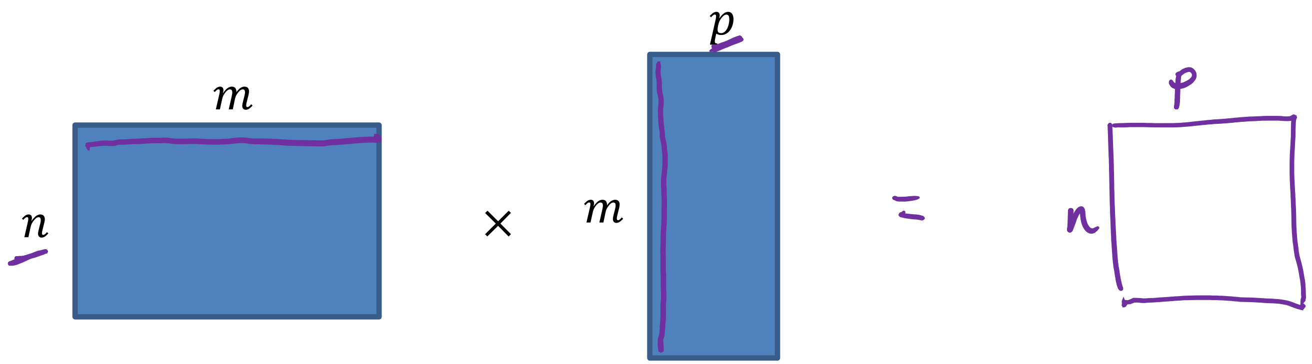
Warm Up

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

(don’t overthink this)

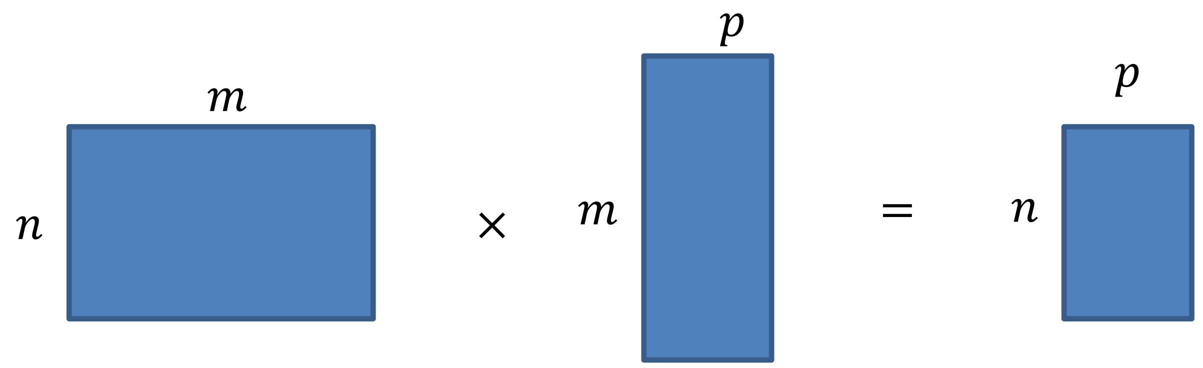


$n \cdot p \cdot (m + m - 1)$

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: $O(m \cdot n \cdot p)$

Homeworks

- HW4 due 11pm Saturday, October 12
  - Sorting, Divide and Conquer, Dynamic Programming
  - Written (use LaTeX!)
  - Submit **BOTH** a pdf and a zip file (2 separate attachments)
- Midterm Exam: Tuesday October 15
  - in class w/take home coding portion
- Regrade Office Hours
  - Thursdays 11am-12pm @ Rice 210
  - Thursdays 4pm-5pm @ Rice 501

Today’s Keywords

- Dynamic Programming
- Log Cutting
- Matrix Chaining
- Seam Carving

CLRS Readings

- Chapter 8

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

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

Given a log of length n

A list (of length n) of prices P ($P[i]$ is the price of a cut of size i)

Find the best way to cut the log

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

Select a list of lengths $\ell_1, \ldots, \ell_k$ such that:

$\sum \ell_i = n$

to maximize $\sum P[\ell_i]$

Brute Force: $O(2^n)$

Greedy 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

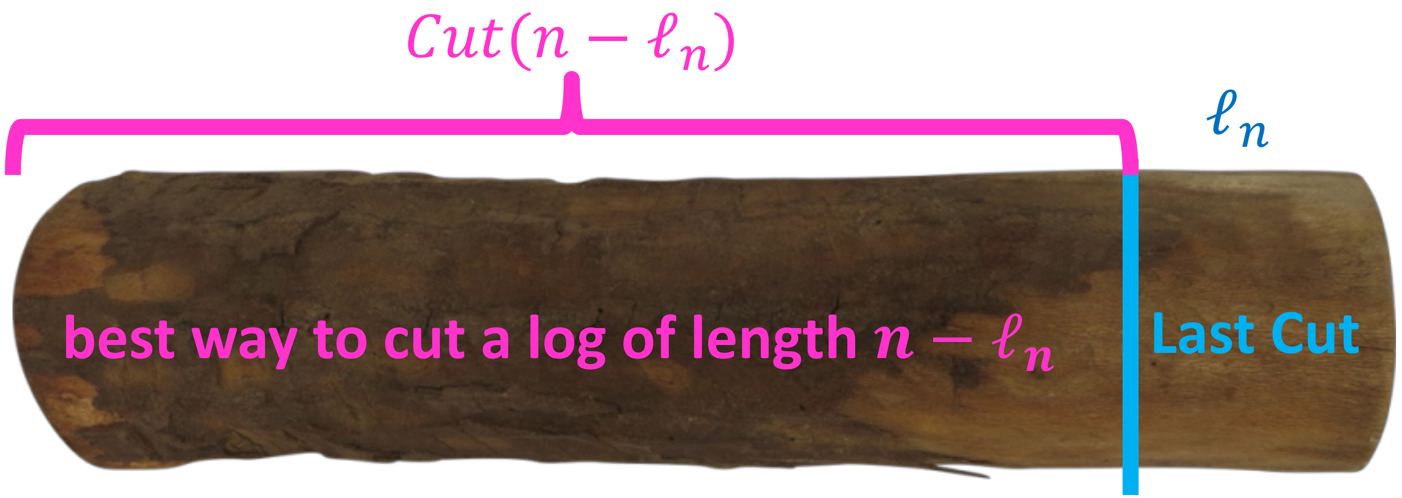
- Requires Optimal Substructure
  - Solution to larger problem contains the solutions to smaller ones
- Idea:
  1. Identify the recursive structure of the problem
     - What is the "last thing" done?
  2. Save the solution to each subproblem in memory
  3. Select a good order for solving subproblems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

0

3. Select a Good Order for Solving Subproblems

Solve Smallest subproblem first

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

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

1

3. Select a Good Order for Solving Subproblems

Solve Smallest subproblem first

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

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

2

3. Select a Good Order for Solving Subproblems

Solve Smallest subproblem first

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

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

3

3. Select a Good Order for Solving Subproblems

Solve Smallest subproblem first

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

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

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

Backtracking Pseudocode

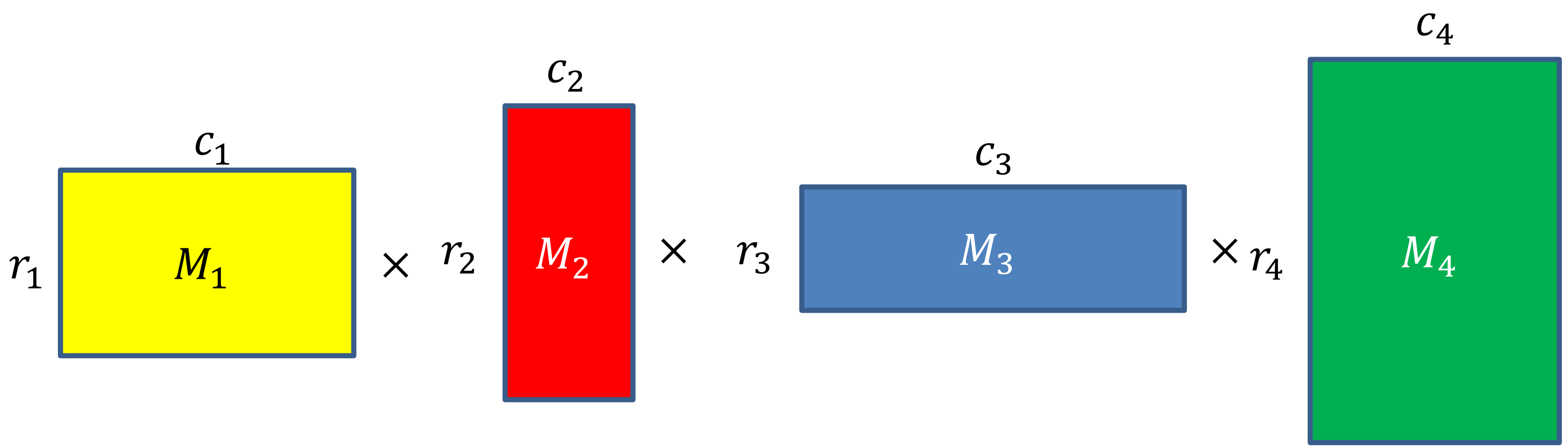
```
i = n
while i > 0:
        print Choices[i]
        i = i – Choices[i]
```

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

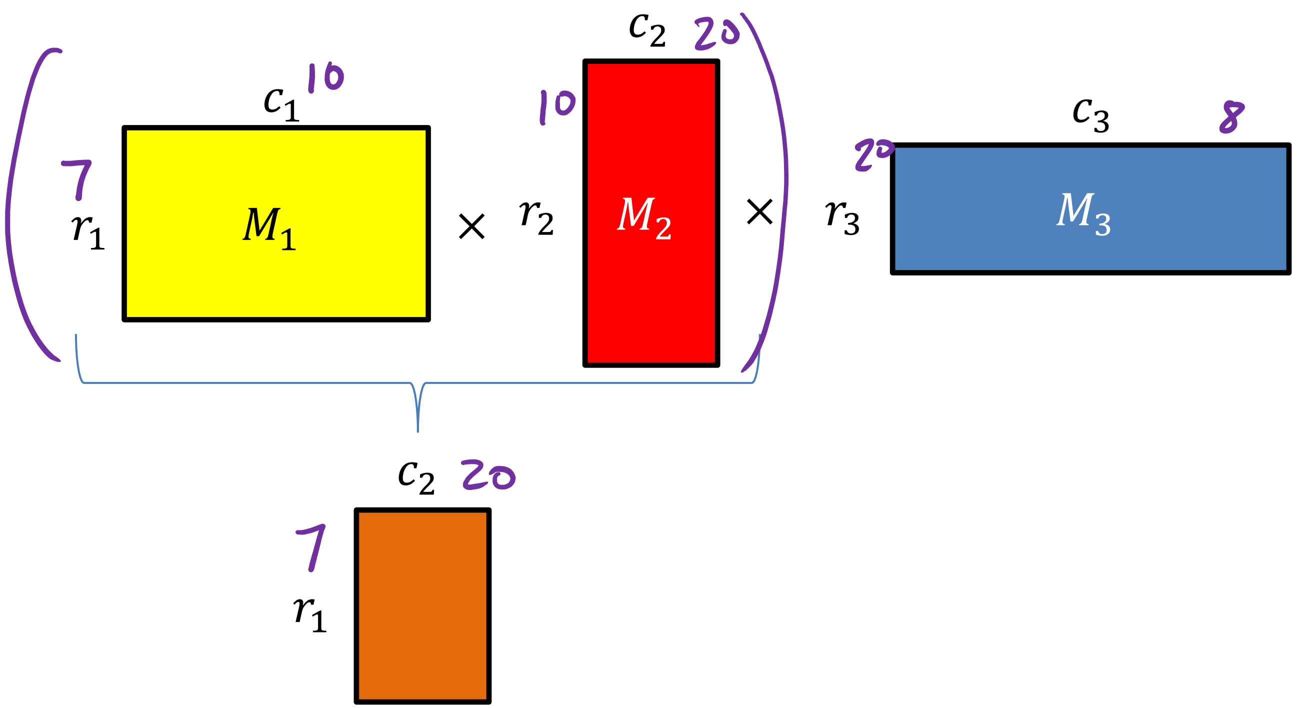
Matrix Chaining

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



Order Matters!

$c_1 = r_2$
$c_2 = r_3$

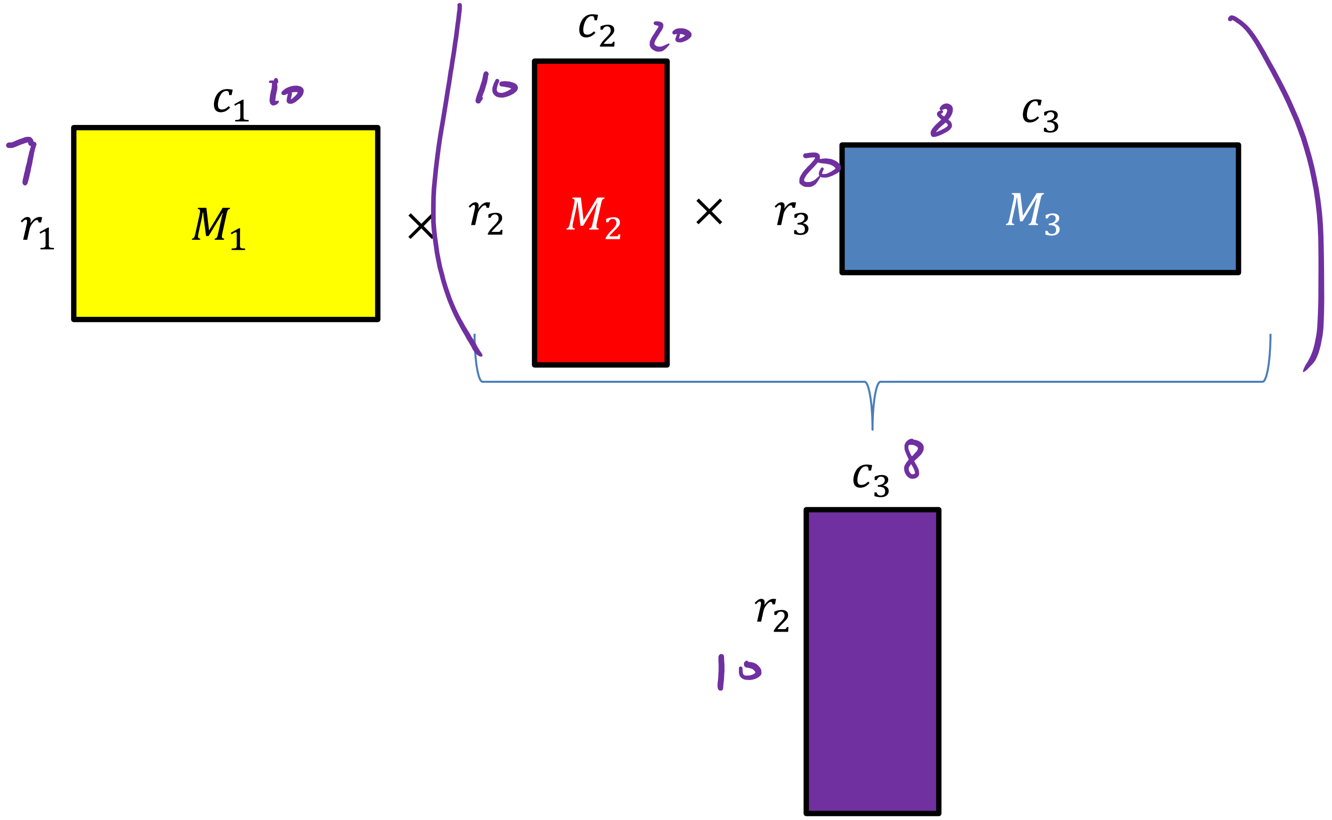


- $(M_1 \times M_2) \times M_3$
  - uses $(c_1 \cdot r_1 \cdot c_2) + c_2 \cdot r_1 \cdot c_3$ operations

(7 × 10 × 20) + 7 · 20 · 8 = 2520
1400 1120

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

$7 \cdot 10 \cdot 8$ $10 \cdot 20 \cdot 8$
560   1600 $= 2160$

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

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

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

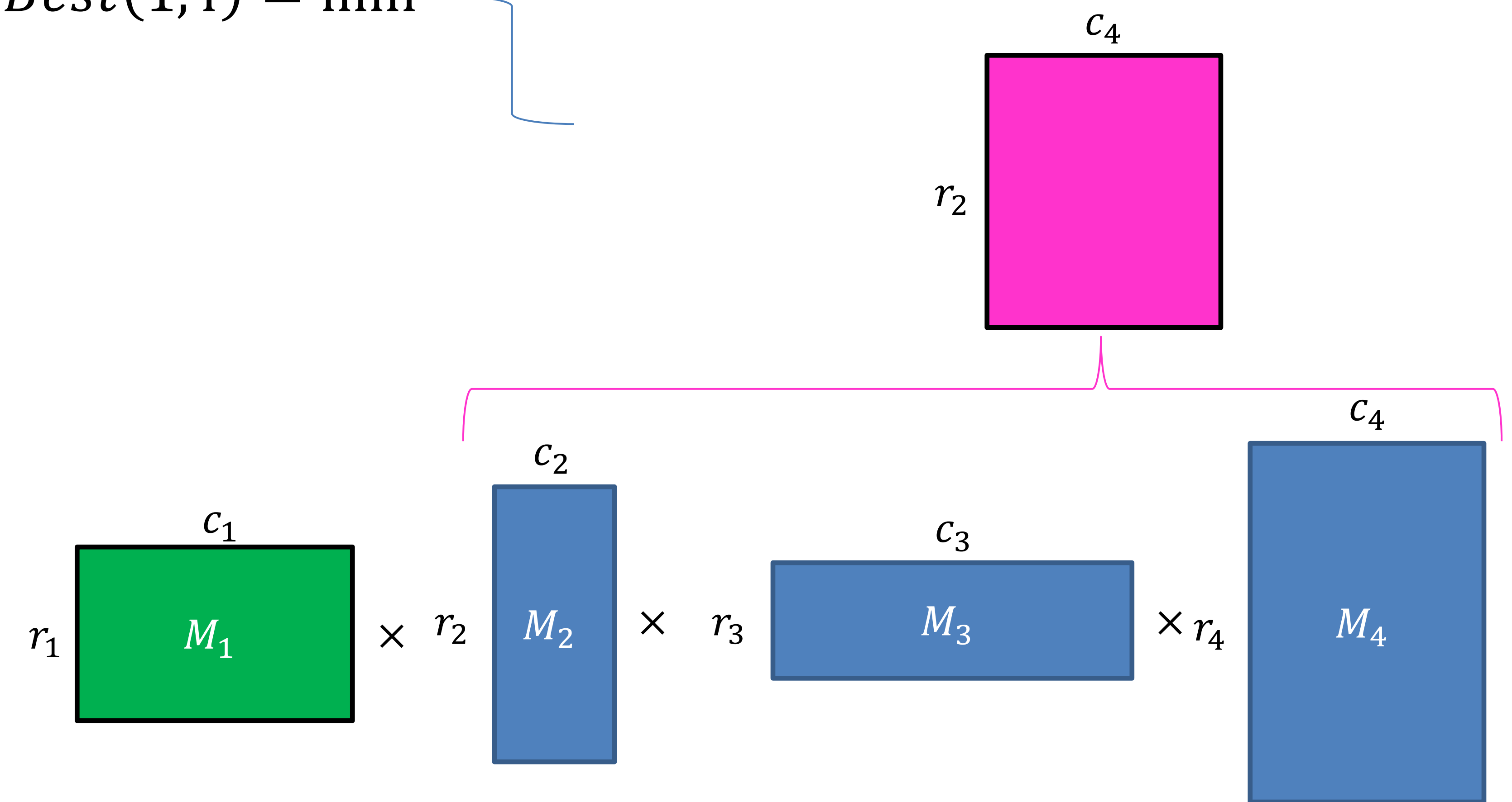
Dynamic Programming

- 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

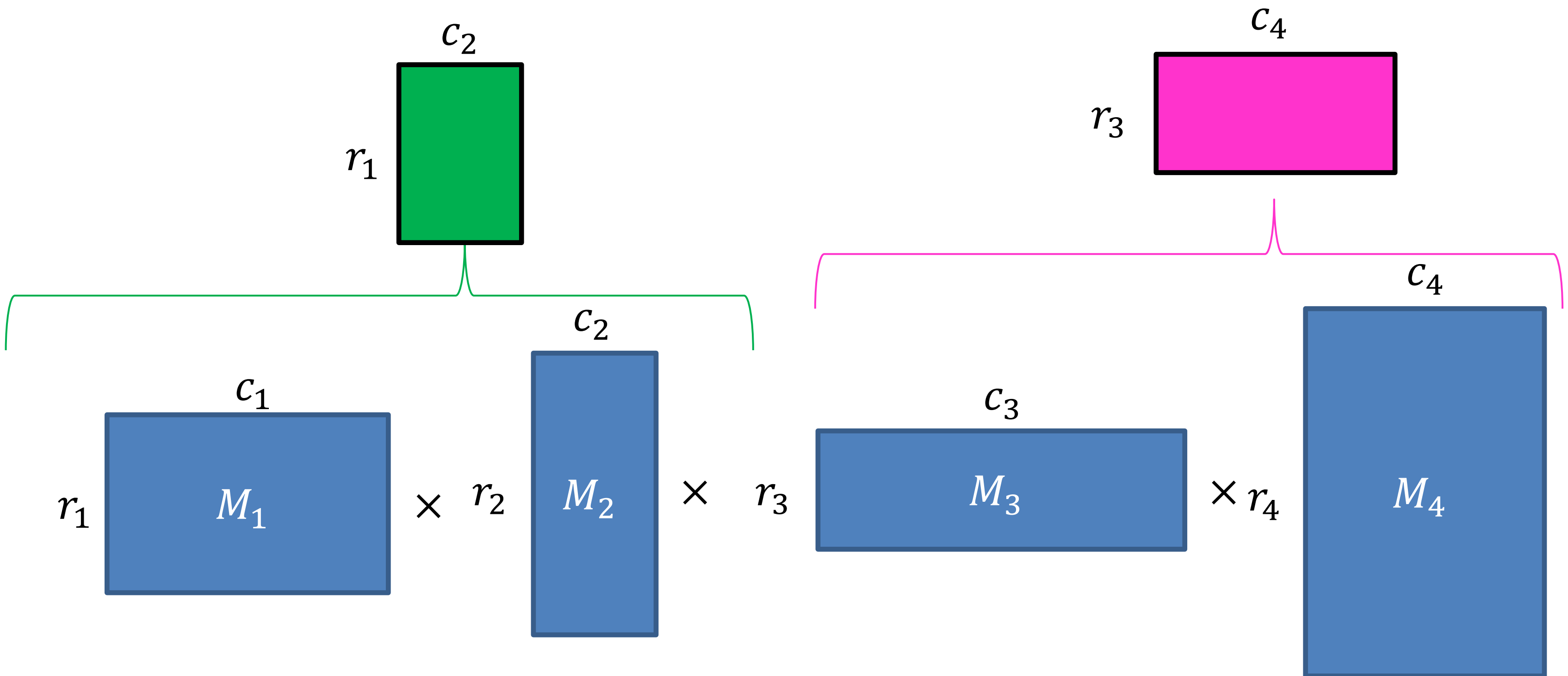
$$Best(1{,}4) = \min \begin{cases} Best(2{,}4) + r_1 r_2 c_4 \\ \\ \end{cases}$$



1. Identify the Recursive Structure of the Problem

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

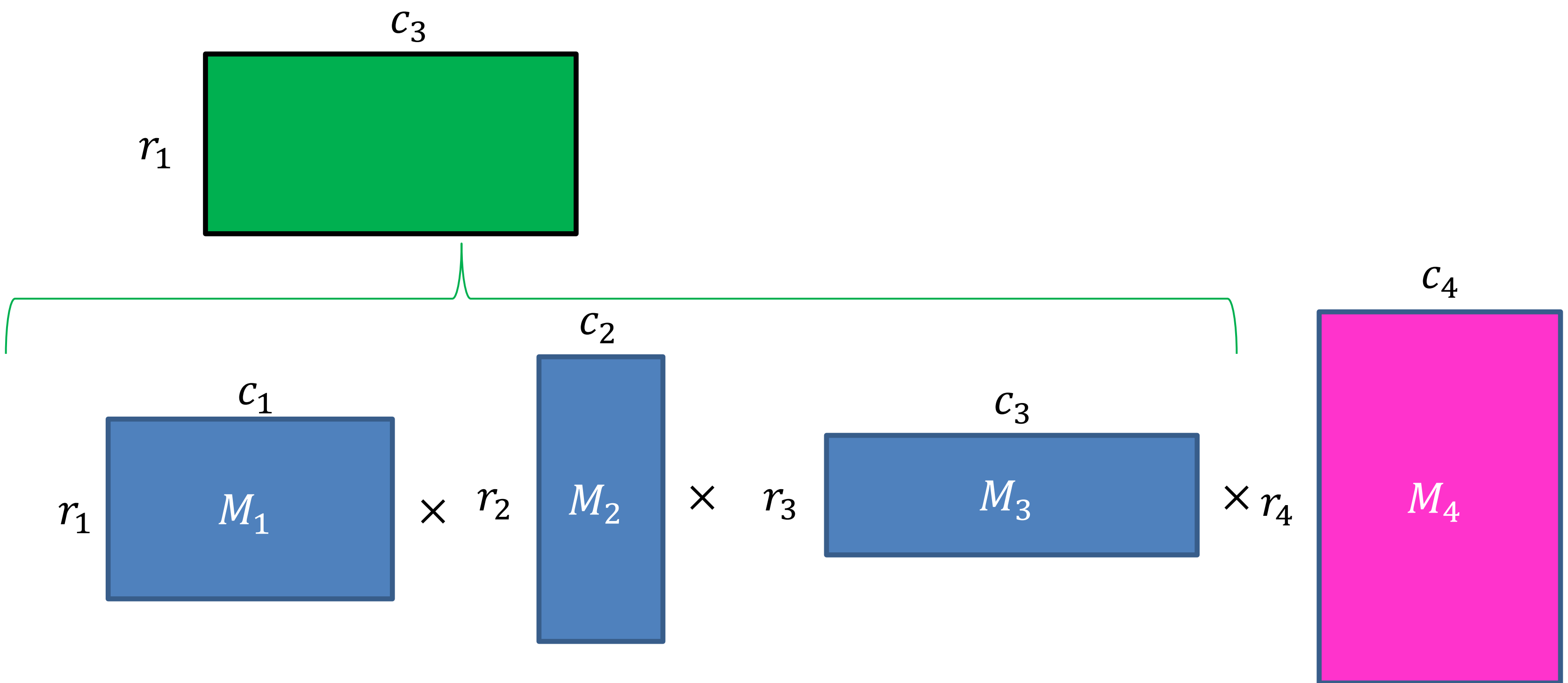
$$Best(1,4) = \min \begin{cases} Best(2,4) + r_1 r_2 c_4 \\ Best(1,2) + Best(3,4) + r_1 r_3 c_4 \end{cases}$$



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

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

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

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

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

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]

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

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

3. Select a good order for solving subproblems

- In general:

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

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

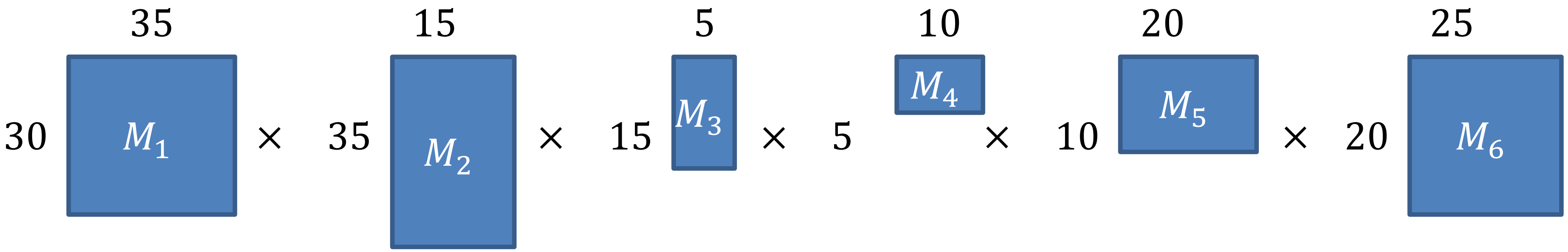
$$Best(i,i) = 0$$

Read from M[n] if present

Save to M[n]

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

3. Select a good order for solving subproblems

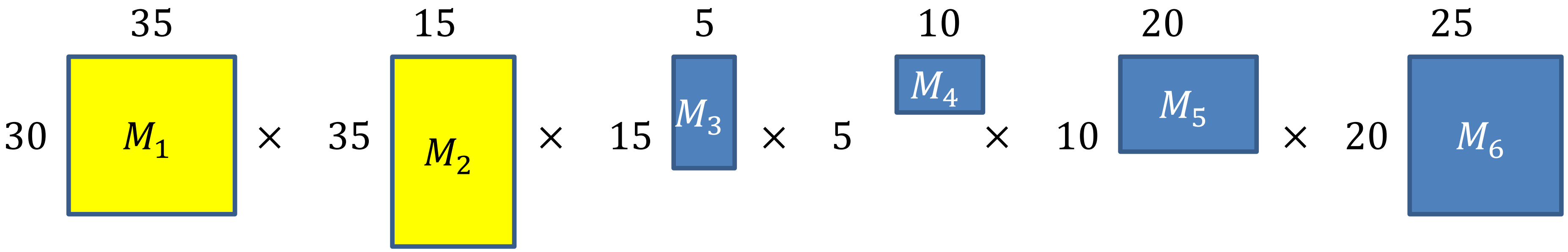


$$Best(i,j) = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$$

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

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

3. Select a good order for solving subproblems



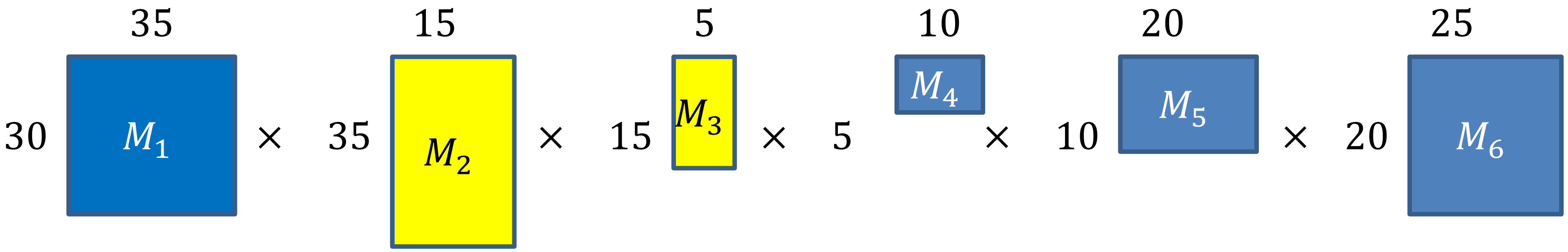
$$Best(i,j) = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$$

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

$j =$ / $i =$	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



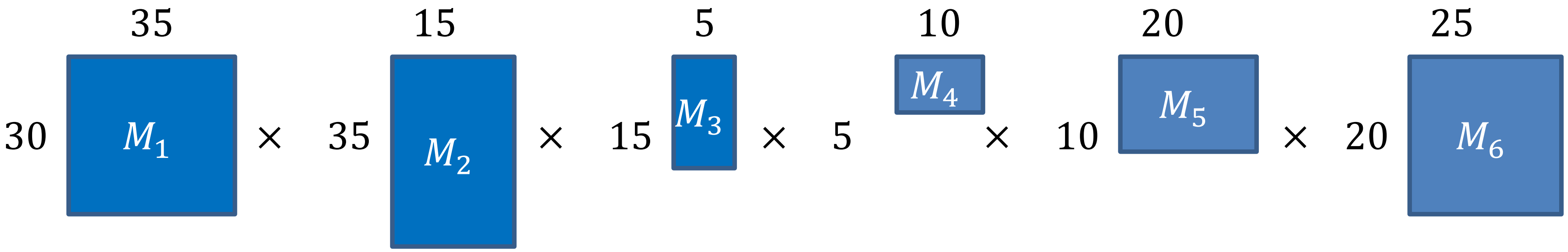
$$Best(i,j) = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$$

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

$j =$ 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

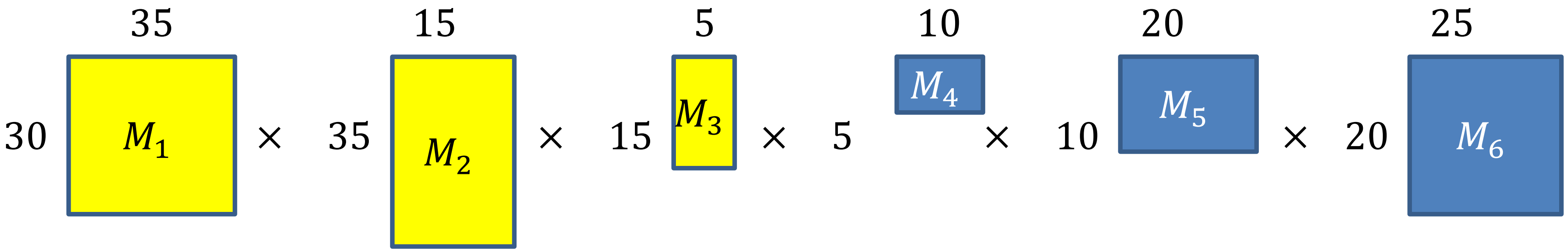


$$Best(i,j) = \min_{k=i}^{j-1}\left(Best(i,k) + Best(k+1,j) + r_i r_{k+1} c_j\right)$$

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

$j =$	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



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

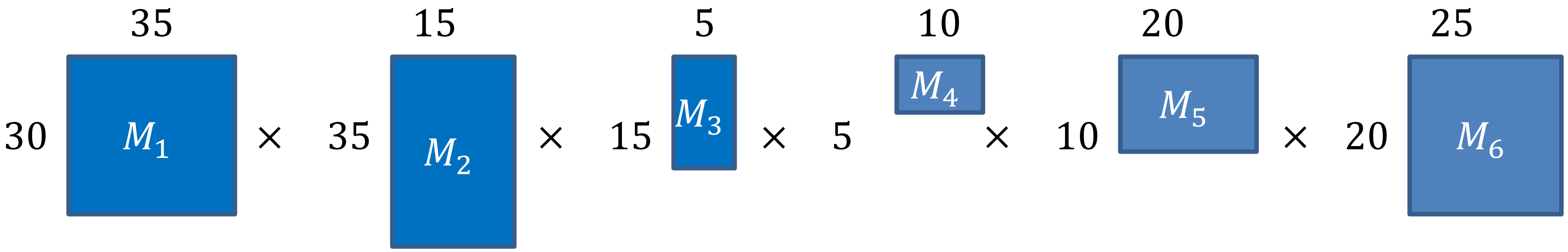
$$Best(i,i) = 0$$

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

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

$j =$	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

3. Select a good order for solving subproblems



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

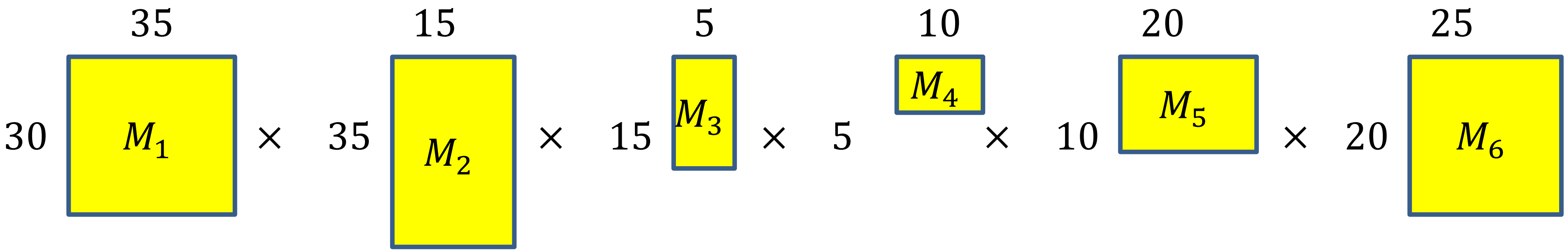
$$Best(i,i) = 0$$

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



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

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

$j =$ 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

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

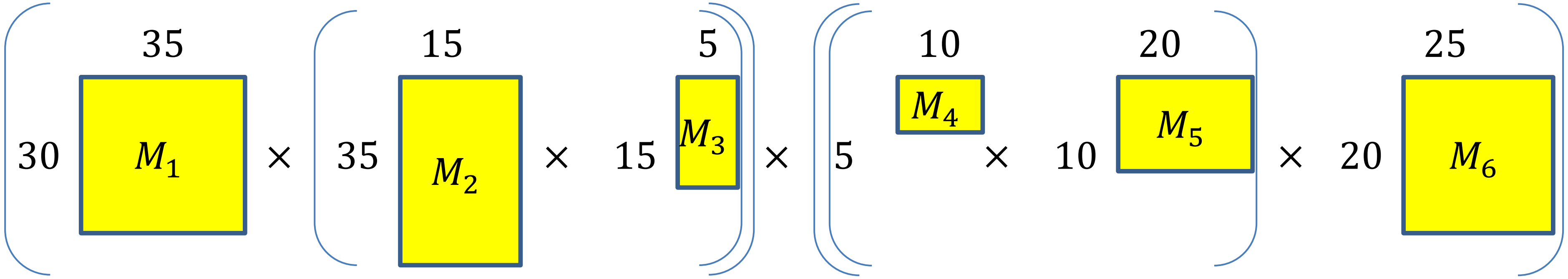
$$Best(i,i) = 0$$

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

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

46

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

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

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