

CS 3100

Data Structures and Algorithms 2

Lecture 16: Dynamic Programming

Co-instructors: Robbie Hott and Tom Horton
Fall 2023

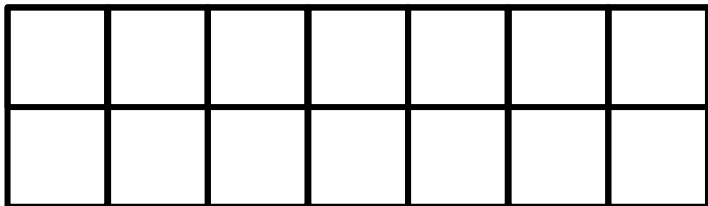
Readings in CLRS 4th edition:

- Chapter 14

Warm Up

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

How many ways to tile this:



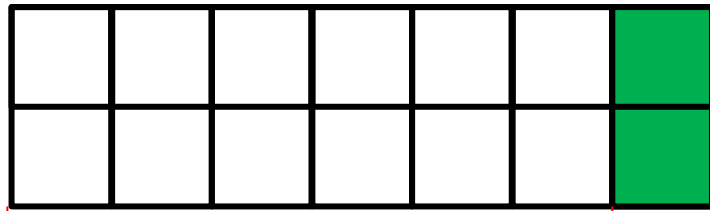
With these?



Warm Up

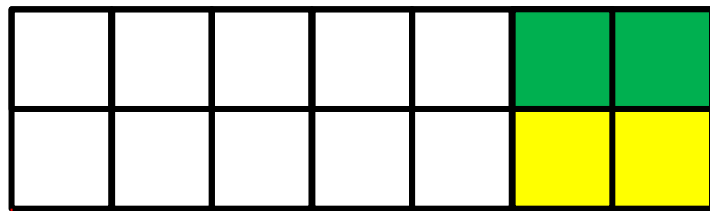
How many ways are there to tile a $2 \times n$ board with dominoes?

Two ways to fill the final column:



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

$n-1$



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

$n-2$

Announcements

- Upcoming dates
 - PS3 (Greedy Algorithms) due October 20, 2023 at 11:59pm
 - PA3 (Clustering) due October 29, 2023 at 11:59pm
- Updated Late Policy!
 - You must submit an extension request **before** the deadline
 - Explain why need you need the extension (up to 48 hours past the deadline)
 - Acknowledge that you're getting an extension
 - The late deadline is not the real deadline 😊
 - You may then take the additional 48 hours as needed
- Course email (comes to both professors and head TAs):

cs3100@cshelpdesk.atlassian.net

How to compute $Tile(n)$?

Tile(n):

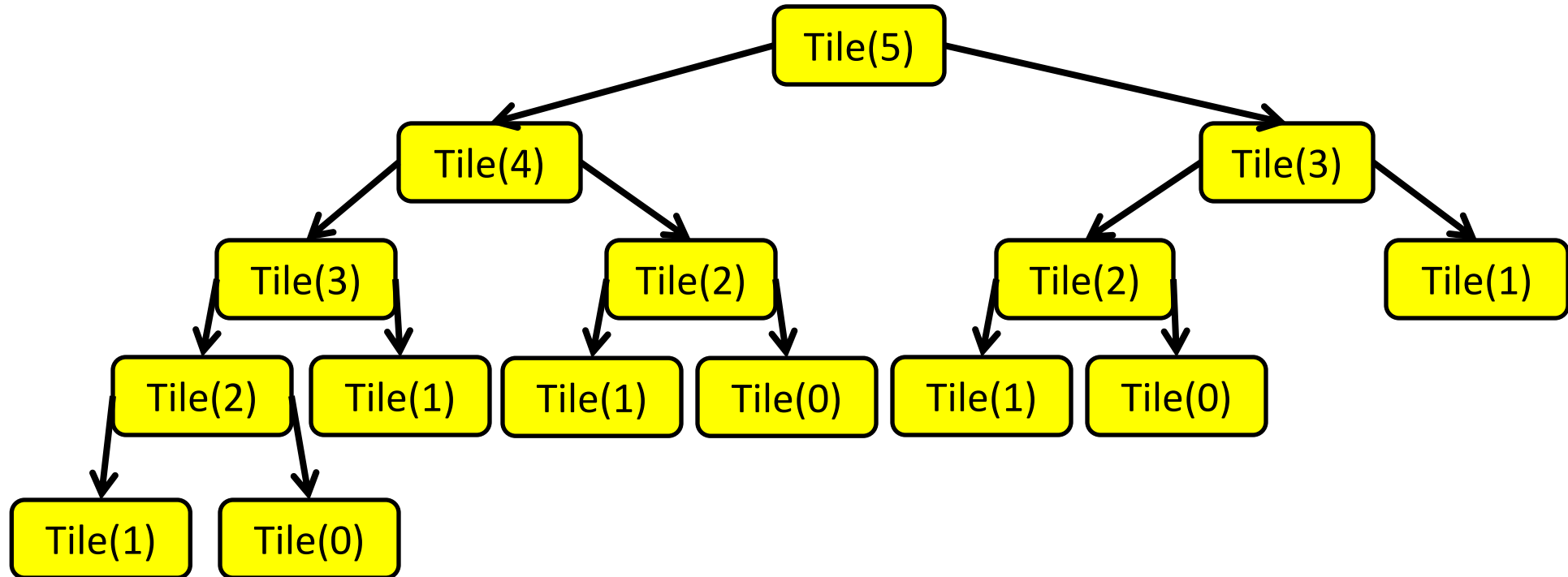
if $n < 2$:

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 $n < 2$:

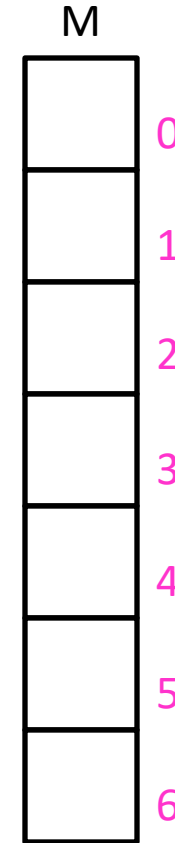
return 1

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]

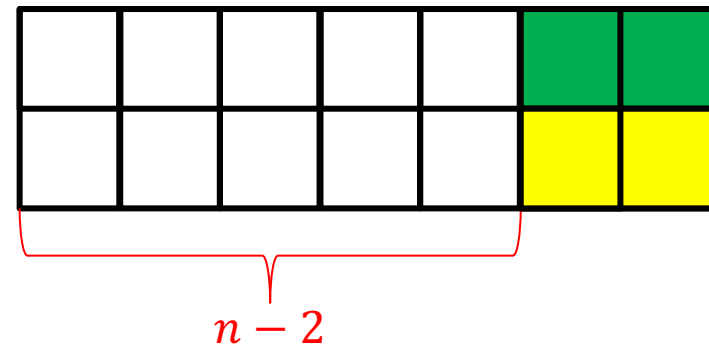
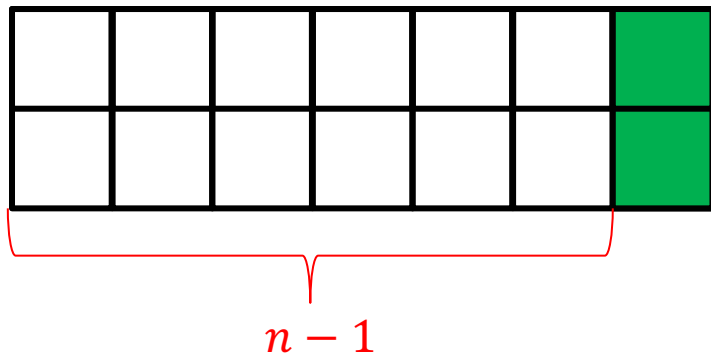
$M[n] = Tile(n-1) + Tile(n-2)$

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(solutions)  
  
    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(solutions)  
    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

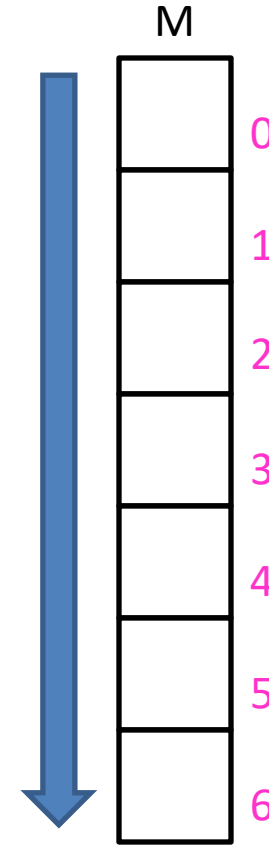
$M[0] = 1$

$M[1] = 1$

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 (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 ℓ_1, \dots, ℓ_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

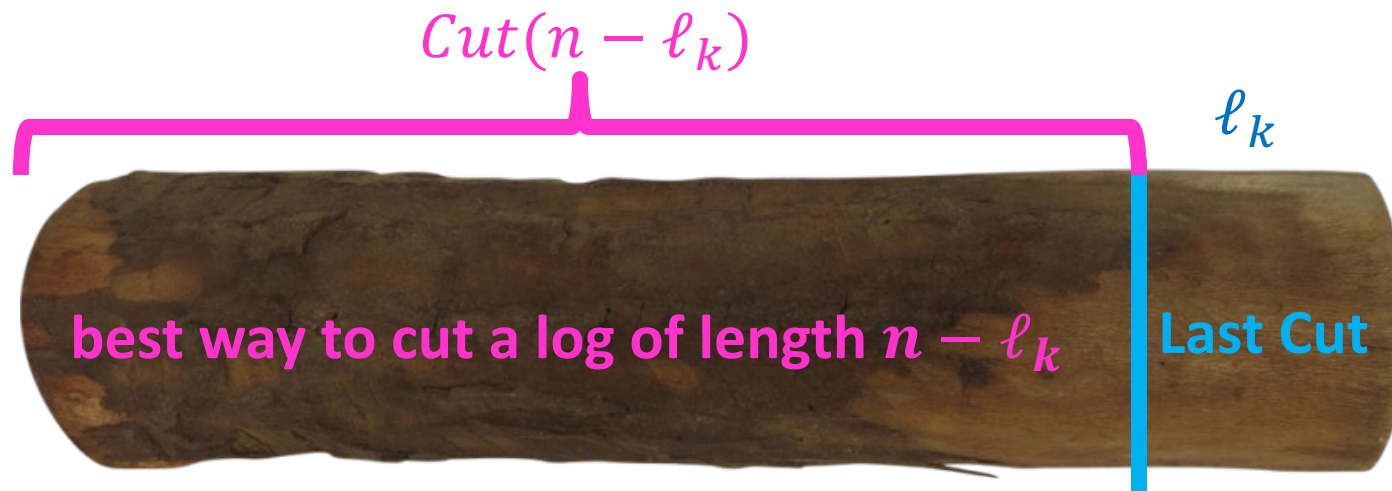
- Requires **Optimal Substructure**
 - Solution to larger problem contains the solutions to smaller ones
- Idea:
 1. Identify the recursive structure of the problem
 - What is the “last thing” done?
 2. Save the solution to each subproblem in memory
 3. Select a good order for solving subproblems
 -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

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



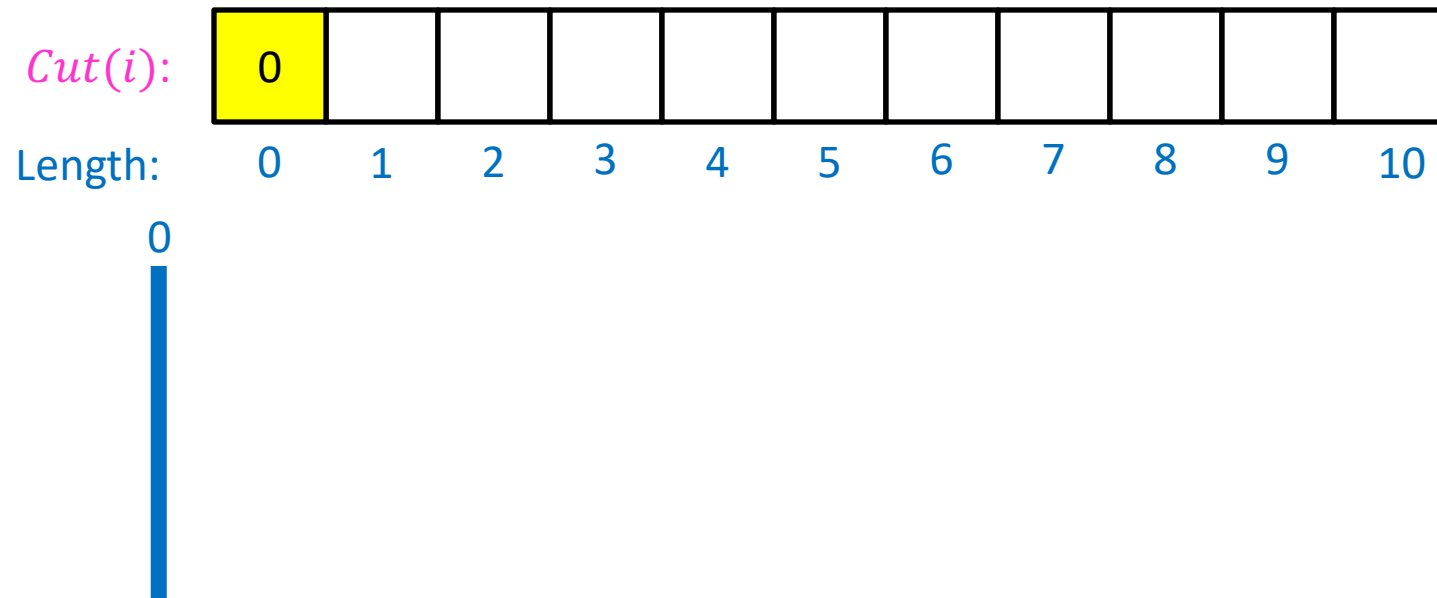
Dynamic Programming

- 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

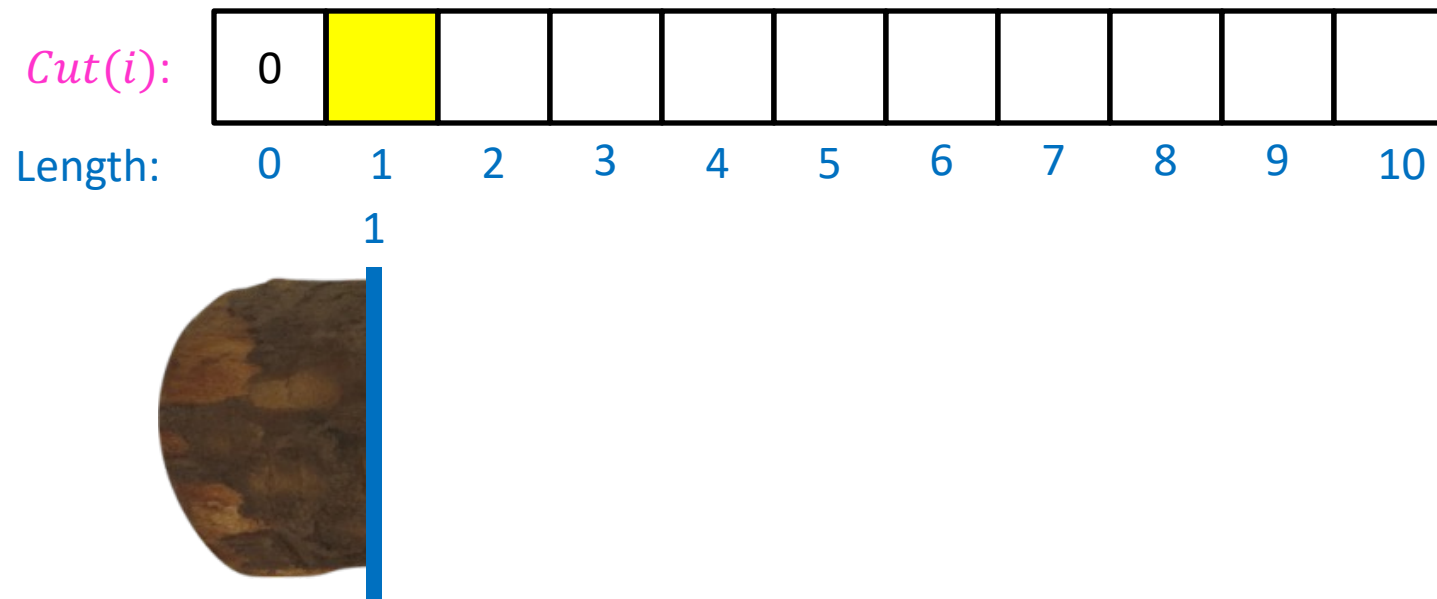
$$\text{Cut}(0) = 0$$



3. Select a Good Order for Solving Subproblems

Solve Smallest subproblem first

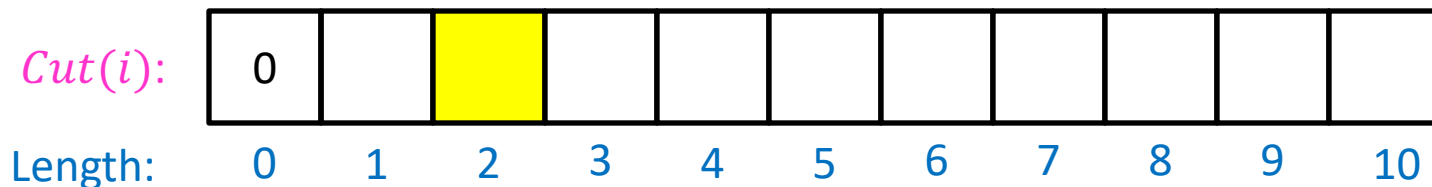
$$Cut(1) = Cut(0) + P[1]$$



3. Select a Good Order for Solving Subproblems

Solve Smallest subproblem first

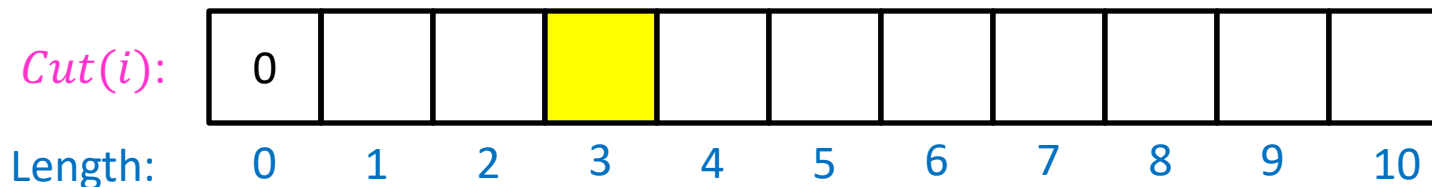
$$Cut(2) = \max \begin{cases} Cut(1) + P[1] \\ Cut(0) + P[2] \end{cases}$$



3. Select a Good Order for Solving Subproblems

Solve Smallest subproblem first

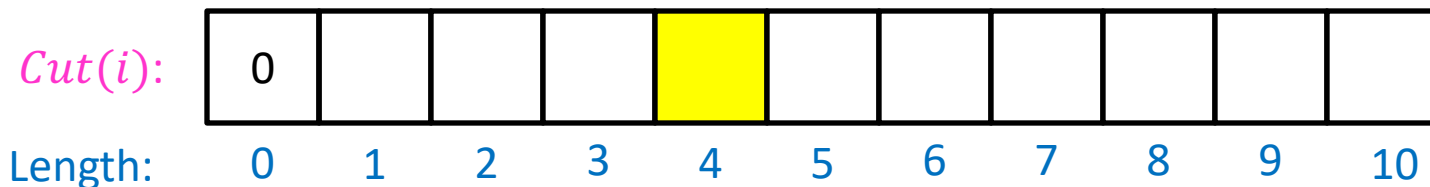
$$Cut(3) = \max \begin{cases} Cut(2) + P[1] \\ Cut(1) + P[2] \\ Cut(0) + P[3] \end{cases}$$



3. Select a Good Order for Solving Subproblems

Solve Smallest subproblem first

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



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

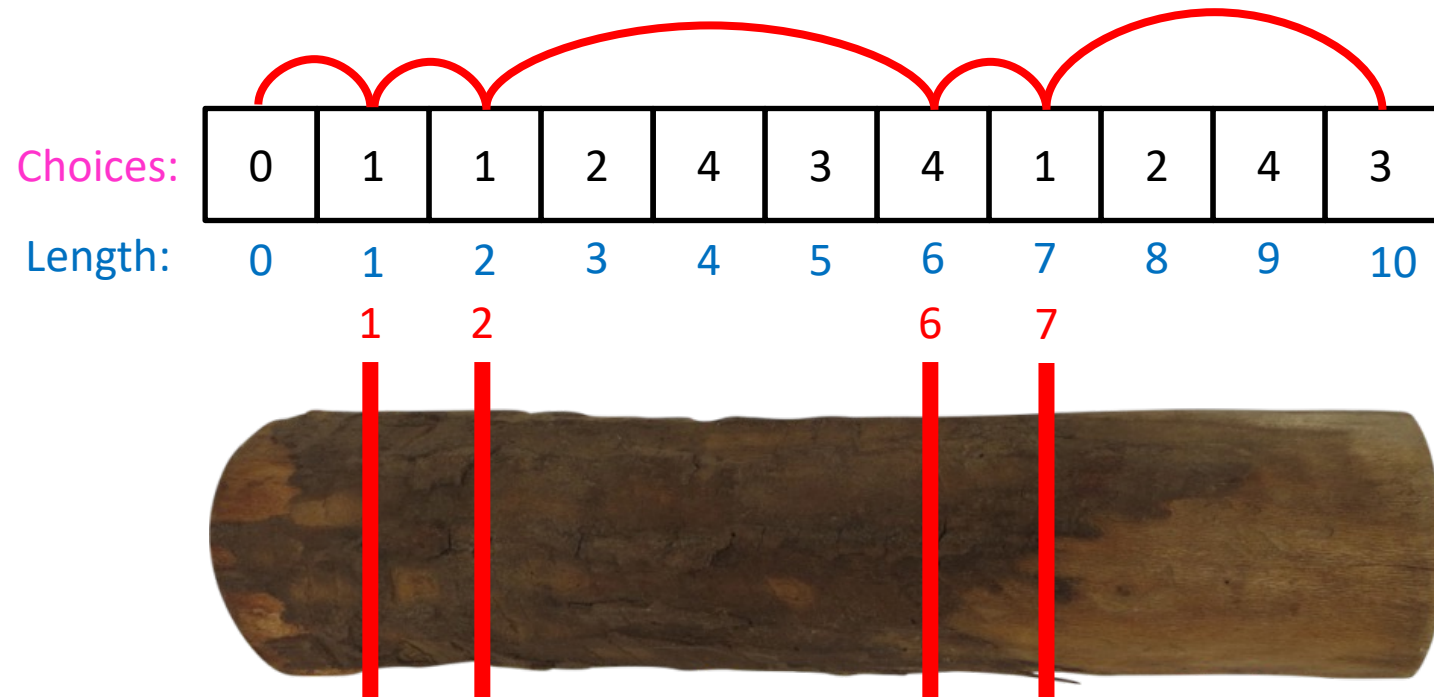
Gives the size
of the last cut

$C[i] = best$

return C[n]

Reconstruct the Cuts

- Backtrack through the choices



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

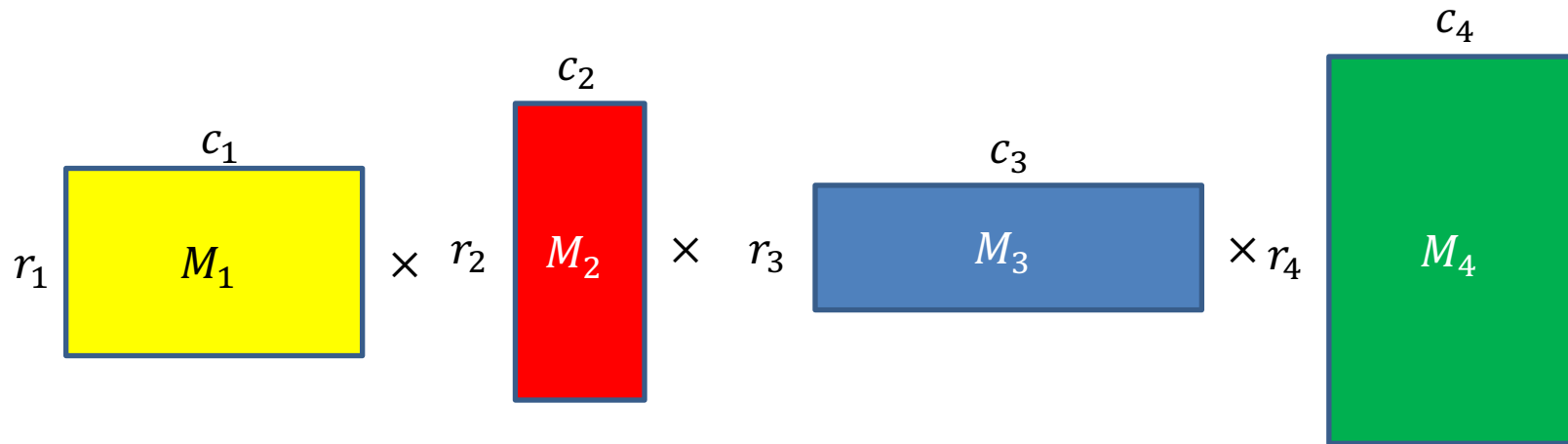
- If n were 5
 - Best score is 13
 - Cut at Choice[n]=2, then cut at
Choice[n-Choice[n]]= Choice[5-2]= Choice[3]=3
- If n were 7
 - Best score is 18
 - Cut at 1, then cut at 6

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

Matrix Chaining

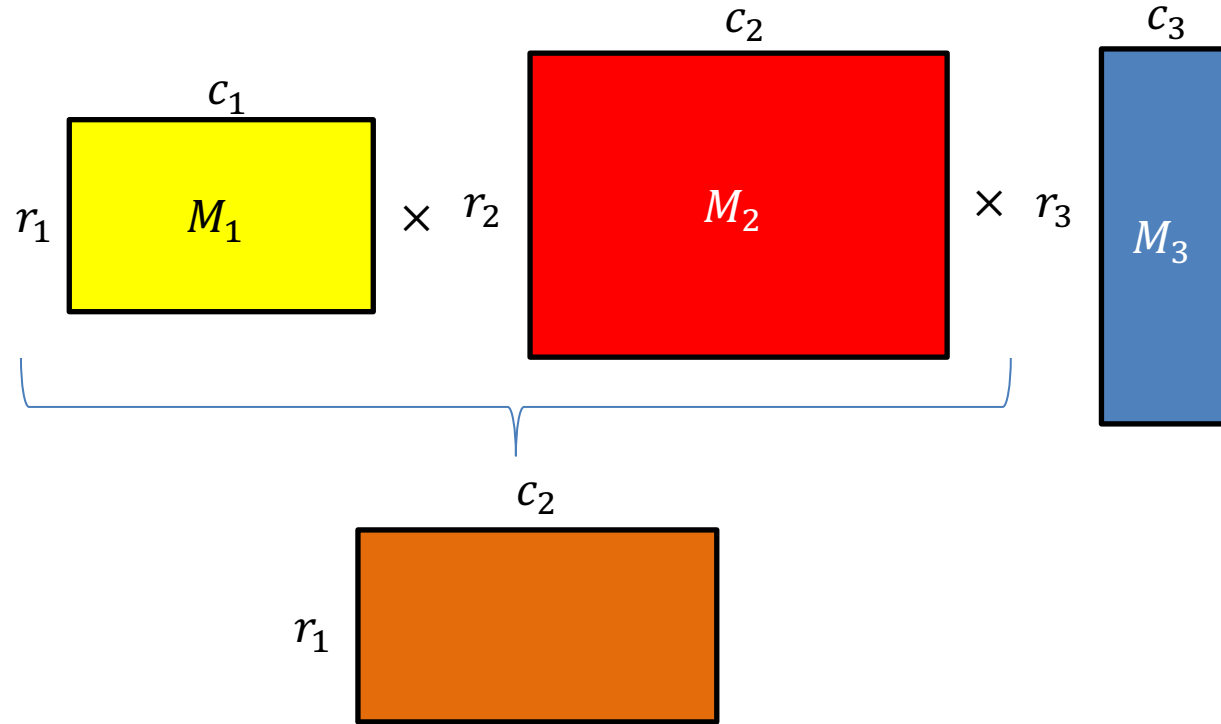
- Given a sequence of Matrices (M_1, \dots, M_n) , what is the most efficient way to multiply them?



Order Matters!

$$c_1 = r_2$$

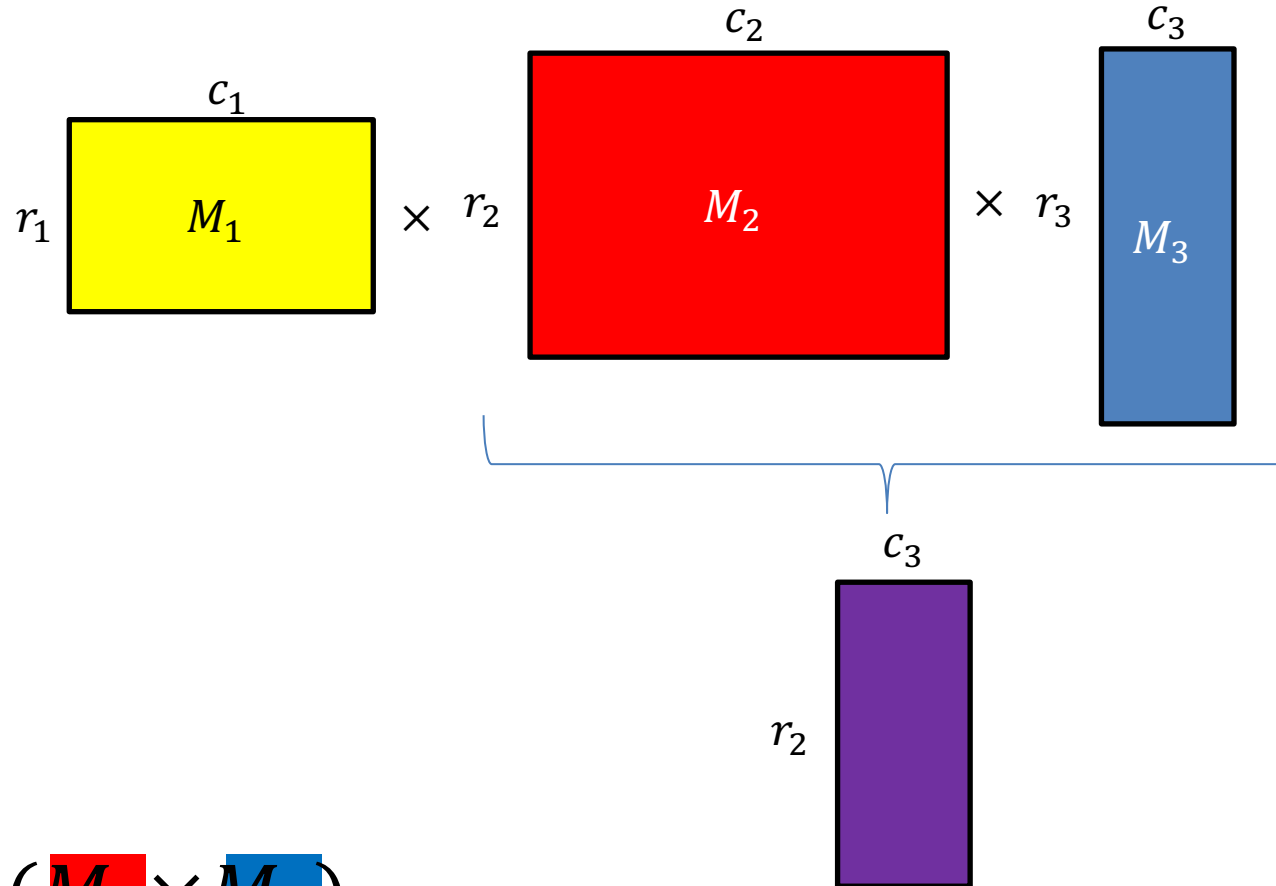
$$c_2 = r_3$$



- $(M_1 \times M_2) \times M_3$
 - uses $(c_1 \cdot r_1 \cdot c_2) + c_2 \cdot r_1 \cdot c_3$ operations

Order Matters!

$$c_1 = r_2$$
$$c_2 = r_3$$



- $M_1 \times (M_2 \times M_3)$
 - uses $c_1 \cdot r_1 \cdot c_3 + (c_2 \cdot r_2 \cdot c_3)$ operations

Order Matters!

$$c_1 = r_2$$

$$c_2 = r_3$$

- $(M_1 \times M_2) \times M_3$

- uses $(c_1 \cdot r_1 \cdot c_2) + c_2 \cdot r_1 \cdot c_3$ operations

- $(10 \cdot 7 \cdot 20) + 20 \cdot 7 \cdot 8 = 2520$

- $M_1 \times (M_2 \times M_3)$

- uses $c_1 \cdot r_1 \cdot c_3 + (c_2 \cdot r_2 \cdot c_3)$ operations

- $10 \cdot 7 \cdot 8 + (20 \cdot 10 \cdot 8) = 2160$

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

$$c_1 = 10$$

$$c_2 = 20$$

$$c_3 = 8$$

$$r_1 = 7$$

$$r_2 = 10$$

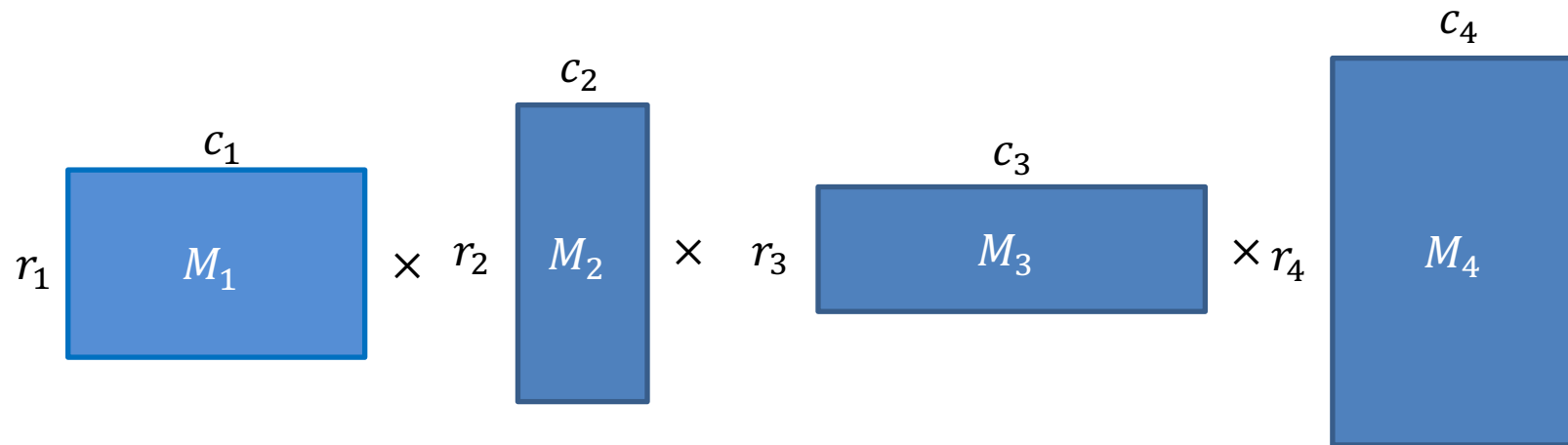
$$r_3 = 20$$

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

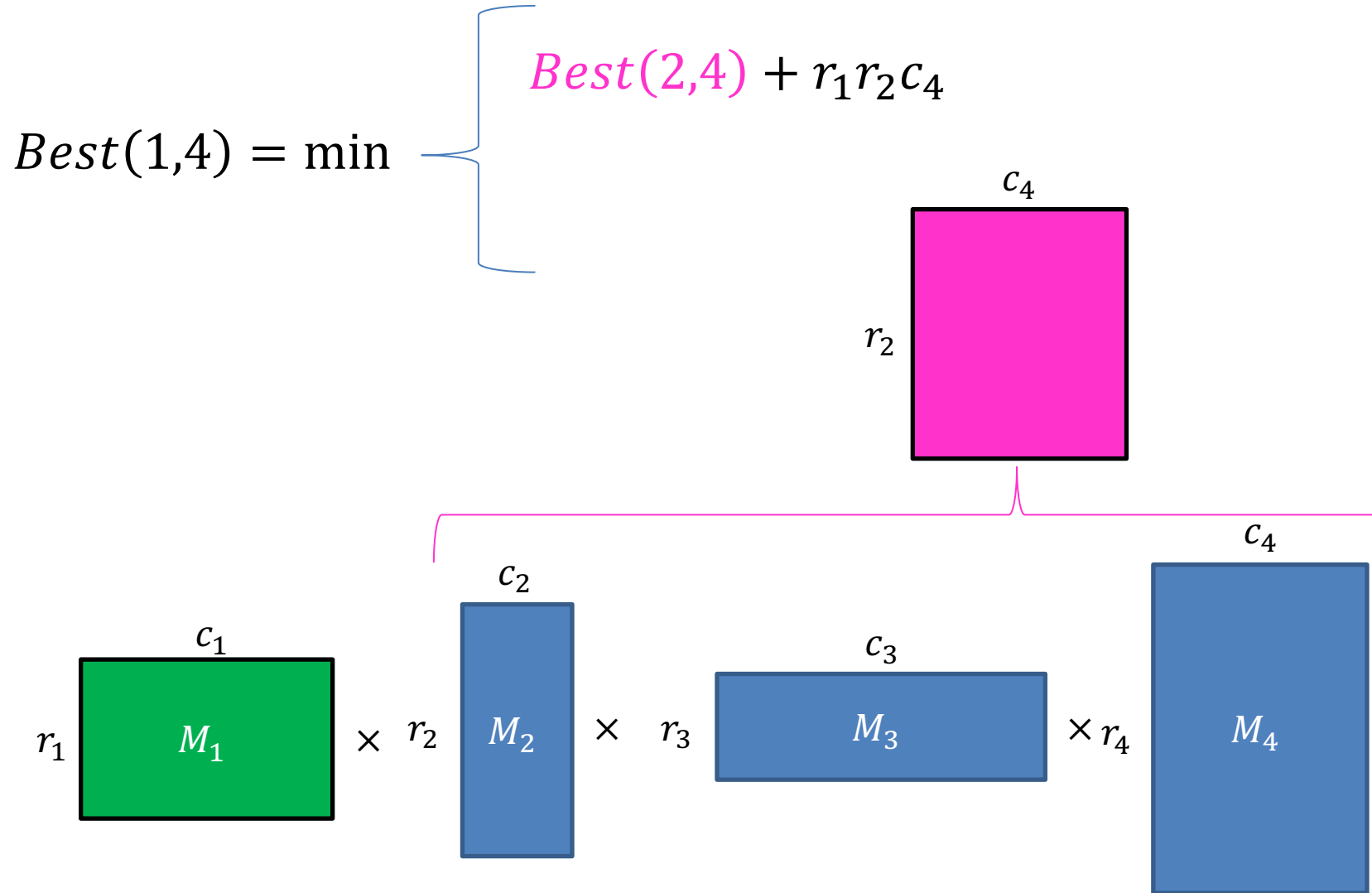
1. Identify the Recursive Structure of the Problem

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



1. Identify the Recursive Structure of the Problem

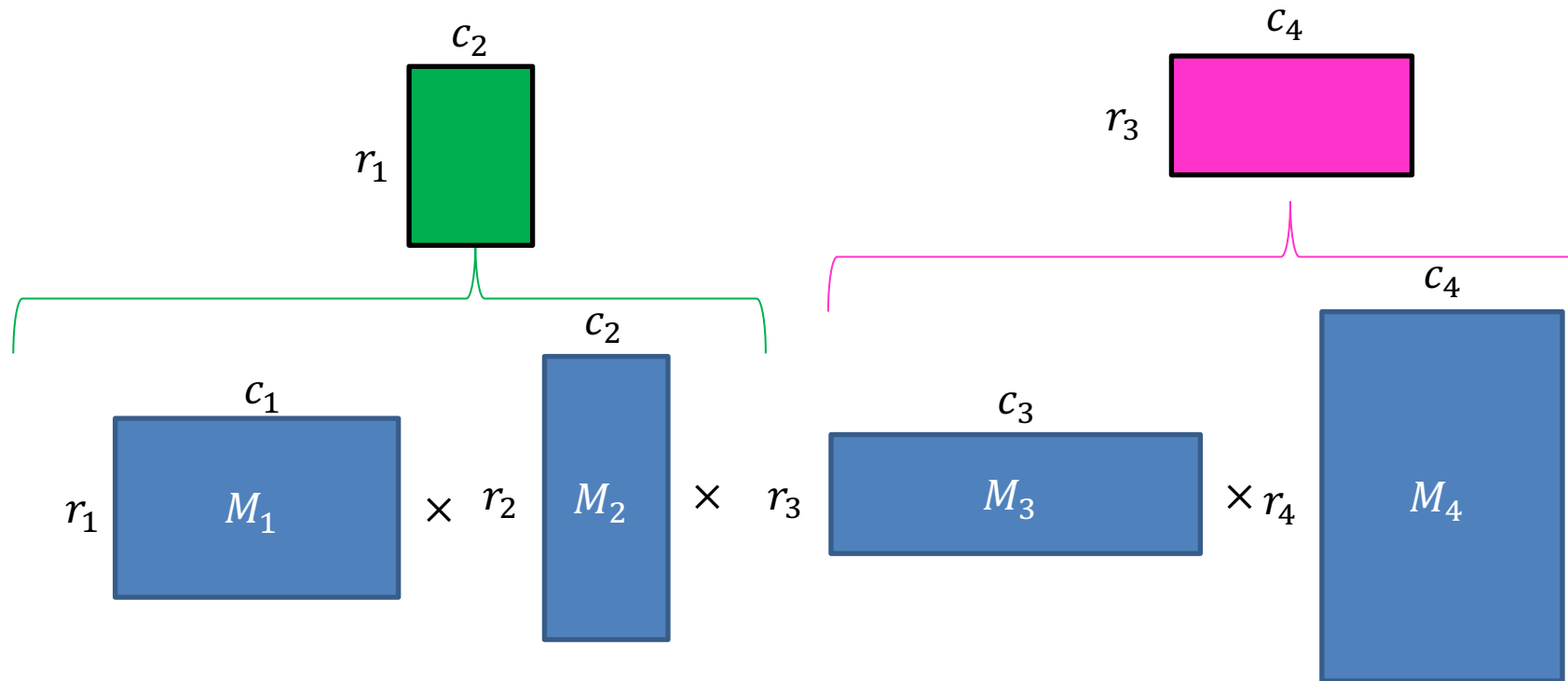
$Best(1, n)$ = cheapest way to multiply together M_1 through M_n



1. Identify the Recursive Structure of the Problem

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

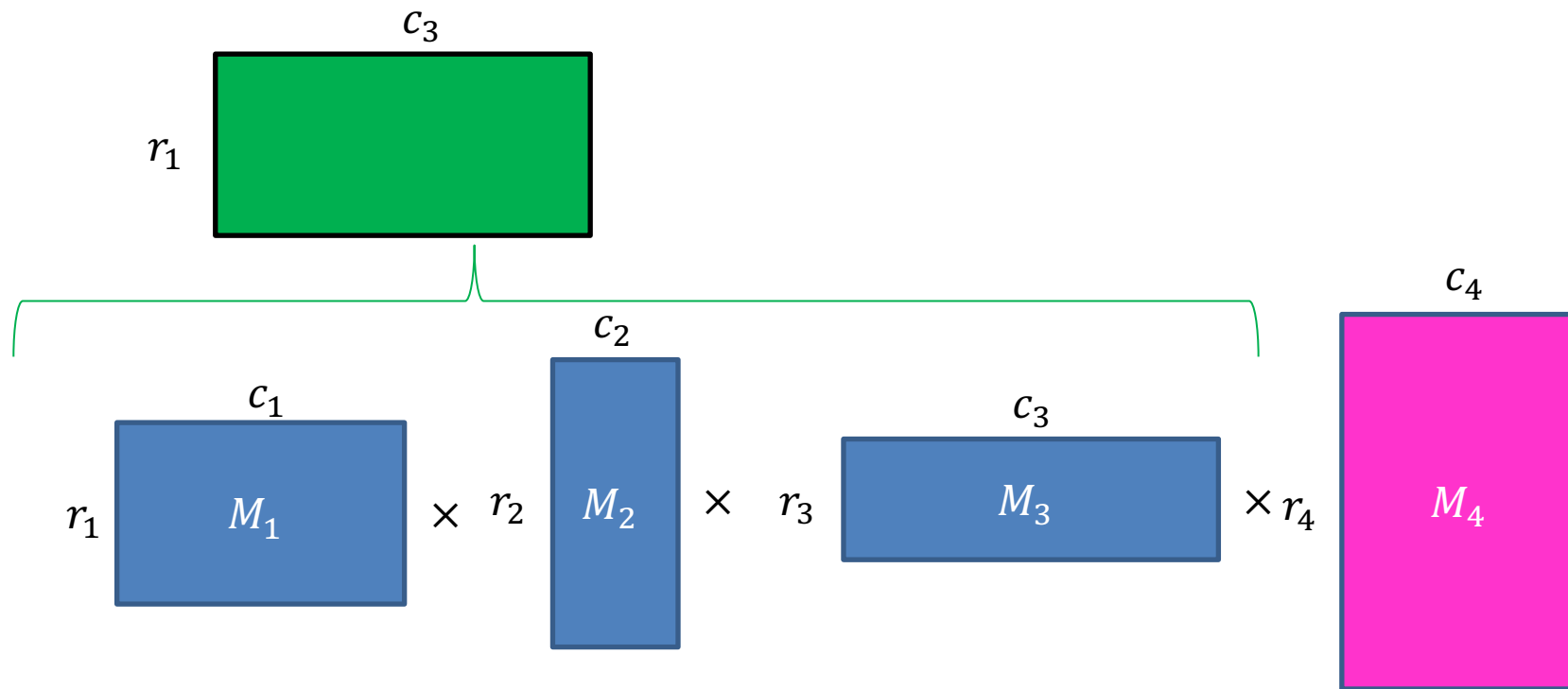
$$Best(1,4) = \min \left\{ \begin{array}{l} Best(2,4) + r_1 r_2 c_4 \\ Best(1,2) + Best(3,4) + r_1 r_3 c_4 \end{array} \right.$$



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

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

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

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

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

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

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

Save to $M[n]$

Read from $M[n]$
if present

$$Best(1, n) = \min$$

$$Best(2, n) + r_1 r_2 c_n$$

$$Best(1, 2) + Best(3, n) + r_1 r_3 c_n$$

$$Best(1, 3) + Best(4, n) + r_1 r_4 c_n$$

$$Best(1, 4) + Best(5, n) + r_1 r_5 c_n$$

...

$$Best(1, n-1) + r_1 r_n c_n$$

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

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

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

Save to $M[n]$

Read from $M[n]$
if present

$$Best(1, n) = \min$$

$$Best(2, n) + r_1 r_2 c_n$$

$$Best(1, 2) + Best(3, n) + r_1 r_3 c_n$$

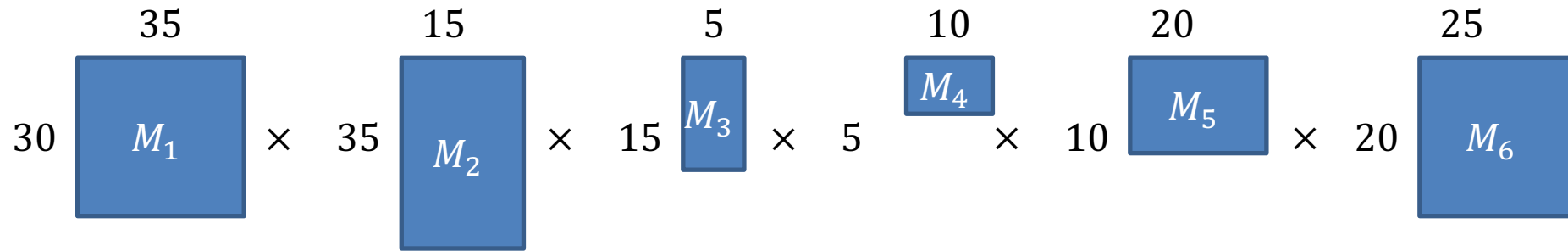
$$Best(1, 3) + Best(4, n) + r_1 r_4 c_n$$

$$Best(1, 4) + Best(5, n) + r_1 r_5 c_n$$

...

$$Best(1, n-1) + r_1 r_n c_n$$

3. Select a good order for solving subproblems

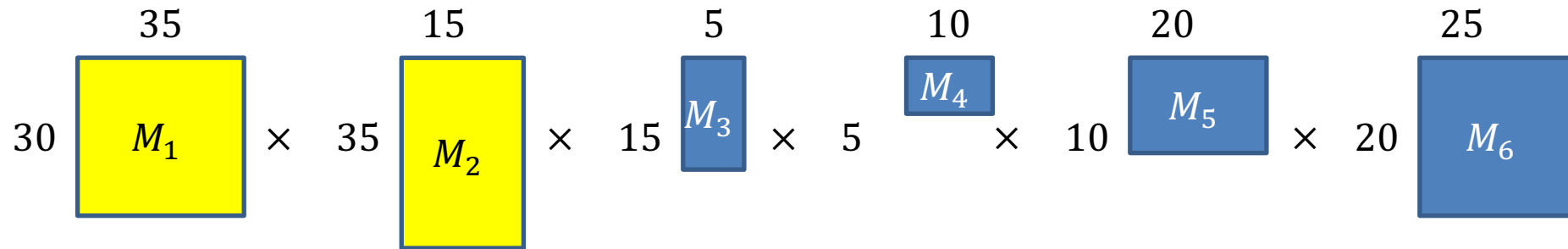


$$Best(i, j) = \min_{k=i}^{j-1} (Best(i, k) + Best(k+1, j) + r_i r_{k+1} c_j)$$

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

$j =$	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



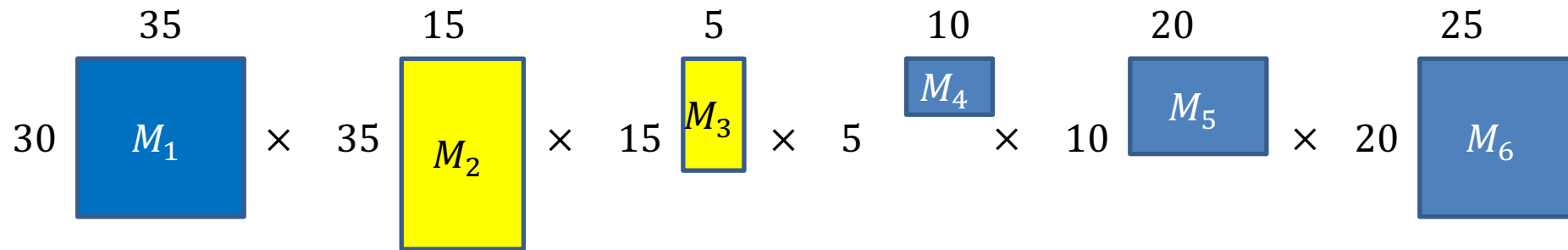
$$Best(i, j) = \min_{k=i}^{j-1} (Best(i, k) + Best(k+1, j) + r_i r_{k+1} c_j)$$

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

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

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

3. Select a good order for solving subproblems



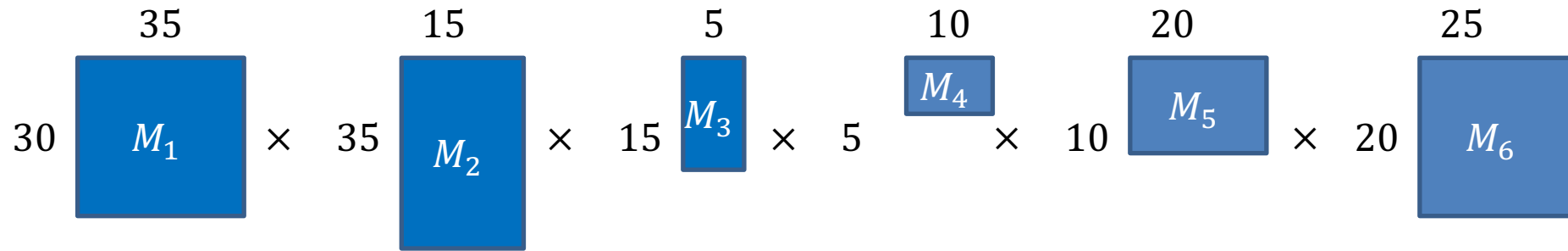
$$Best(i, j) = \min_{k=i}^{j-1} (Best(i, k) + Best(k+1, j) + r_i r_{k+1} c_j)$$

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

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

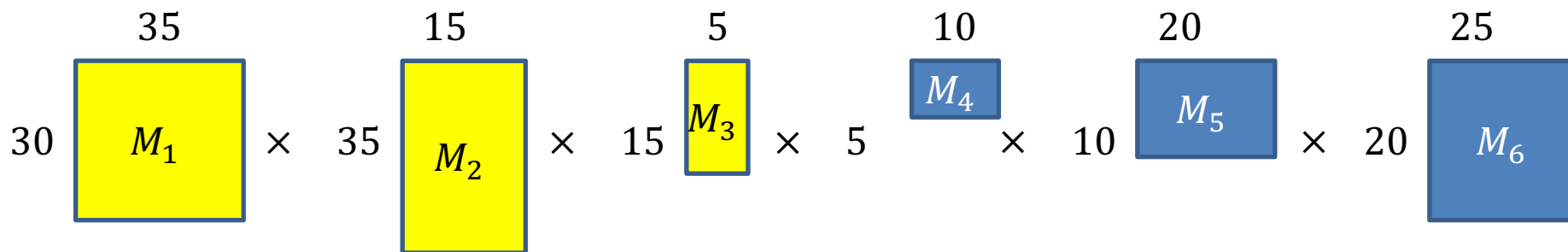


$$Best(i, j) = \min_{k=i}^{j-1} (Best(i, k) + Best(k+1, j) + r_i r_{k+1} c_j)$$

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

$j =$	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



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

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

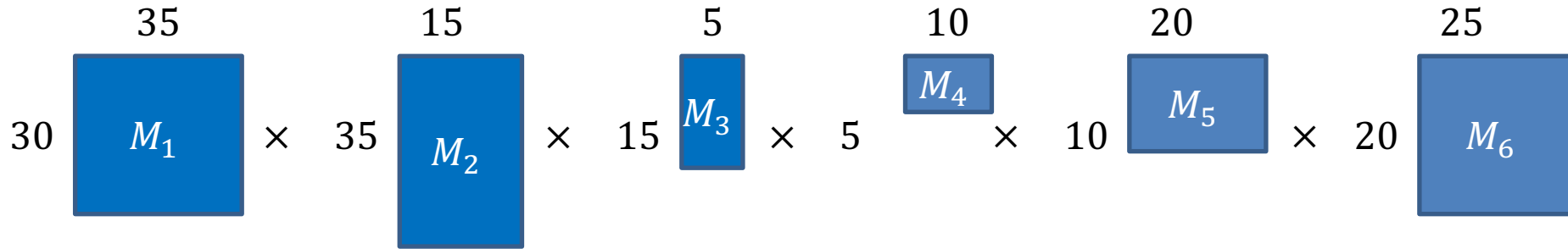
$$r_1 r_2 c_3 = 30 \cdot 35 \cdot 5 = 5250$$

$$r_1 r_3 c_3 = 30 \cdot 15 \cdot 5 = 2250$$

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

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

3. Select a good order for solving subproblems



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

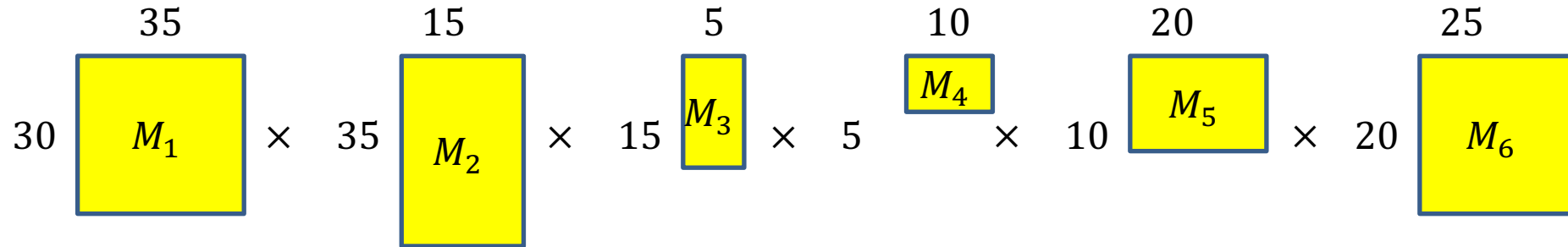
$$Best(i, i) = 0$$

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



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

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

	$j = 1$	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

$Best(1,6) = \min$

- $Best(1,1) + Best(2,6) + r_1 r_2 c_6$
- $Best(1,2) + Best(3,6) + r_1 r_3 c_6$
- $Best(1,3) + Best(4,6) + r_1 r_4 c_6$
- $Best(1,4) + Best(5,6) + r_1 r_5 c_6$
- $Best(1,5) + Best(6,6) + r_1 r_6 c_6$

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. $Best[i, i] = 0$

$\Theta(n)$ options for each cell

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

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

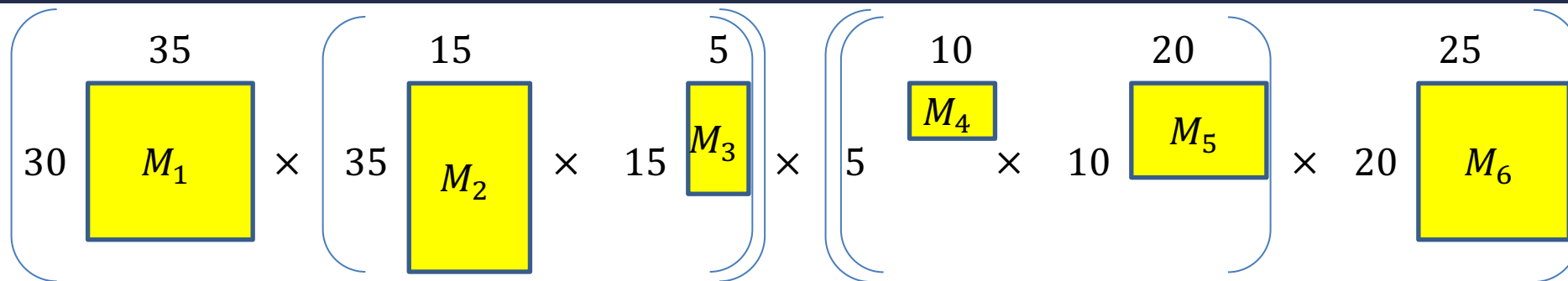
$$Best(i, j) = \min_{k=i}^{j-1} (Best(i, k) + Best(k + 1, j) + r_i r_{k+1} c_j)$$

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

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

$Best(1,6) = \min$	$Best(1,1) + Best(2,6) + r_1 r_2 c_6$ $Best(1,2) + Best(3,6) + r_1 r_3 c_6$ $Best(1,3) + Best(4,6) + r_1 r_4 c_6$ $Best(1,4) + Best(5,6) + r_1 r_5 c_6$ $Best(1,5) + Best(6,6) + r_1 r_6 c_6$	
--------------------	---	--

Matrix Chaining



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

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

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

$$Best(1, 6) = \min$$

- $Best(1, 1) + Best(2, 6) + r_1 r_2 c_6$
- $Best(1, 2) + Best(3, 6) + r_1 r_3 c_6$
- $Best(1, 3) + Best(4, 6) + r_1 r_4 c_6$
- $Best(1, 4) + Best(5, 6) + r_1 r_5 c_6$
- $Best(1, 5) + Best(6, 6) + r_1 r_6 c_6$

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
- 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