: Y_0

4. for j = 1 to n   M[0,j] = 0   // special case: X_0

5. for i = 1 to m   // for all Y_i

6.   for j = 1 to n   // for all X_j

7.   if (X[i] == Y[j])

8.   M[i,j] = M[i-1,j-1] + 1

9.   else M[i,j] = max(M[i-1,j], M[i,j-1])

10. return M[m,n]   // return LCS length for Y and X

Run Time?

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i-1, j-1) + 1 & \text{if } X[i] = Y[j] \\ \max(LCS(i, j-1), LCS(i-1, j)) & \text{otherwise} \end{cases}$$

$Y =$ \ $X =$	0	A 1	T 2	C 3	T 4	G 5	A 6	T 7
0	0	0	0	0	0	0	0	0
T 1	0	0	1	1	1	1	1	1
G 2	0	0	1	1	1	2	2	2
C 3	0	0	1	2	2	2	2	2
A 4	0	1	1	2	2	2	3	3
T 5	0	1	2	2	3	3	3	4
A 6	0	1	2	2	3	3	4	4

Run Time: $\Theta(n \cdot m)$ (for $|X| = n$, $|Y| = m$)

Reconstructing the LCS

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i-1, j-1) + 1 & \text{if } X[i] = Y[j] \\ \max(LCS(i, j-1), LCS(i-1, j)) & \text{otherwise} \end{cases}$$

$Y =$	$X =$		A	T	C	T	G	A	T
		0	1	2	3	4	5	6	7
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
T	1	**0**	0	1	1	1	1	1	1
G	2	**0**	0	1	1	1	2	2	2
C	3	**0**	0	1	2	2	2	2	2
A	4	**0**	1	1	2	2	2	3	3
T	5	**0**	1	2	2	3	3	3	4
A	6	**0**	1	2	2	3	3	4	4

Start from bottom right,

if symbols matched, print that symbol then go diagonally

else go to largest adjacent

Reconstructing the LCS

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i-1, j-1) + 1 & \text{if } X[i] = Y[j] \\ \max(LCS(i, j-1), LCS(i-1, j)) & \text{otherwise} \end{cases}$$

$Y =$ \ $X =$	0	A 1	T 2	C 3	T 4	G 5	A 6	T 7
0	0	0	0	0	0	0	0	0
T 1	0	0	1	1	1	1	1	1
G 2	0	0	1	1	1	2	2	2
C 3	0	0	1	2	2	2	2	2
A 4	0	1	1	2	2	2	3	3
T 5	0	1	2	2	3	3	3	4
A 6	0	1	2	2	3	3	4	4

Start from bottom right,

if symbols matched, print that symbol then go diagonally

else go to largest adjacent

Reconstructing the LCS

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i-1, j-1) + 1 & \text{if } X[i] = Y[j] \\ \max(LCS(i, j-1), LCS(i-1, j)) & \text{otherwise} \end{cases}$$

$Y =$ \ $X =$		0	A 1	$\boxed{T}$ 2	$\boxed{C}$ 3	$\boxed{T}$ 4	G 5	$\boxed{A}$ 6	T 7
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
$\boxed{T}$	1	**0**	0	1	1	1	1	1	1
G	2	**0**	0	1	1	1	2	2	2
$\boxed{C}$	3	**0**	0	1	2	2	2	2	2
A	4	**0**	1	1	2	2	2	3	3
$\boxed{T}$	5	**0**	1	2	2	3	3	3	4
$\boxed{A}$	6	**0**	1	2	2	3	3	4	4

Start from bottom right,

if symbols matched, print that symbol then go diagonally

else go to largest adjacent

Top-Down Solution with Memoization

We need two functions; one will be recursive.

```
LCS-Length(X, Y) // Y is M's cols.
1. n = length(X)
2. m  = length(Y)
3. Create table M[m,n]
4. Assign -1 to all cells M[i,j]
// get value for entire sequences
5. return LCS-recur(X, Y, M, m, n)
```

```
LCS-recur(X, Y, M, i, j)
1. if (i == 0 || j == 0) return 0
// have we already calculated this subproblem?
2. if (M[i,j] != -1) return M[i,j]
3. if ( X[i] == Y[j] )
4.   M[i,j] = LCS-recur(X, Y, M, i-1, j-1) + 1
5. else
6.    M[i,j] = max( LCS-recur(X, Y, M, i-1, j),
                    LCS-recur(X, Y, M, i, j-1) )
7. return M[i,j]
```

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

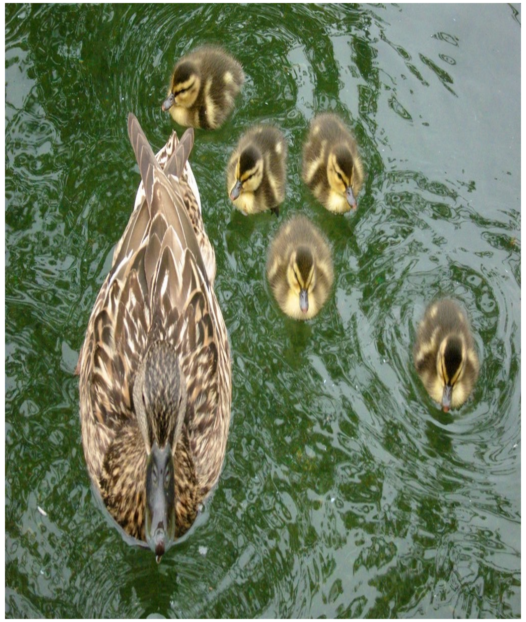
- Method for image resizing that doesn't scale/crop the image

Seam Carving

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

Cropped

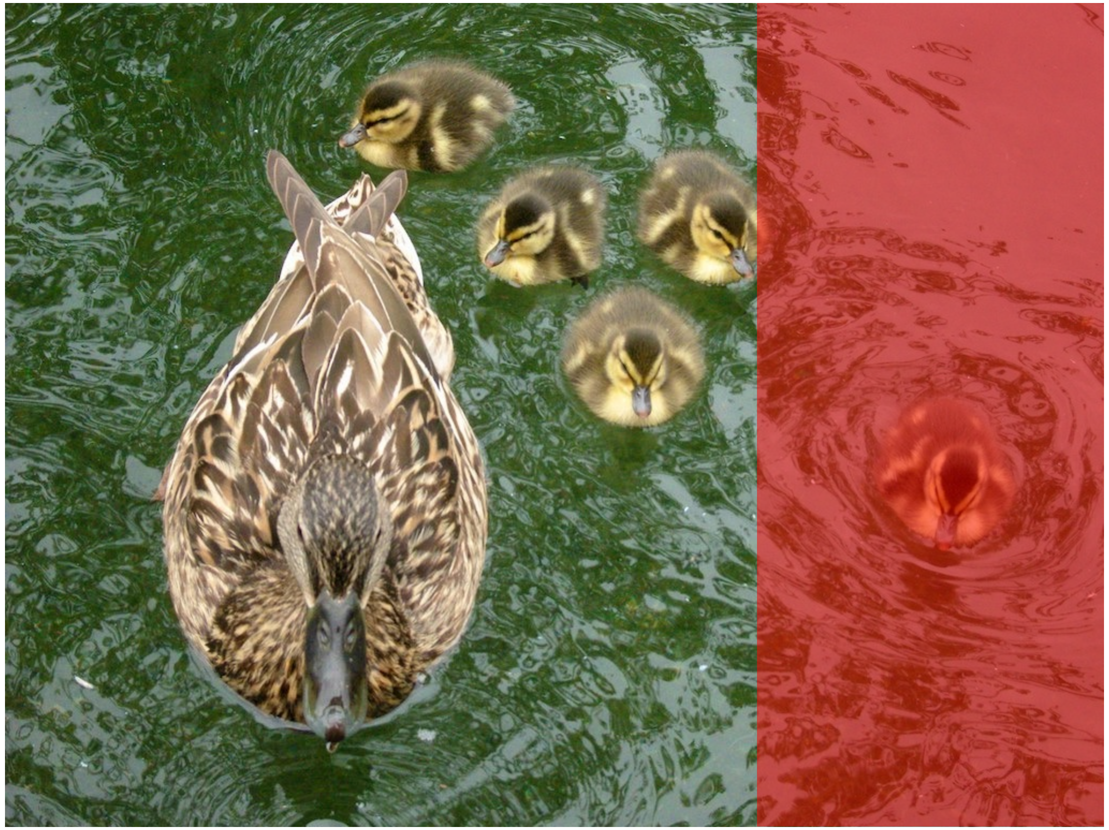
Scaled



Carved

Cropping

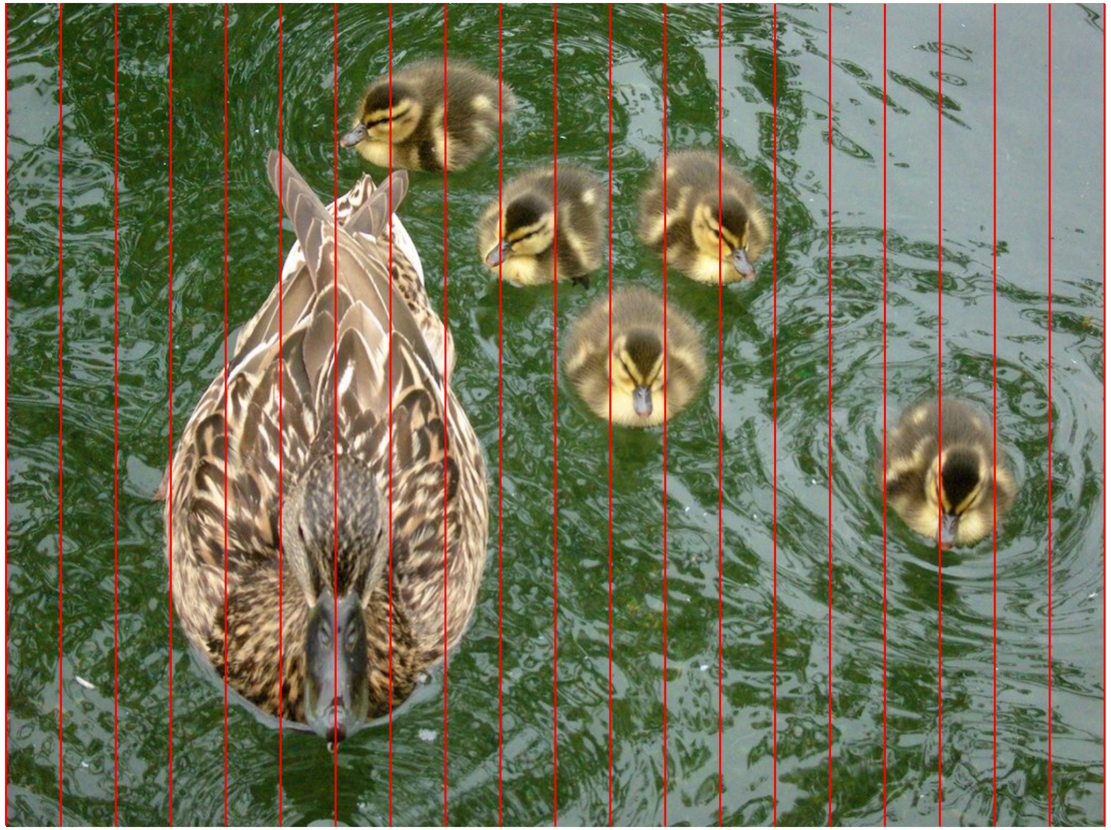
- Removes a “block” of pixels



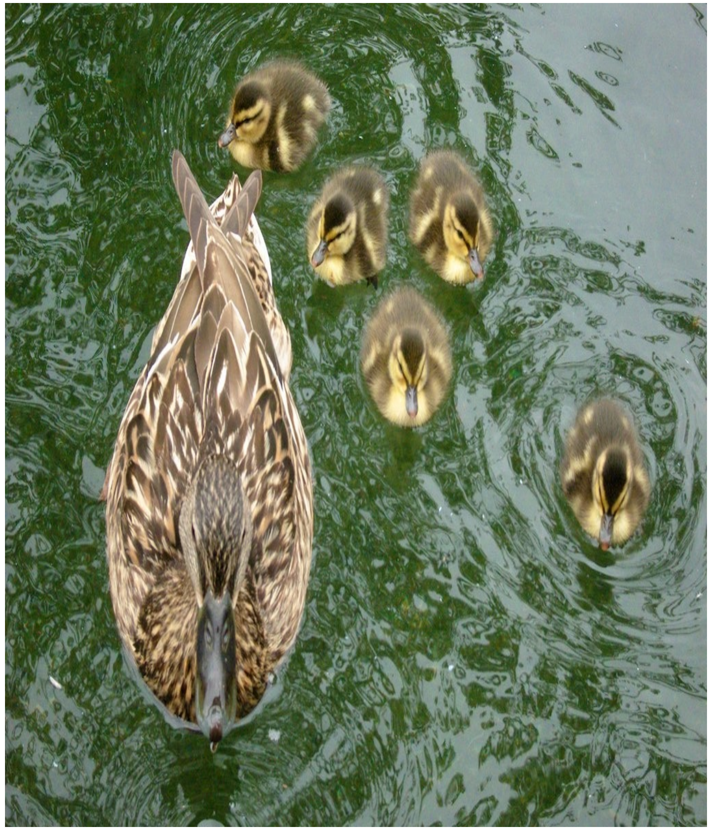
Cropped

Scaling

- Removes "stripes" of pixels

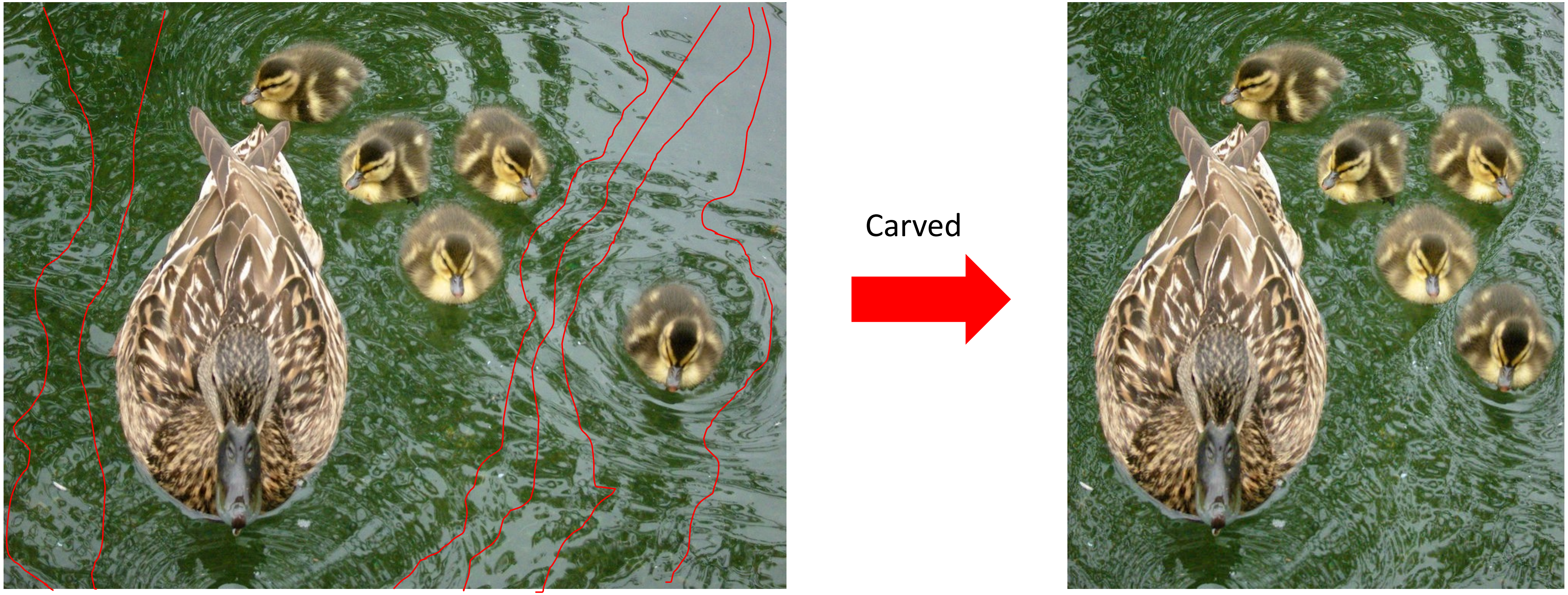


Scaled



Seam Carving

- Removes “least energy seam” of pixels from bottom to top
  - https://www.aryan.app/seam-carving/ (also see Wikipedia page)
  - http://rsizr.com/ - old, uses Flash ☹



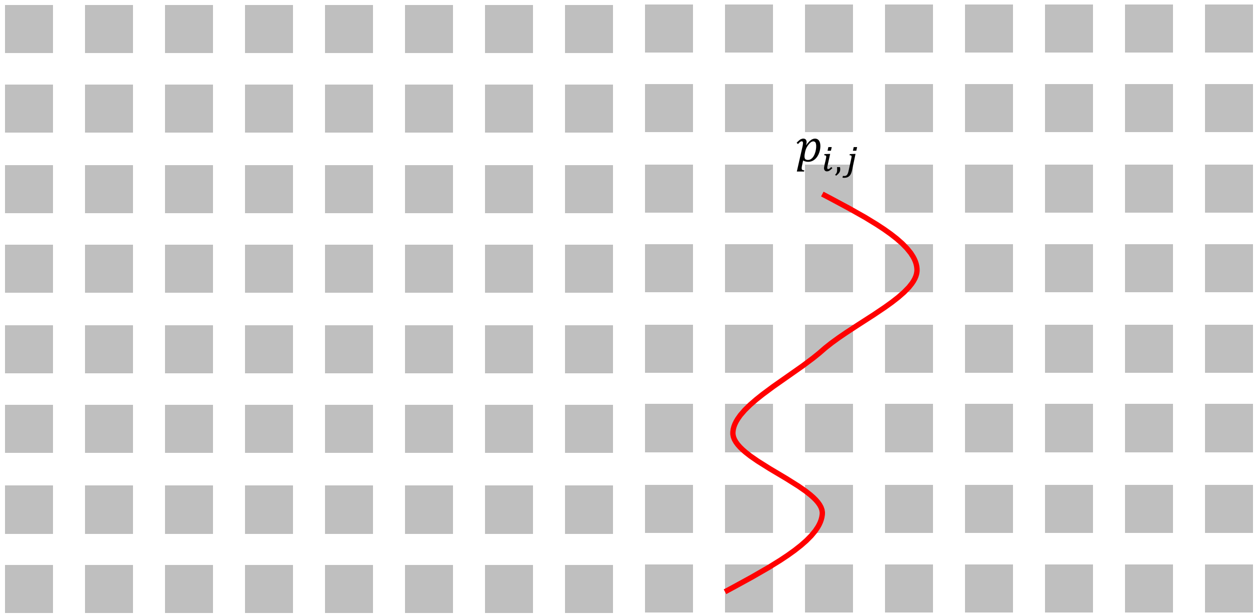
Seattle Skyline

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

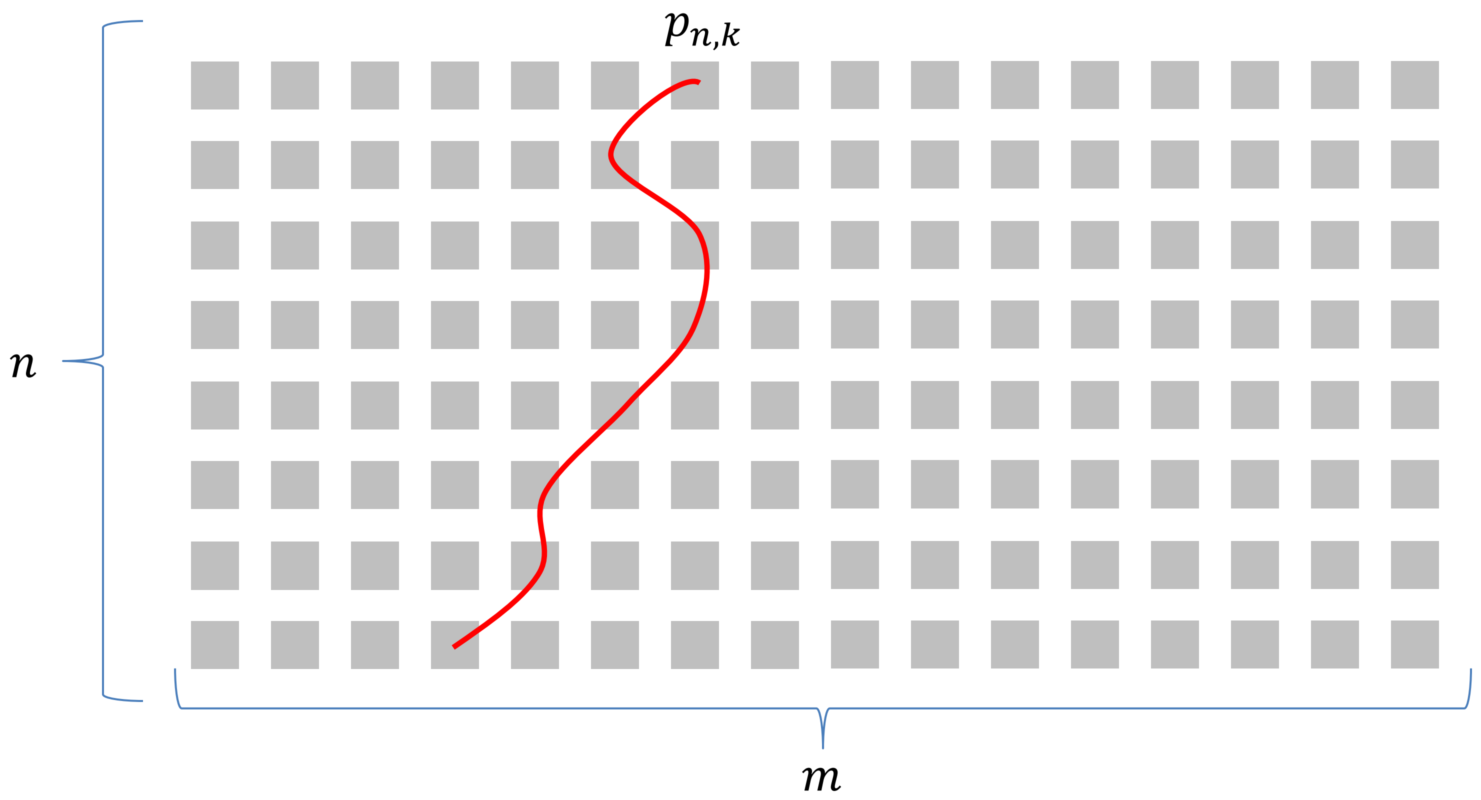
Let $S(i,j)$ = least energy seam from the bottom of the image up to pixel $p_{i,j}$



Finding the Least Energy Seam

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

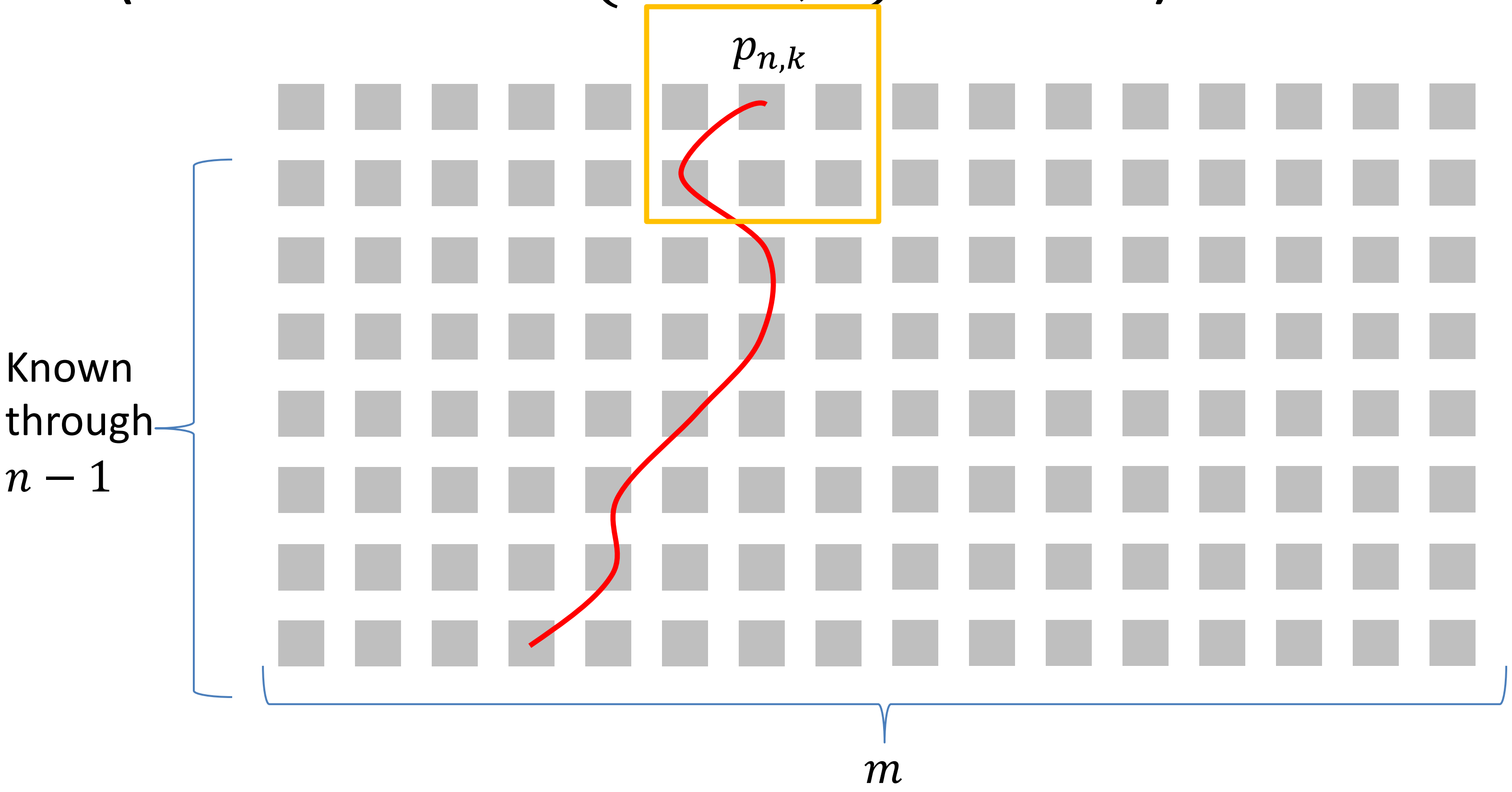
$$\min_{k=1}^{m}(S(n,k))$$



Computing $S(n,k)$

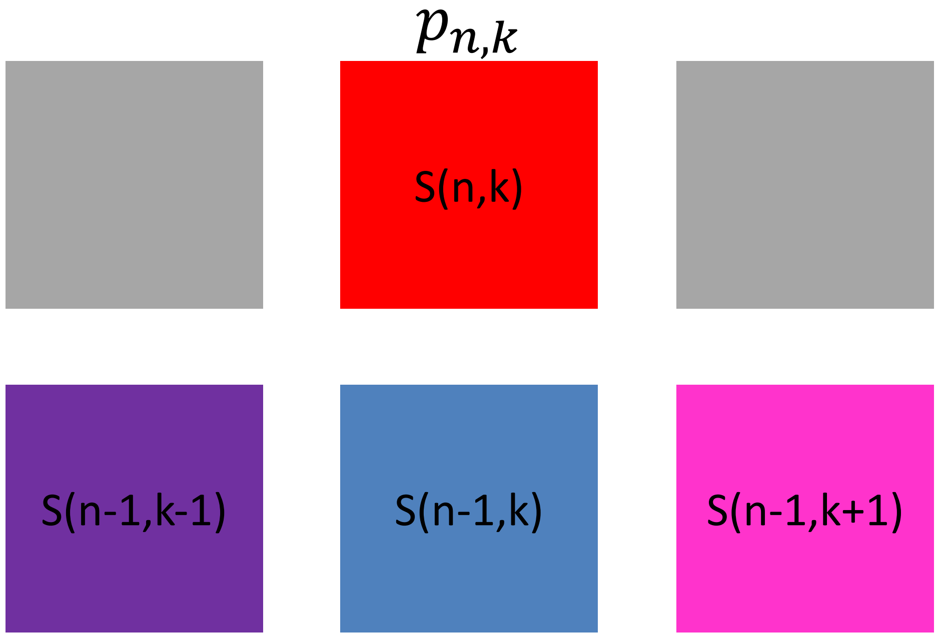
Assume we know the least energy seams for all of row $n-1$

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



Computing $S(n,k)$

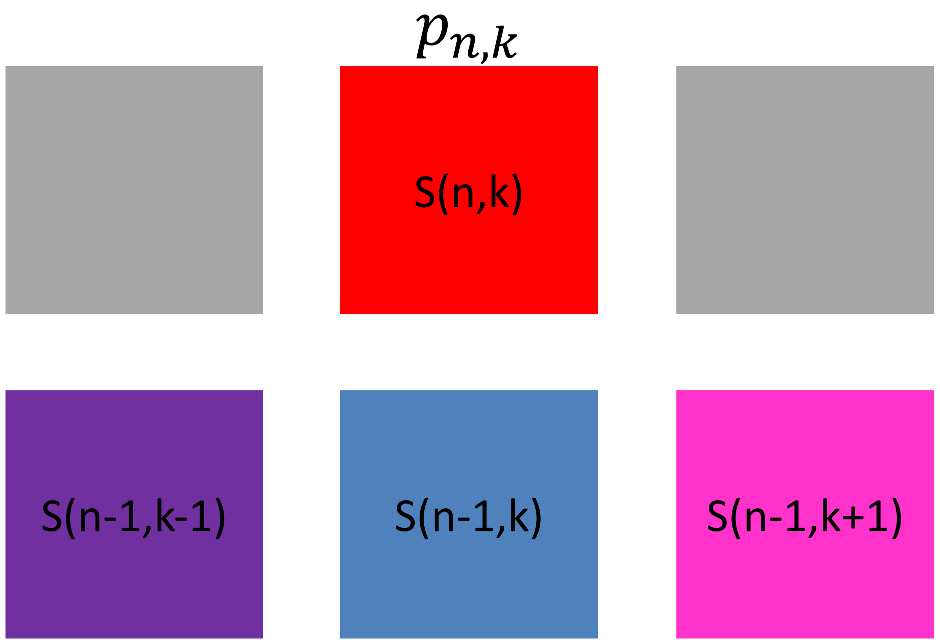
Assume we know the least energy seams for all of row $n-1$ (i.e. we know $S(n-1,\ell)$ for all ℓ)



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)

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



Seam Carving

- Details left to you! Unit C Programming assignment
  - Note: Python or Java implementations only this time

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