CS 3100
Data Structures and Algorithms 2

Lecture 18: Seam Carving

Co-instructors: Robbie Hott and Ray Pettit

Spring 2024

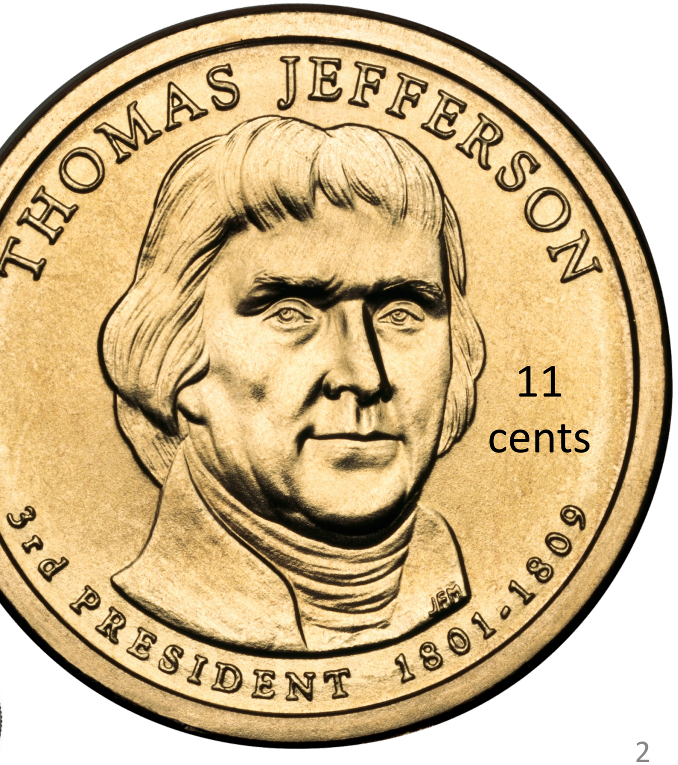
Readings in CLRS 4th edition:

- Chapter 14

Warm Up!

Remember change making?

Given access to unlimited quantities of pennies, nickels, dimes, toms, and quarters (worth value 1, 5, 10, 11, 25 respectively), give 90 cents change using the **fewest** number of coins.



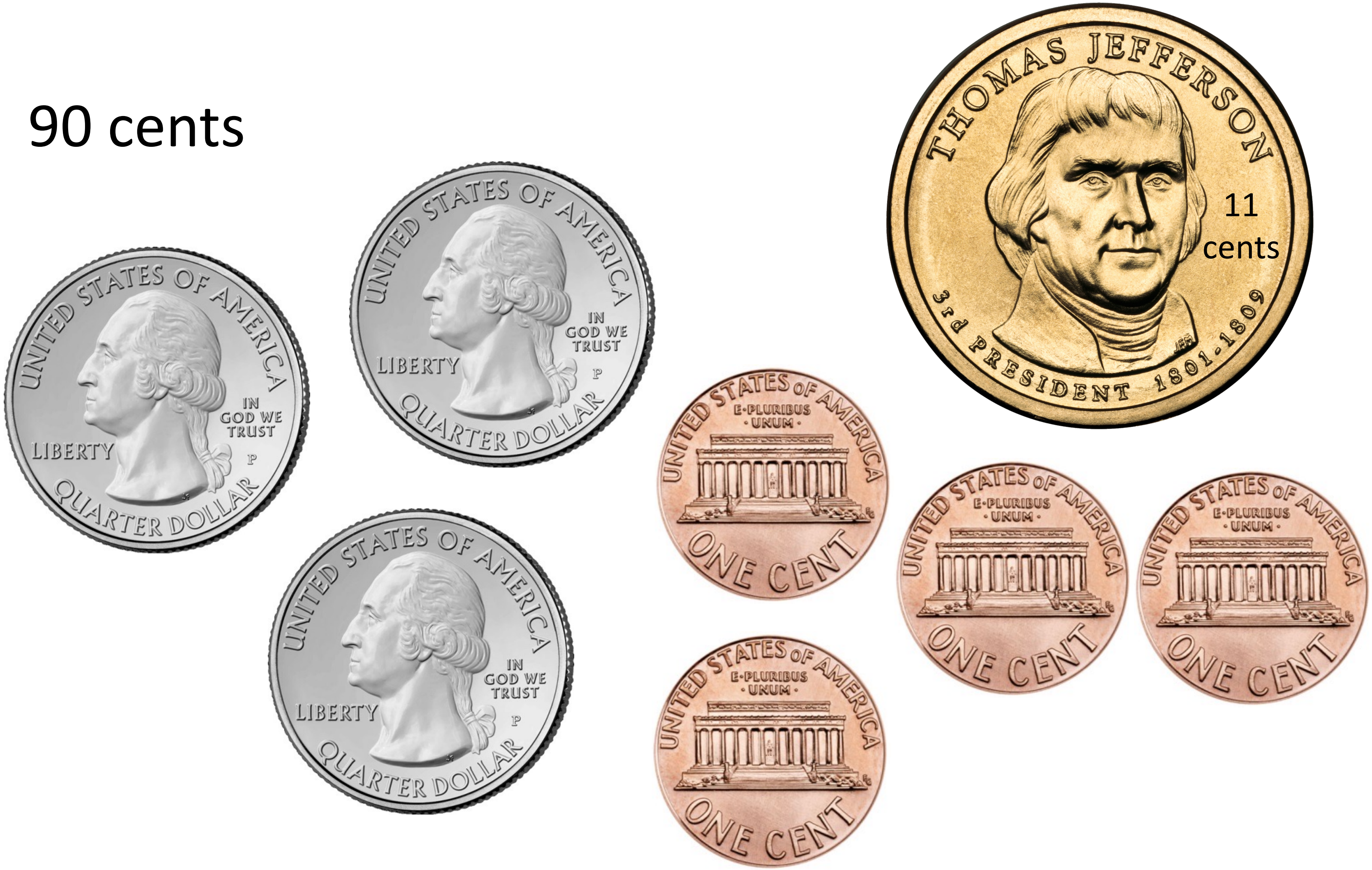
Remember: Greedy Change Making Algorithm

- Given: target value x, list of coins $C = [c_1, \dots, c_n]$
  (in this case $C = [1,5,10,25]$)
- Repeatedly select the largest coin less than the remaining target value:

```
while($x > 0$)
    let $c = \max(c_i \in \{c_1, \dots, c_n\} \mid c_i \leq x)$
    print $c$
    $x = x - c$
```

Greedy solution

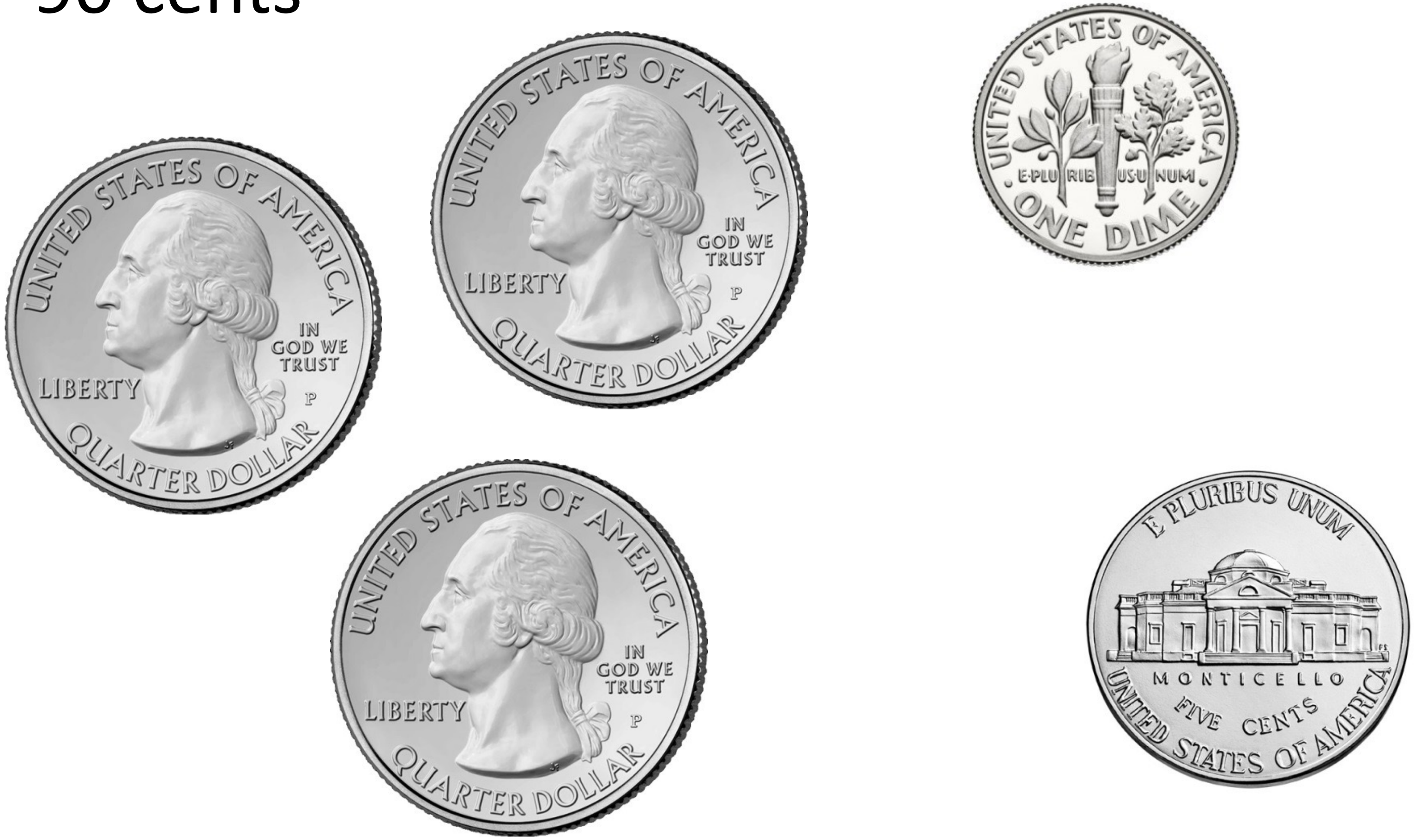
90 cents



11 cents

Greedy solution

90 cents



Why does greedy always work for US coins?

- If $x < 5$, then pennies only
  - Else 5 pennies can be exchanged for a nickel

Only case Greedy uses pennies!

- If $5 \leq x < 10$ we must have a nickel
  - Else 2 nickels can be exchanged for a dime

Only case Greedy uses nickels!

- If $10 \leq x < 25$ we must have at least 1 dime
  - Else 3 dimes can be exchanged for a quarter and a nickel

Only case Greedy uses dimes!

- If $x \geq 25$ we must have at least 1 quarter

Only case Greedy uses quarters!

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

Identify Recursive Structure

$\text{Change}(n)$: minimum number of coins needed to give change for n cents

Possibilities for last coin	Coins needed	
	$\text{Change}(n-25)+1$	if $n \geq 25$
	$\text{Change}(n-11)+1$	if $n \geq 11$
	$\text{Change}(n-10)+1$	if $n \geq 10$
	$\text{Change}(n-5)+1$	if $n \geq 5$
	$\text{Change}(n-1)+1$	if $n \geq 1$

Identify Recursive Structure

$\text{Change}(n)$: minimum number of coins needed to give change for n cents

$$\text{Change}(n) = \min \begin{cases} \text{Change}(n-25)+1 & \text{if } n \geq 25 \\ \text{Change}(n-11)+1 & \text{if } n \geq 11 \\ \text{Change}(n-10)+1 & \text{if } n \geq 10 \\ \text{Change}(n-5)+1 & \text{if } n \geq 5 \\ \text{Change}(n-1)+1 & \text{if } n \geq 1 \end{cases}$$

Correctness: The optimal solution must be contained in one of these configurations

Base Case: $\text{Change}(0) = 0$

Input size is $O(k \log n)$

Running time: $O(kn)$

k is number of possible coins

Is this efficient?

No, this is pseudo-polynomial time

Announcements

- PS8 available soon
- PA4 now available!
- Office hours updates
  - Prof Hott Office Hours:
    - Tomorrow: 2-3pm only (no 10am hours)
    - Monday 4/1: 10-11am
    - Tuesday 4/2: 2-3pm

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 $\ell_1, \dots, \ell_k$ such that:

$\sum \ell_i = n$

to maximize $\sum P[\ell_i]$

Brute Force: $O(2^n)$

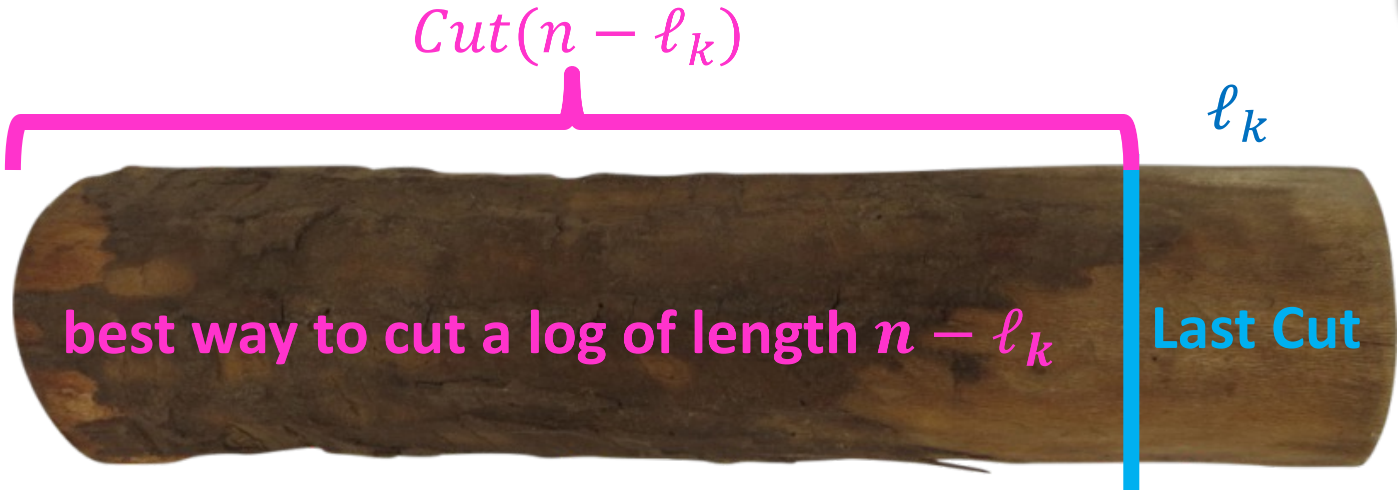
1. Identify Recursive Structure

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

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

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

2. Save sub-solutions to memory!



3. Select a Good Order for Solving Subproblems

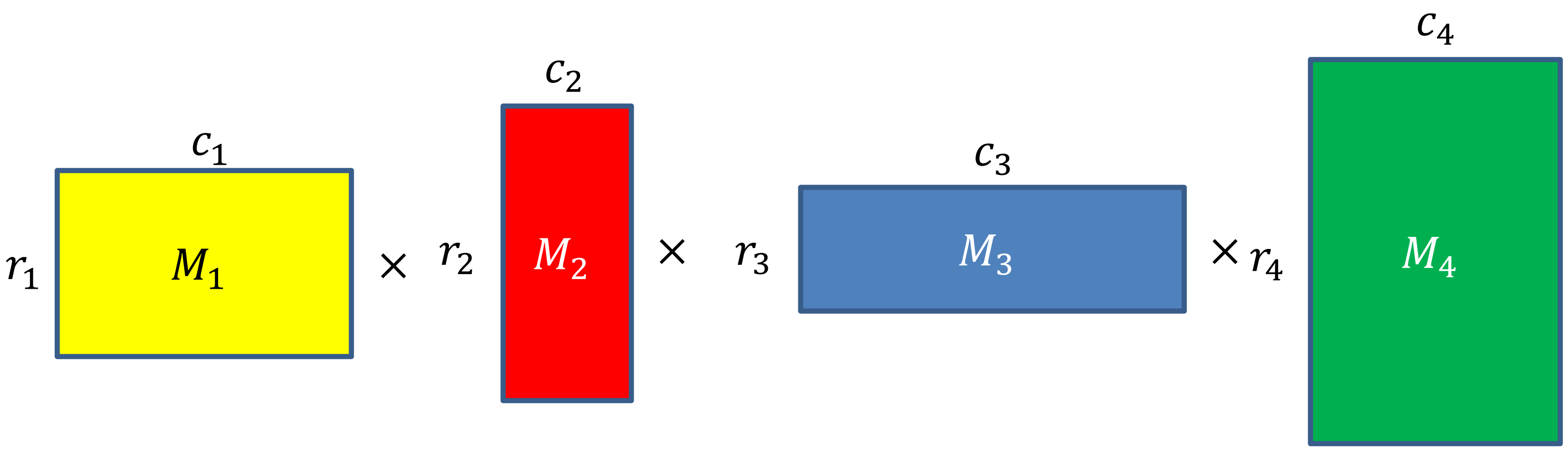
Solve Smallest subproblem first

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

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

Matrix Chaining

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



1. Identify the Recursive Structure of the Problem

- In general:

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

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

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

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

2. Save Subsolutions in Memory

- In general:

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

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

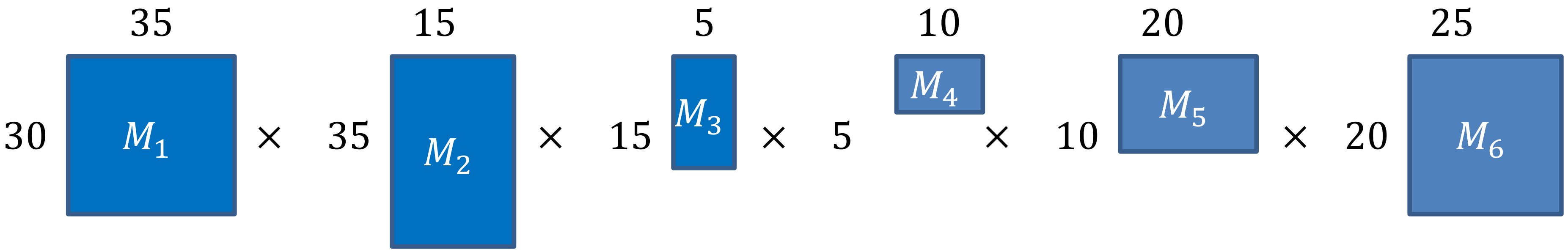
$$Best(i, i) = 0$$

Read from M[n] if present

Save to M[n]

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

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	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

Seinfeld break

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

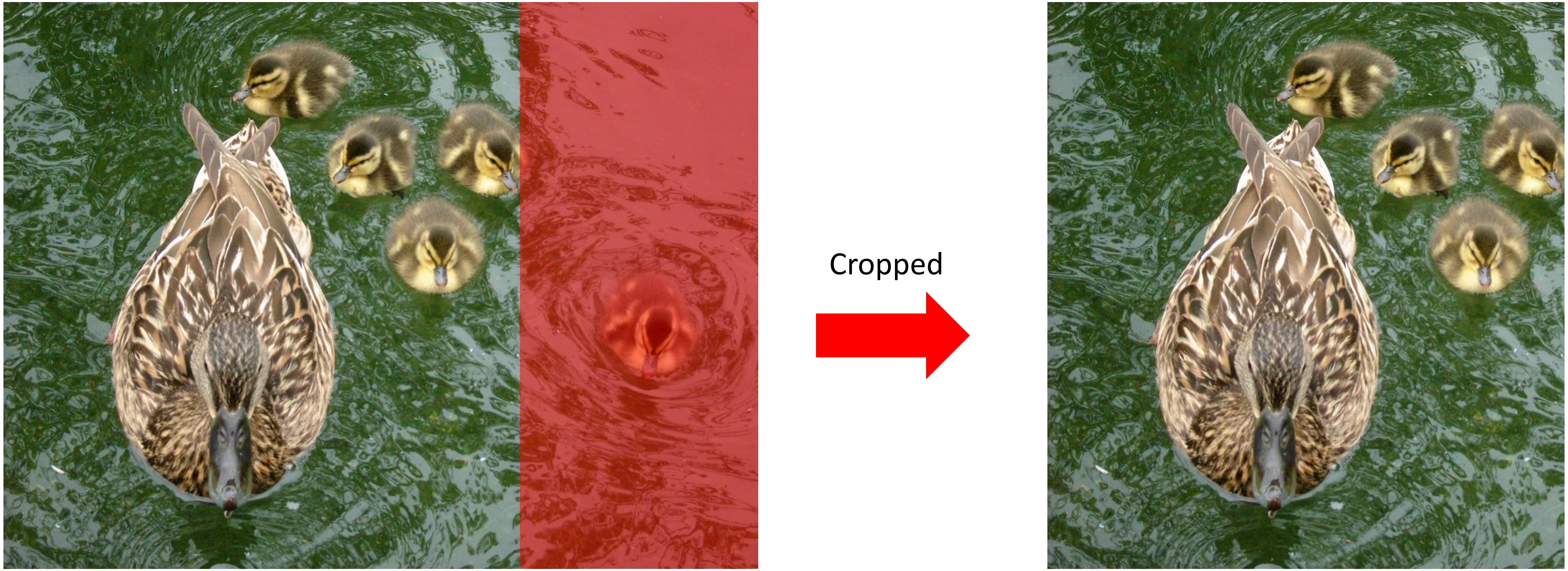
- Method for image resizing that doesn’t scale/crop the image

Seam Carving

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

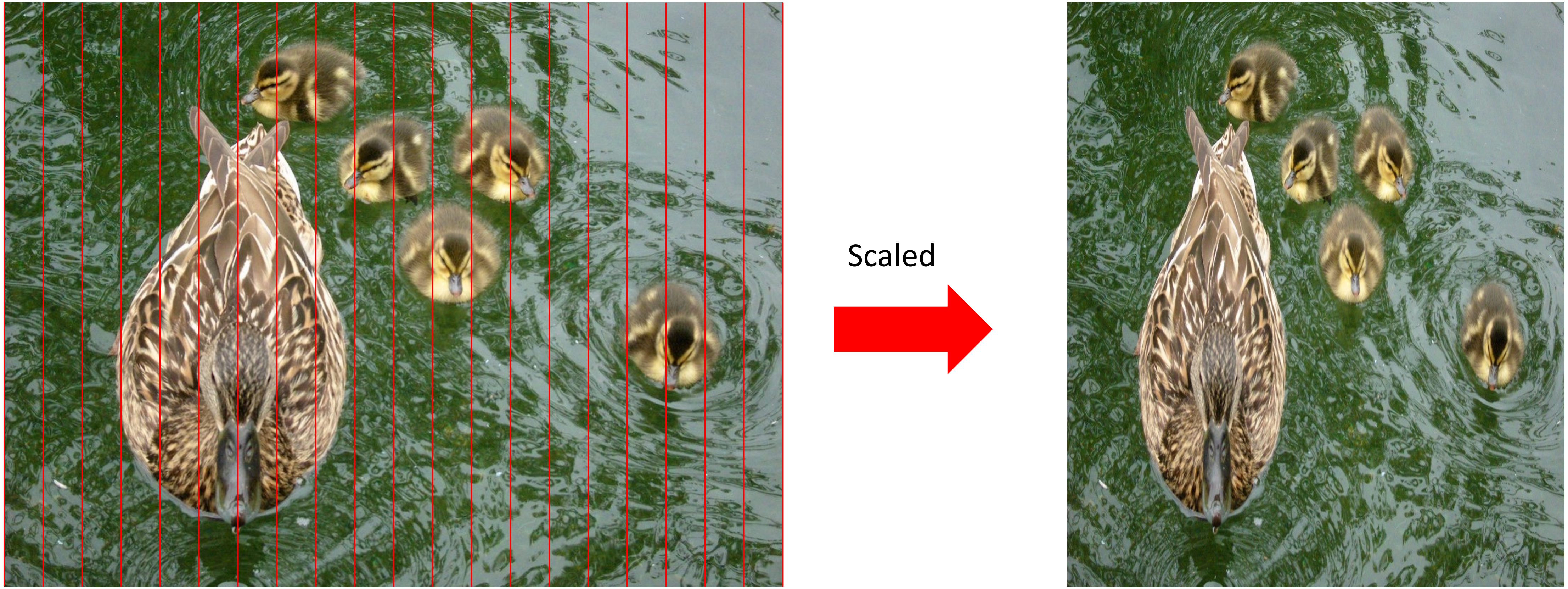
Cropping

- Removes a “block” of pixels



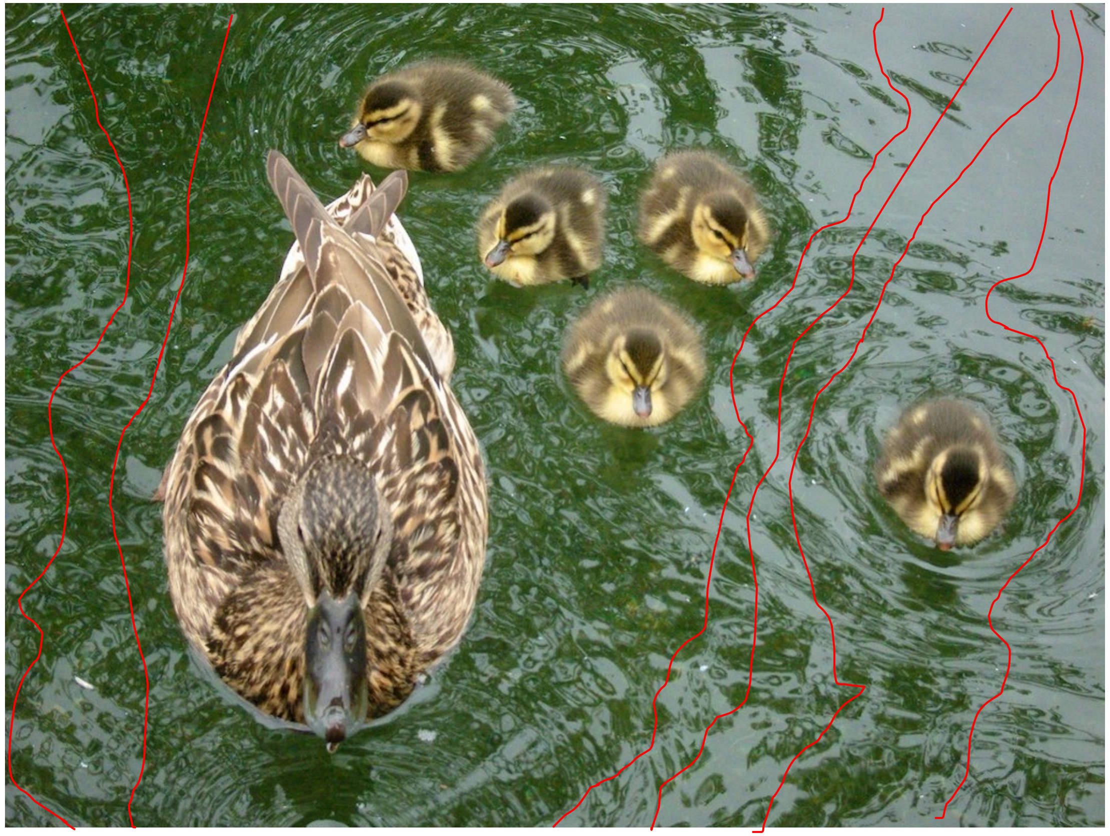
Scaling

- Removes “stripes” of pixels

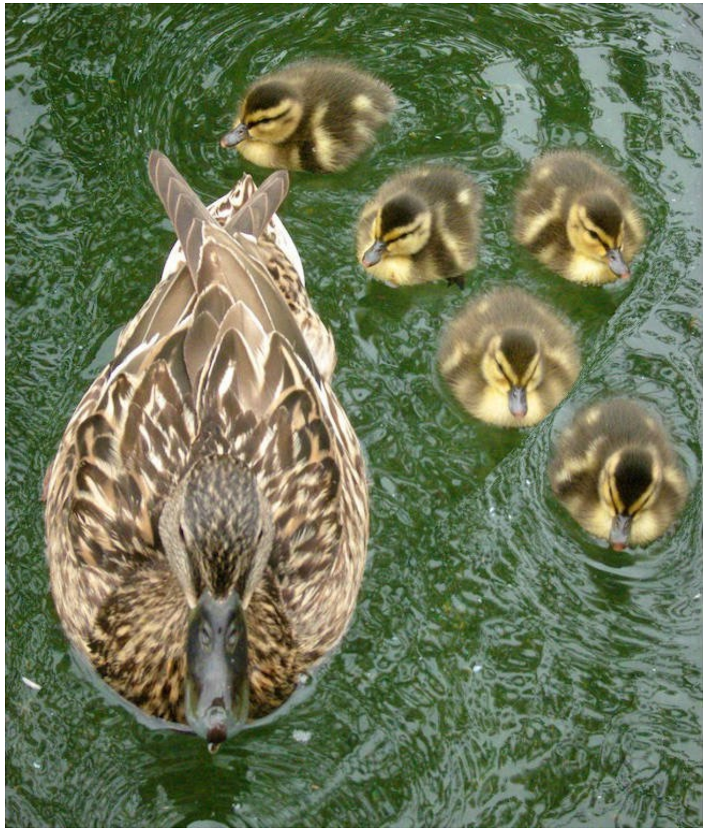


Seam Carving

- Removes "least energy seam" of pixels
- https://trekhleb.dev/js-image-carver/



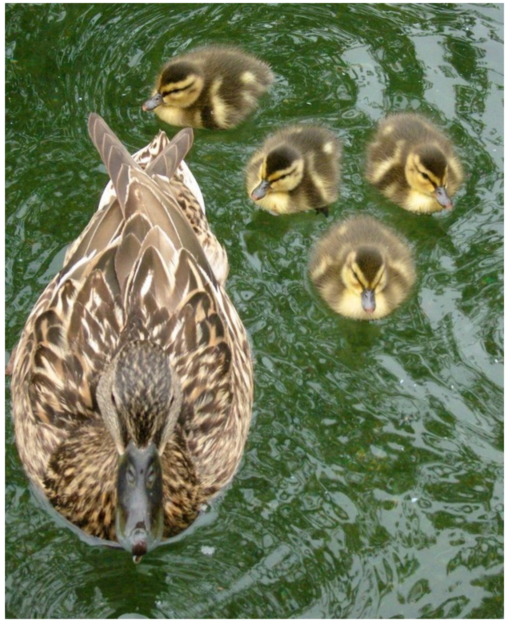
Carved



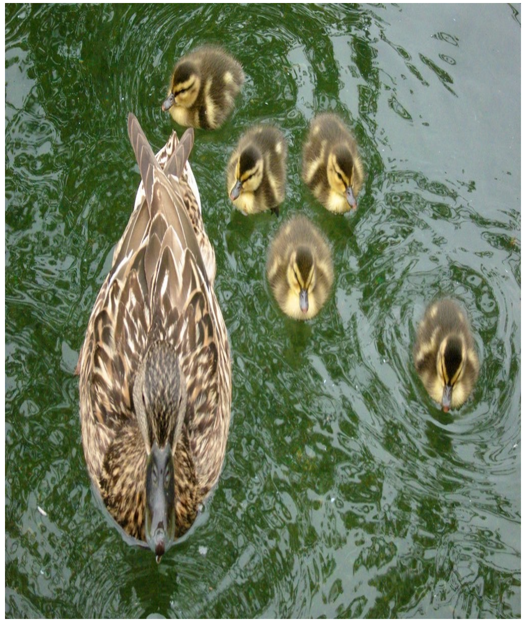
Seam Carving

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

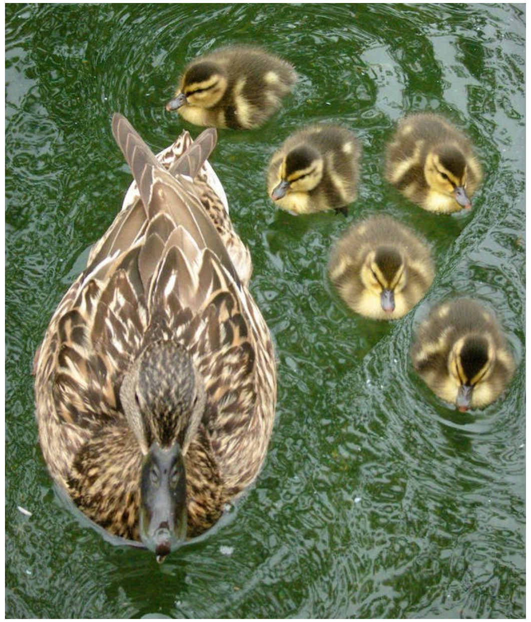
Cropped



Scaled



Carved



Seattle Skyline

Energy of a Seam

- Sum of the energies of each pixel

$$e(p) = \text{energy of pixel } p$$

- Many choices for pixel energy
  - 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”

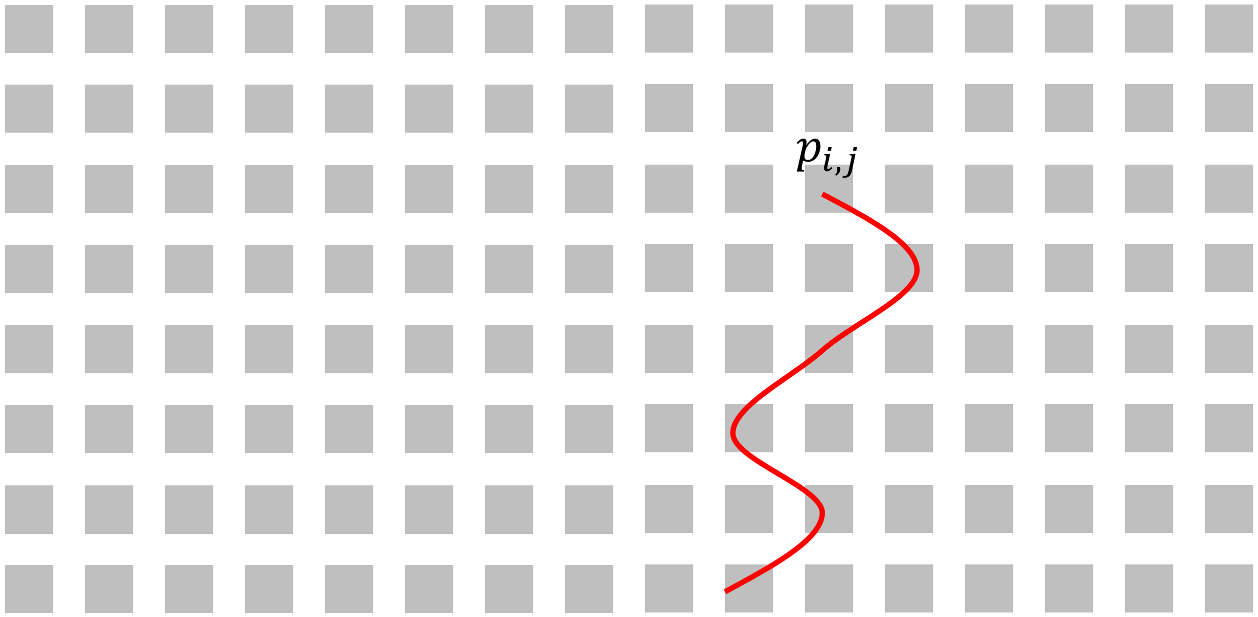
- Goal: find least-energy seam to remove

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

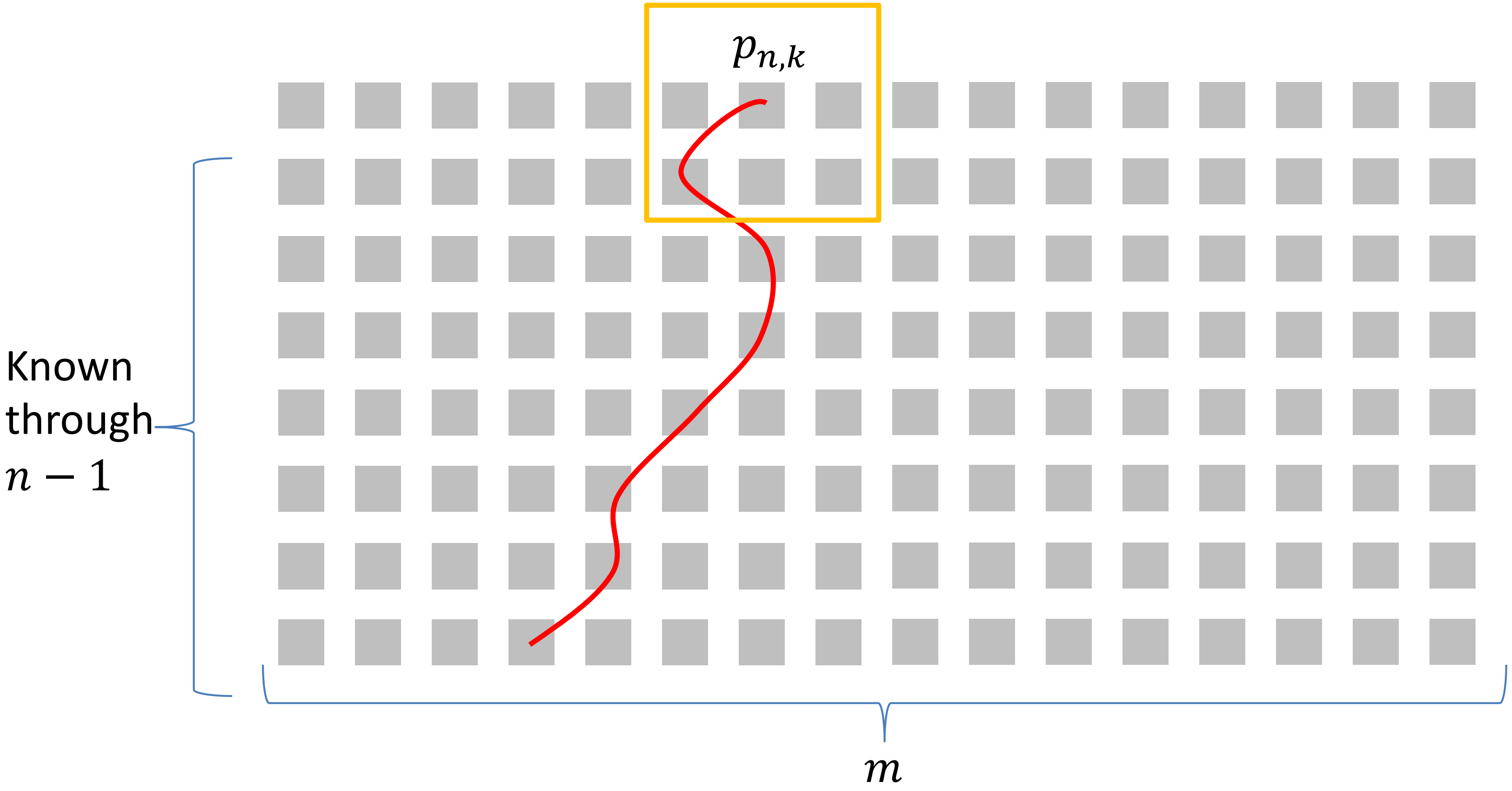
Identify Recursive Structure

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



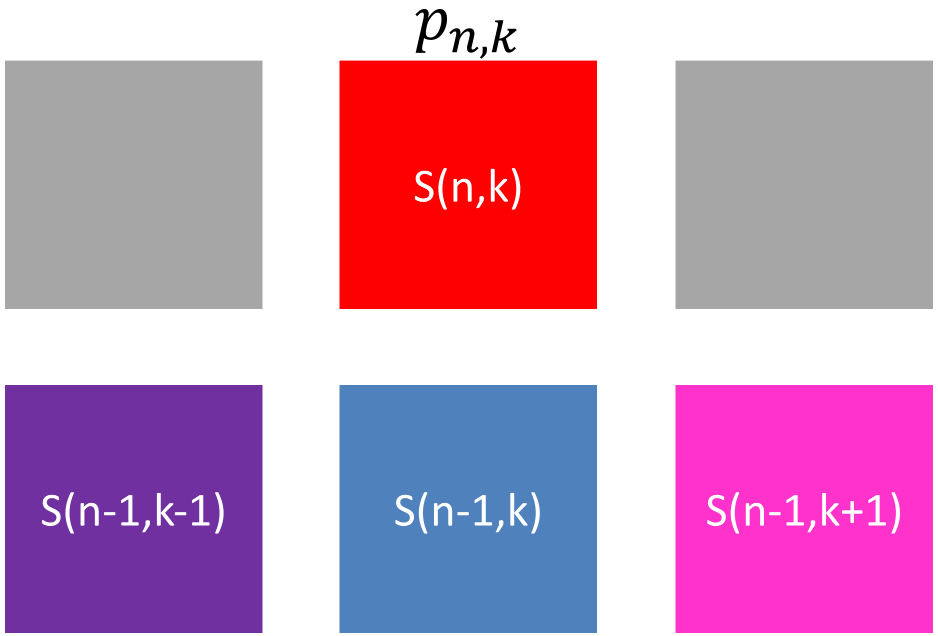
Computing $S(n, k)$

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)



$p_{n,k}$

S(n,k)

S(n-1,k-1)

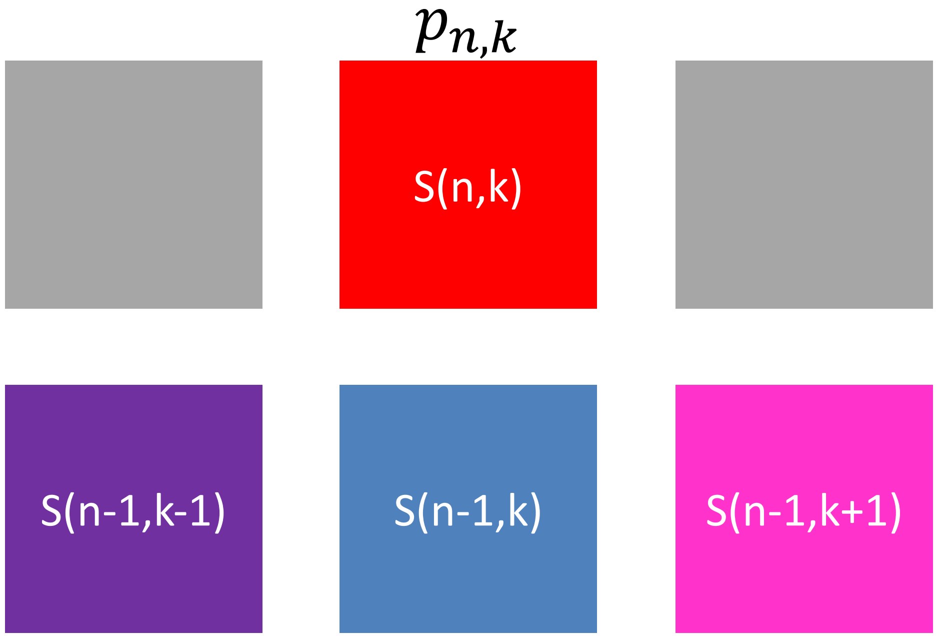
S(n-1,k)

S(n-1,k+1)

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)

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



$p_{n,k}$

S(n,k)

S(n-1,k-1)

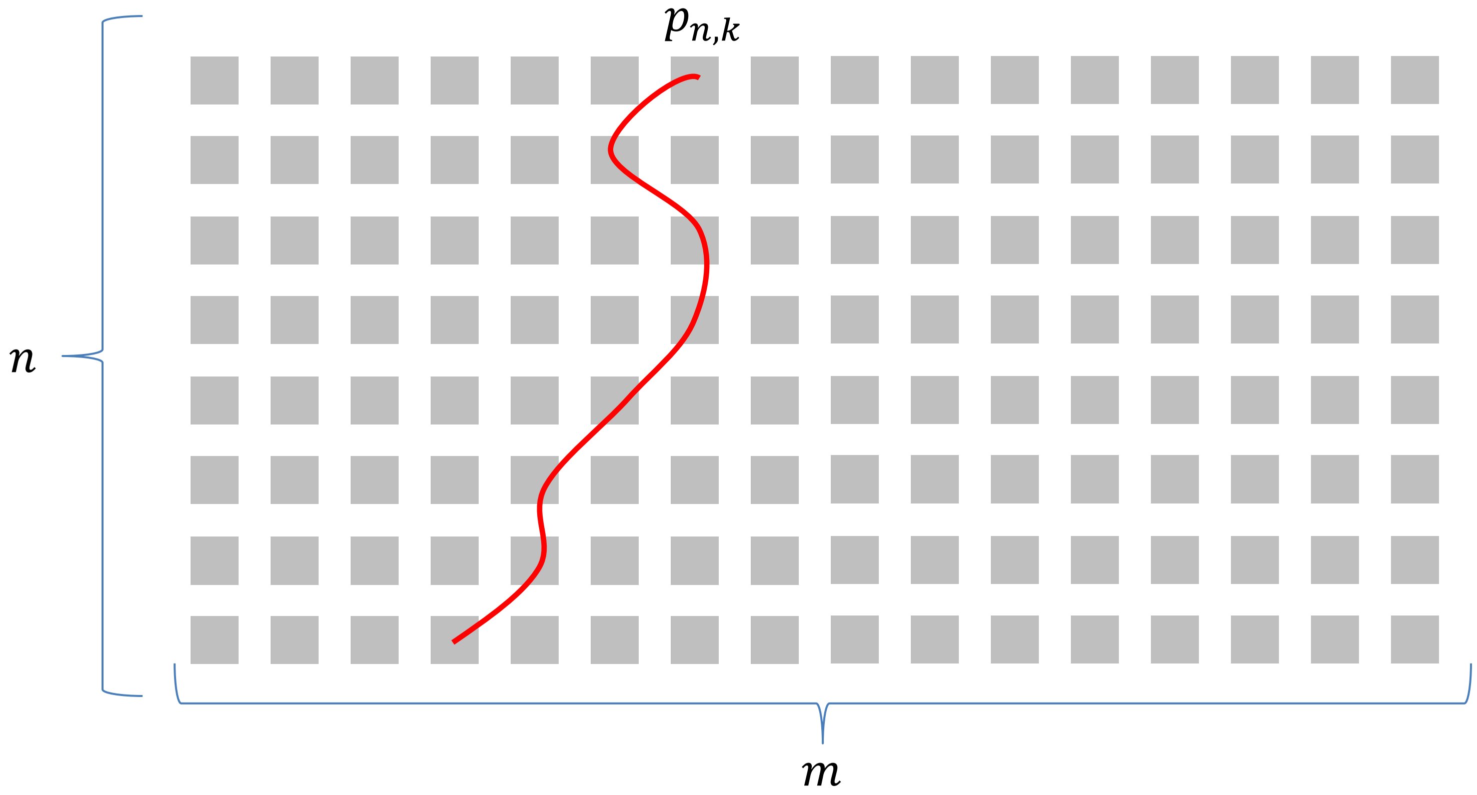
S(n-1,k)

S(n-1,k+1)

Finding the Least Energy Seam

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

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



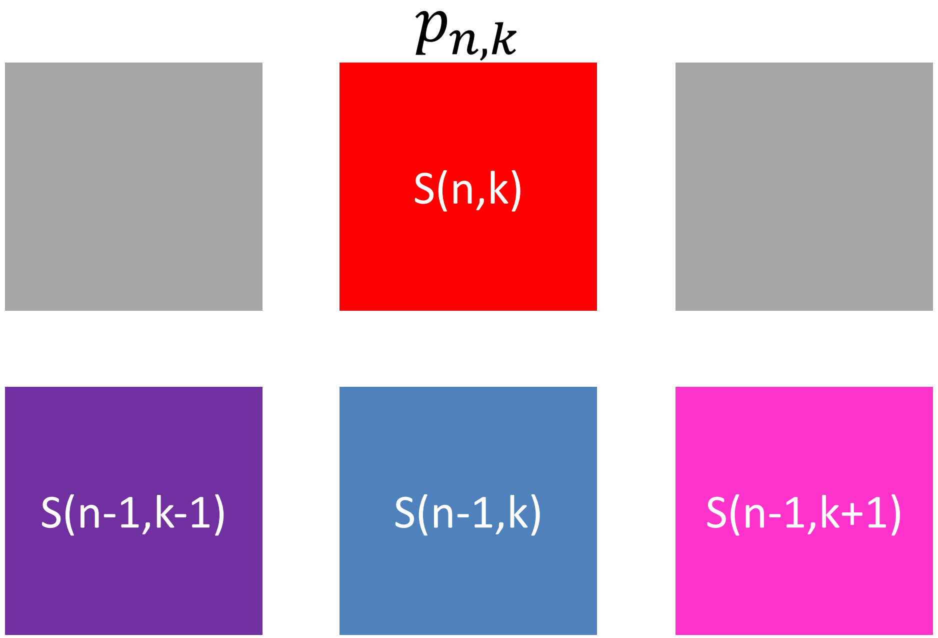
Dynamic Programming

- 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

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)

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



$p_{n,k}$

S(n,k)

S(n-1,k-1)

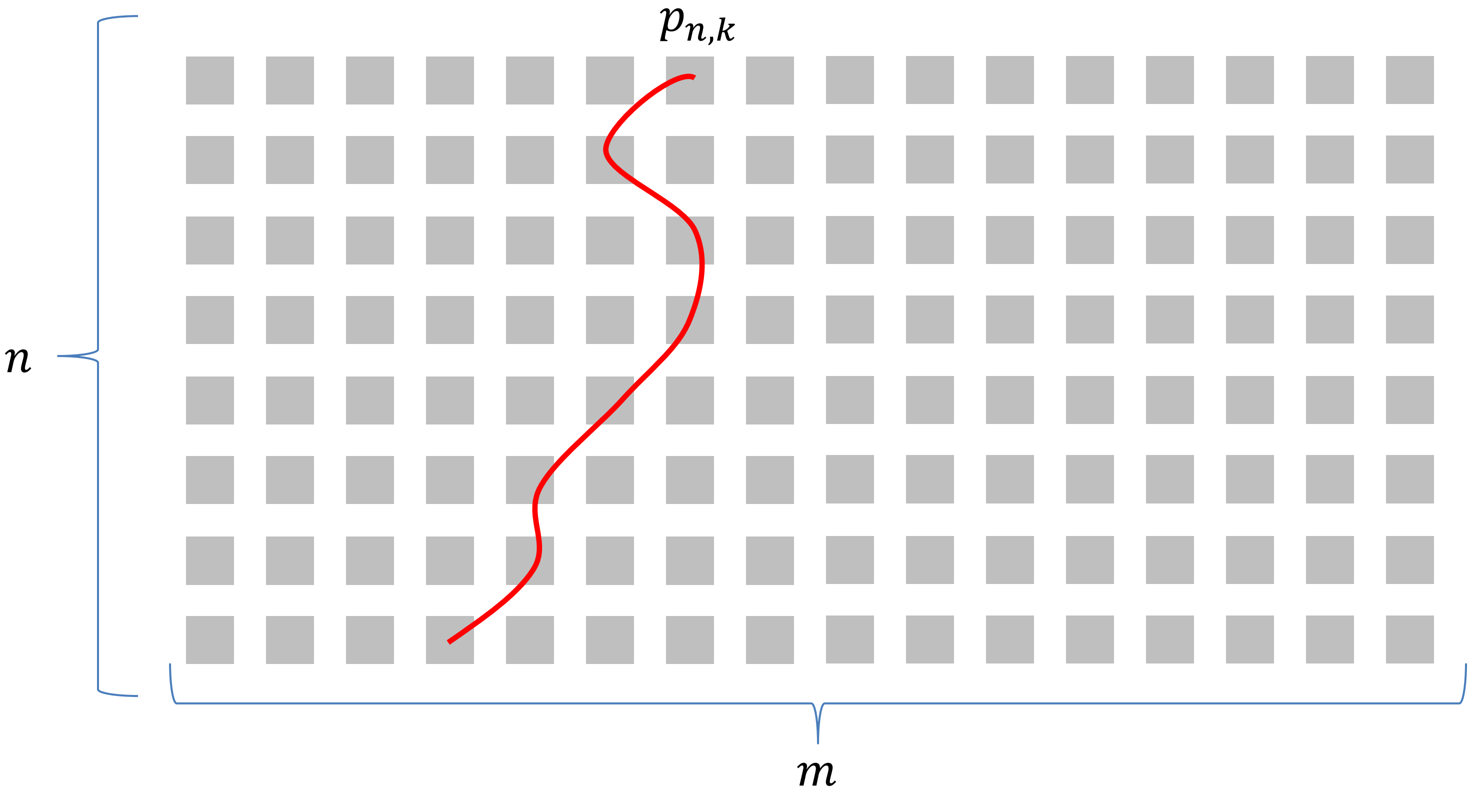
S(n-1,k)

S(n-1,k+1)

Finding the Least Energy Seam

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

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



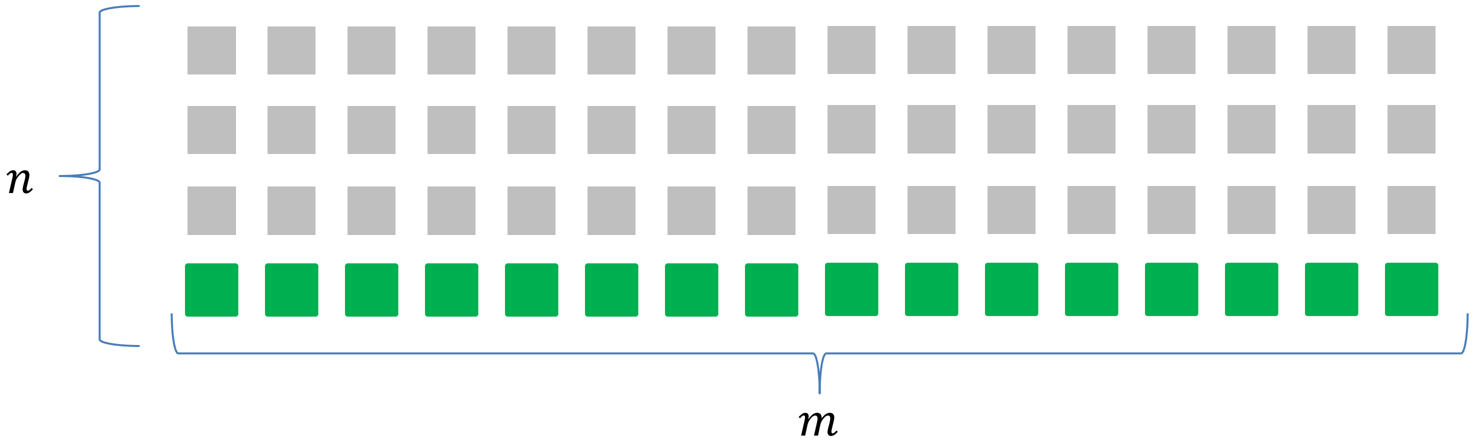
Dynamic Programming

- 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

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



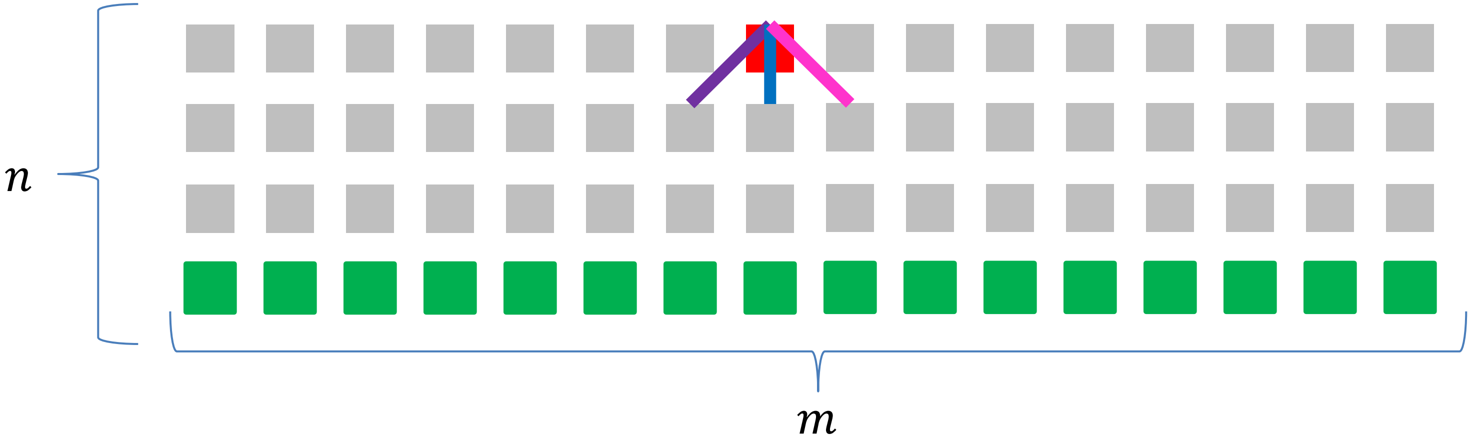
Energy of the seam initialized to the energy of that pixel

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

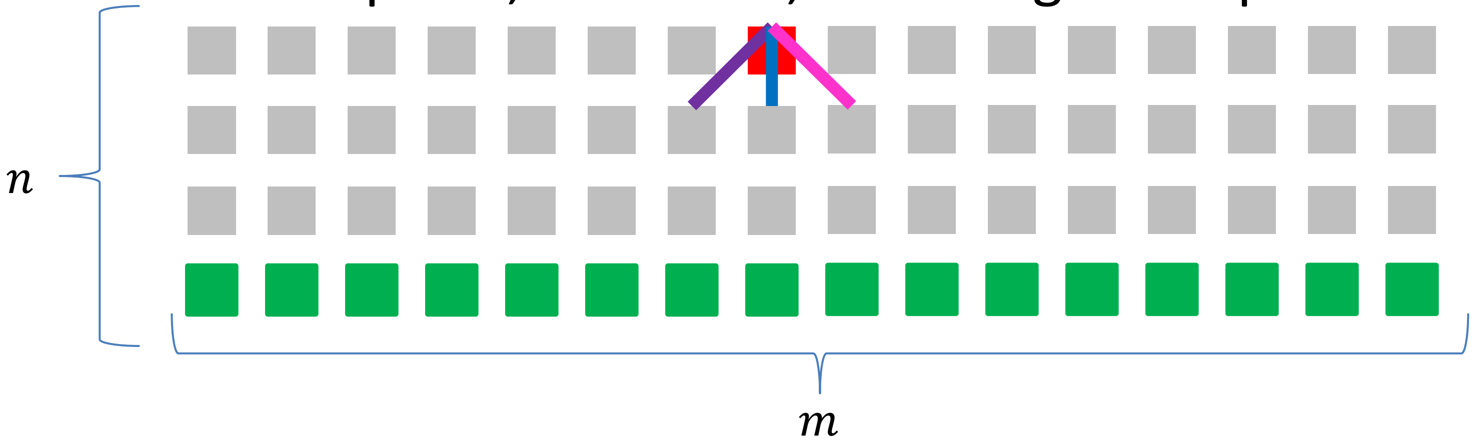
Finding the Seam

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

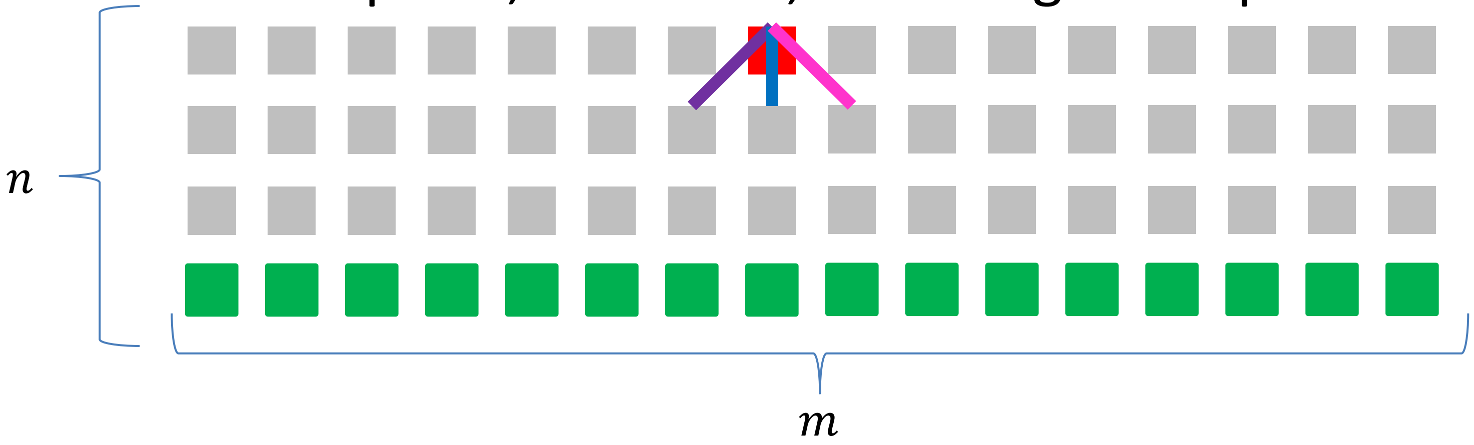
Run Time?

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

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

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

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



Energy of the seam initialized to the energy of that pixel

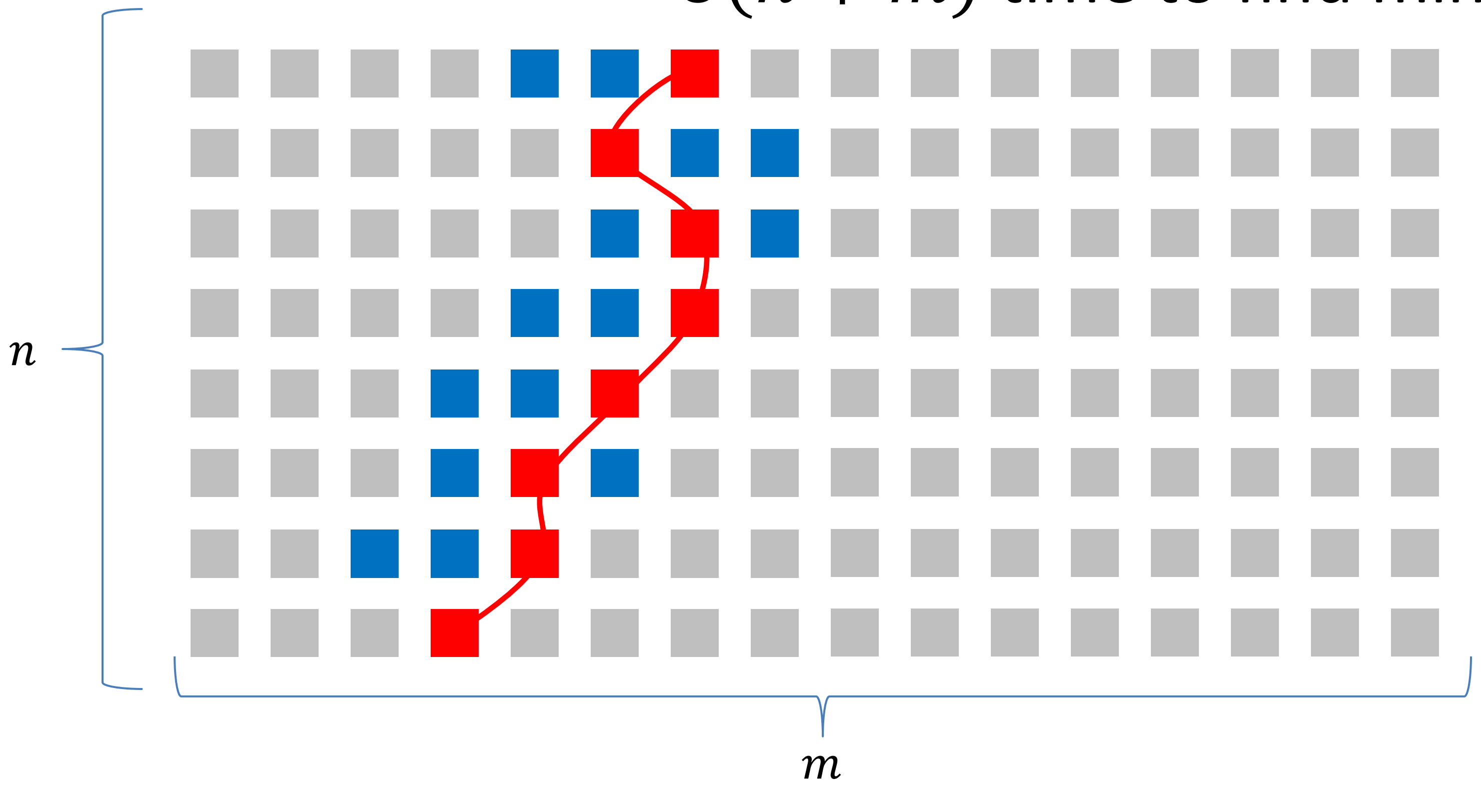
Repeated Seam Removal

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



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

$\Omega(2^n)$

Dynamic Programming

- Requires Optimal Substructure
  - Solution to larger problem is the (optimal) solutions to a smaller one plus one “decision”
- Idea:
  1. Identify the substructure of the problem
     - What are the options for the “last thing” done? What subproblem comes from each?
  2. Save the solution to each subproblem in memory
  3. Select an 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: $X[i] \neq Y[j]$

$X{=}ATCTGCGA$

$Y{=}TGCATAT$

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

$X{=}ATCTGCGT$

$Y{=}TGCATAC$

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

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

Dynamic Programming

- Requires Optimal Substructure
  - Solution to larger problem is the (optimal) solutions to a smaller one plus one "decision"
- Idea:
  1. Identify the substructure of the problem
     - What are the options for the "last thing" done? What subproblem comes from each?
  2. Save the solution to each subproblem in memory
  3. Select an 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: $X[i] \neq Y[j]$

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

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

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

Read from M[i,j] if present

Save to M[i,j]

```
X = “alkjdflaksjdf”
Y = “lakjsdflkasjdlfs”
M = 2d array of len(X) rows and len(Y) columns, initialized to -1
def LCS(int i, int j):
        # returns the length of the LCS shared between the length-i prefix of  X and length-j prefix of Y
        # memoization
        if M[i,j] > -1:
                return M[i,j]
        #base case:
        if i == 0 or j == 0:
                ans = 0
        elif X[i] == Y[j]:
                ans = LCS(i-1, j-1) + 1
        else:
                ans = max( LCS(i, j-1), LCS(i-1, j) )
        M[i,j] = ans
        return ans
print(LCS(len(X), len(Y))) # the answer for the entirety of X and Y
```

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

Dynamic Programming

- Requires Optimal Substructure
  - Solution to larger problem is the (optimal) solutions to a smaller one plus one "decision"
- Idea:
  1. Identify the substructure of the problem
     - What are the options for the "last thing" done? What subproblem comes from each?
  2. Save the solution to each subproblem in memory
  3. Select an order for solving subproblems
     - "Top Down": Solve each recursively
     - "Bottom Up": Iteratively solve smallest to largest

3. Solve in a Good Order

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

	$X =$		A	T	C	T	G	A	T
$Y =$		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

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)

Run Time?

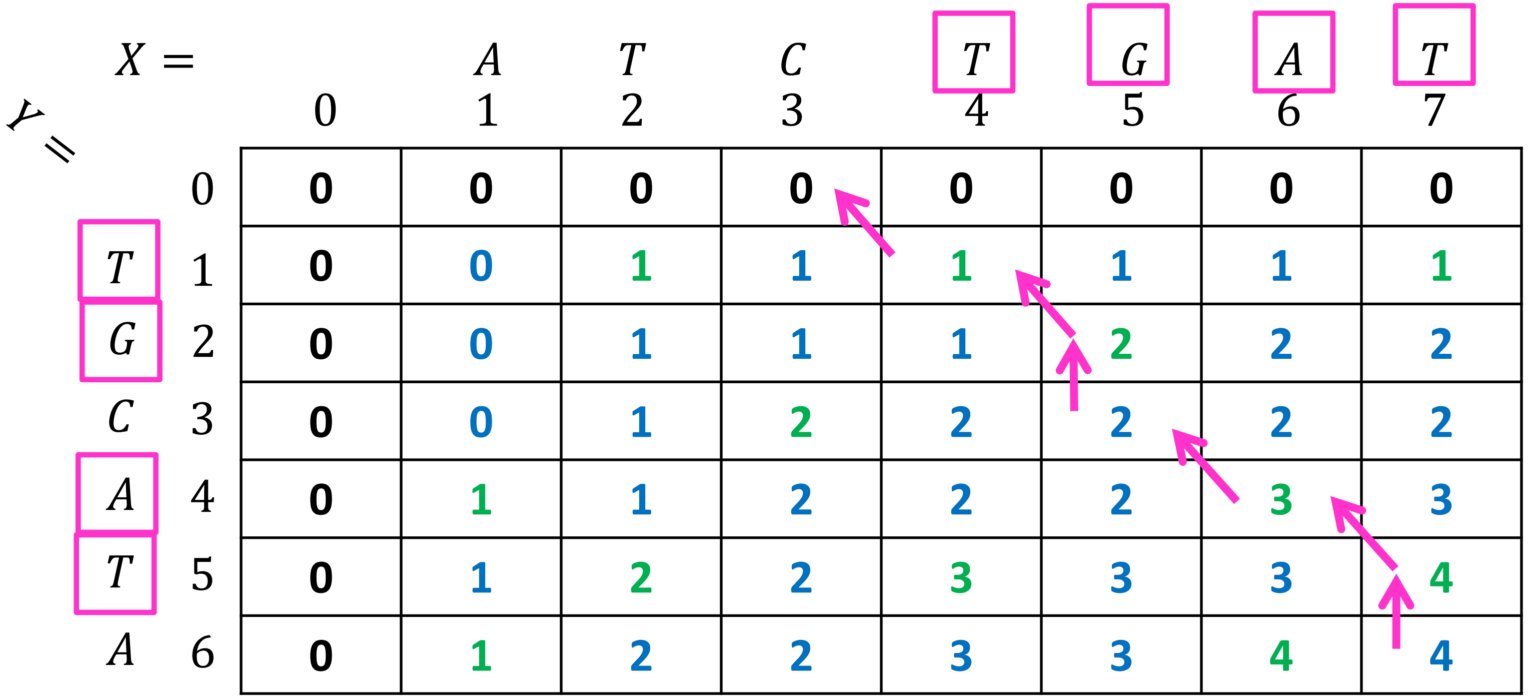
$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i-1, j-1) + 1 & \text{if } X[i] = Y[j] \\ \max(LCS(i, j-1), LCS(i-1, j)) & \text{otherwise} \end{cases}$$

$X =$			A	T	C	T	G	A	T
$Y =$		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

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

Reconstructing the LCS

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



$Y =$	$X =$	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

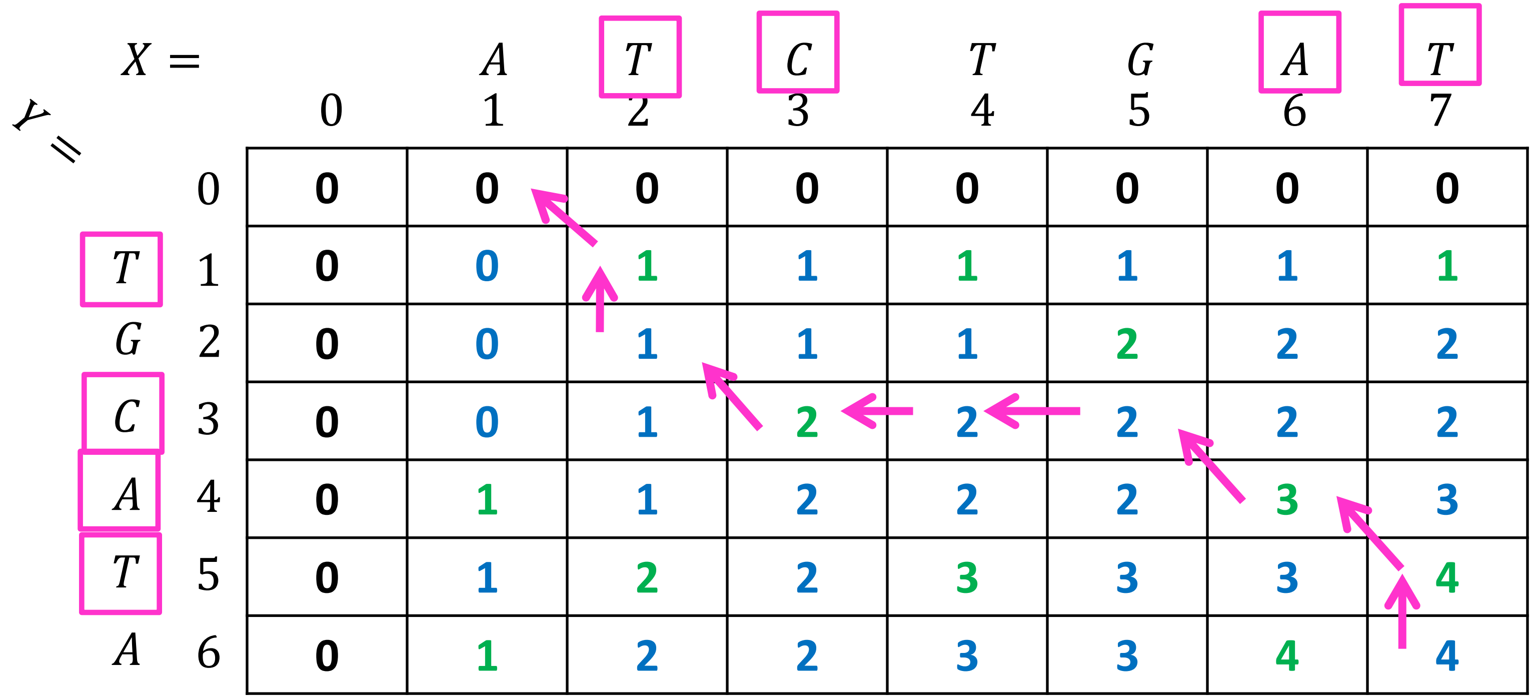
Start from bottom right,

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	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

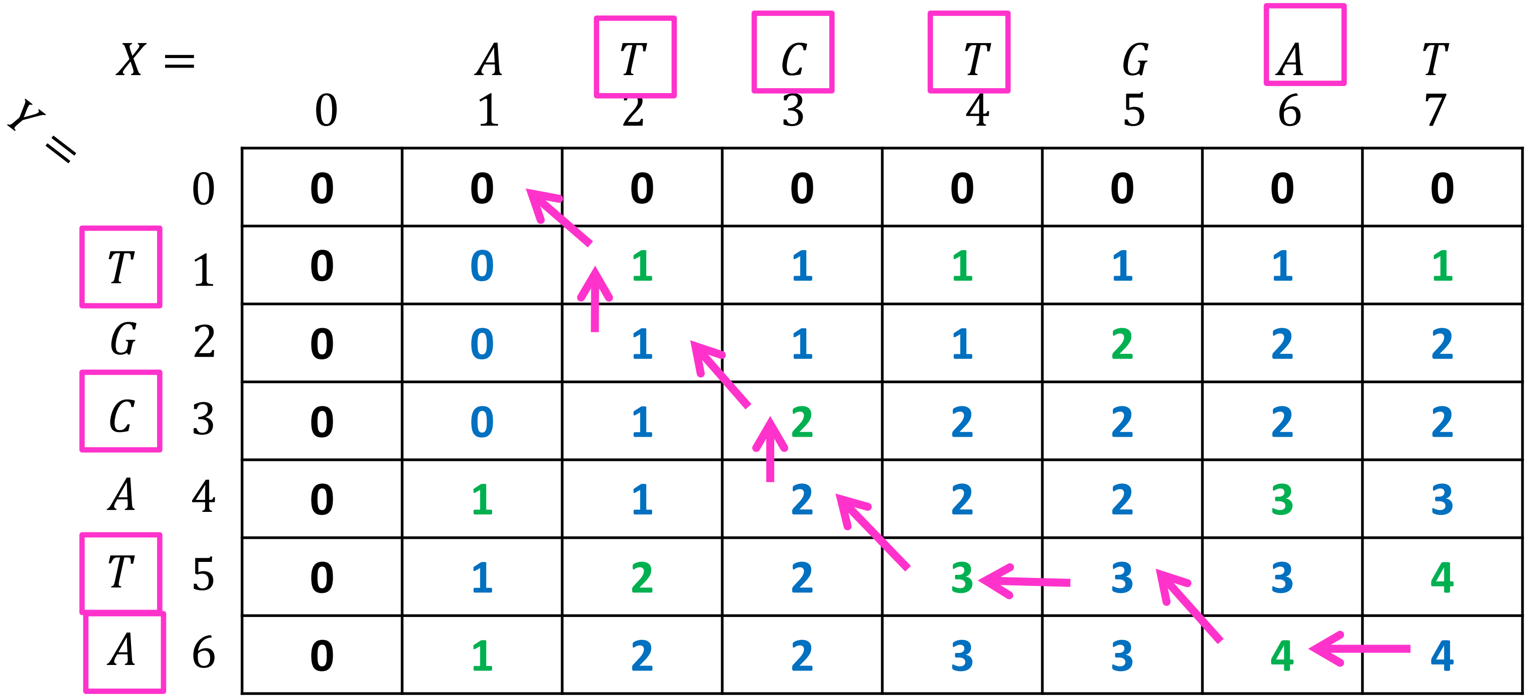
Start from bottom right,

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	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