Dynamic Programming!

- Example of Dynamic programming:
  - Fibonacci numbers.
- What is dynamic programming, exactly?
- Applications:
  - Matrix-chain Multiplication
  - Longest Common Subsequence
  - 0/1 knapsack Algorithm
  - Optimal Binary Search Tree

Fibonacci Numbers

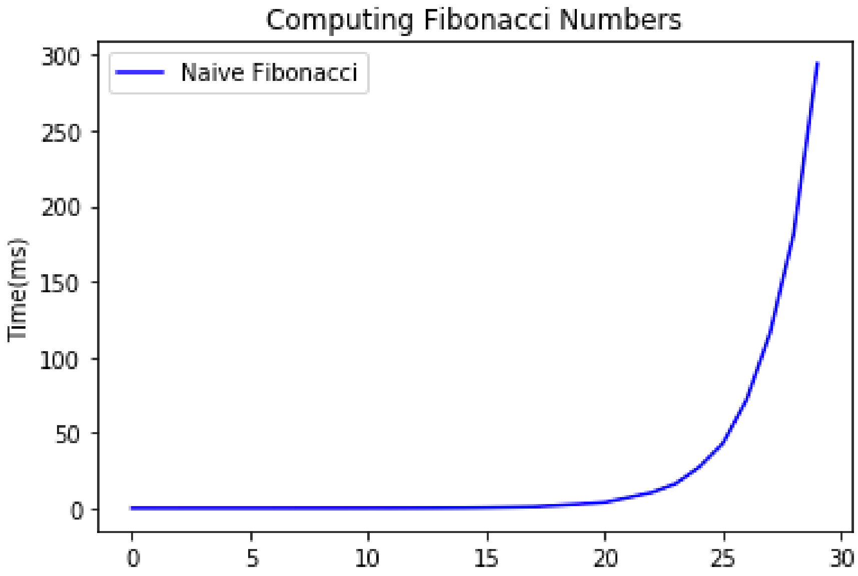
- Definition:
  - $F(n) = F(n\text{-}1) + F(n\text{-}2)$, with $F(1) = F(2) = 1$.
  - The first several are:
    - 1
    - 1
    - 2
    - 3
    - 5
    - 8
    - 13, 21, 34, 55, 89, 144,…
- Question:
  - Given n, what is $F(n)$?

Candidate algorithm

```
• def Fibonacci(n):
    • if n == 0, return 0
    • if n == 1, return 1
    • return Fibonacci(n-1) + Fibonacci(n-2)
```

Running time?

- T(n) = T(n-1) + T(n-2) + O(1)
- T(n) $\geq$ T(n-1) + T(n-2) for n $\geq$ 2
- So T(n) grows *at least* as fast as the Fibonacci numbers themselves…
- This is **EXPONENTIALLY QUICKLY!**



IPython notebook

Why do the Fibonacci numbers grow exponentially quickly?

- $T(n) = T(n-1) + T(n-2)$
- $\geq 2T(n-2)$

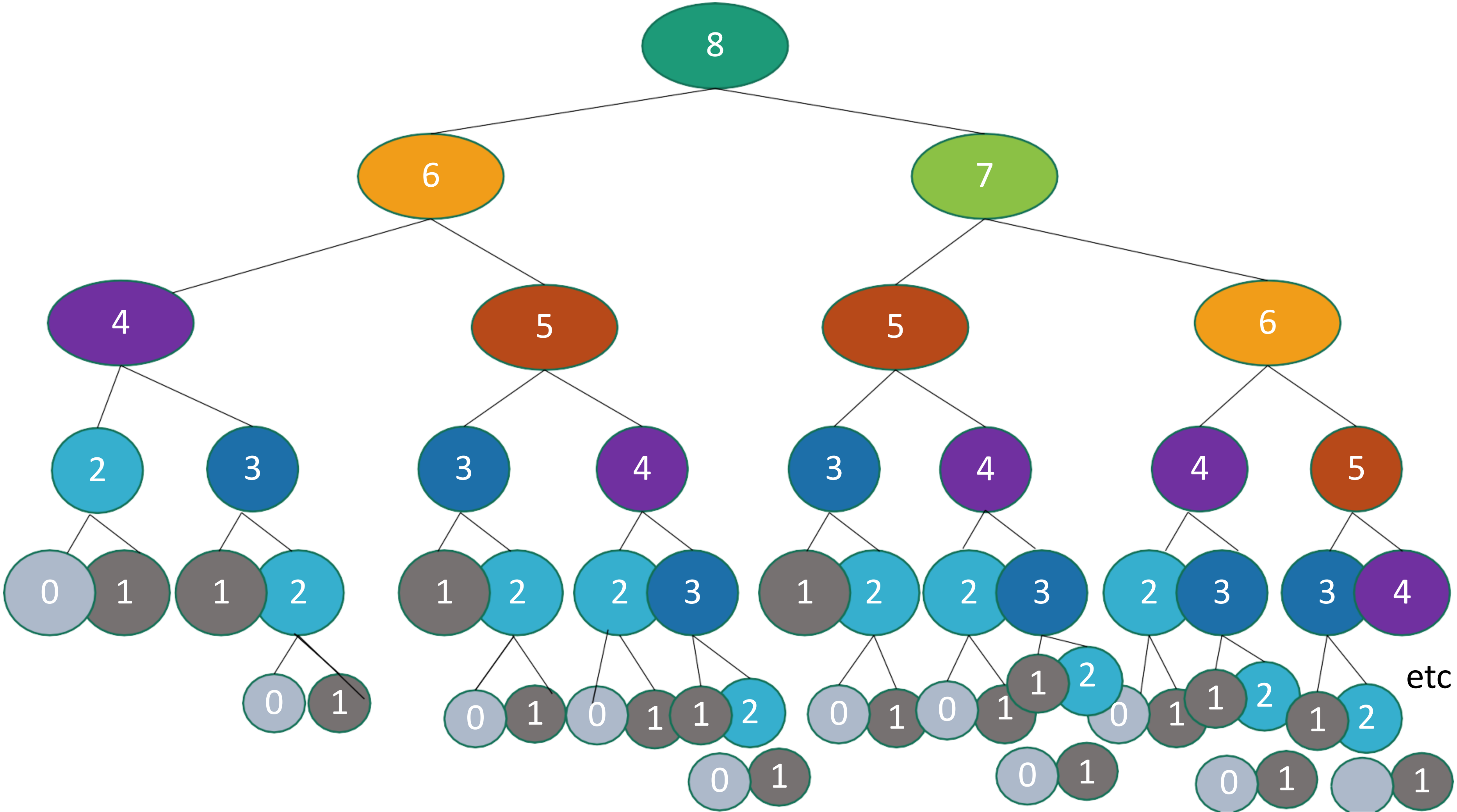
Trying solving this with the Back Substitution method

- $T(n) \geq 2T(n-2)$
- $\geq 4T(n-4)$
- $\geq 8T(n-6)$
- … $\geq 2^{\mathrm{k}} T(n-2\mathrm{k})$ for any $\mathrm{k} < n/2$
- … $\geq 2^{n/2} T(1)$ by plugging in $\mathrm{k} = \frac{n-1}{2}$
- So $\boldsymbol{T(n) \geq 2^{n/2}}$, which is REALLY BIG!!!

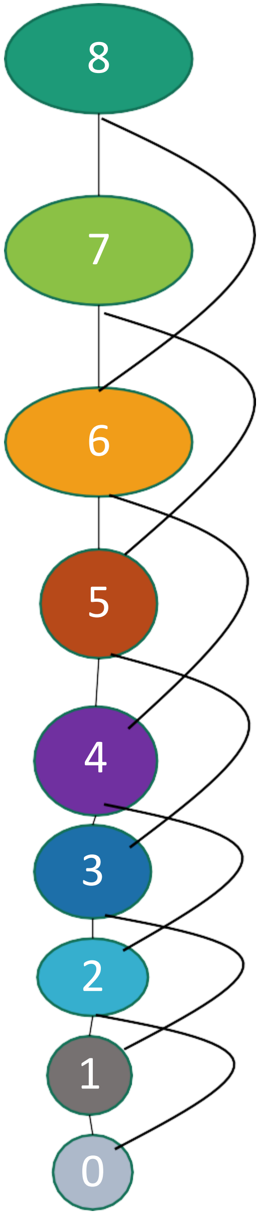
What's going on?

Consider Fib(8)

That's a lot of repeated computation!

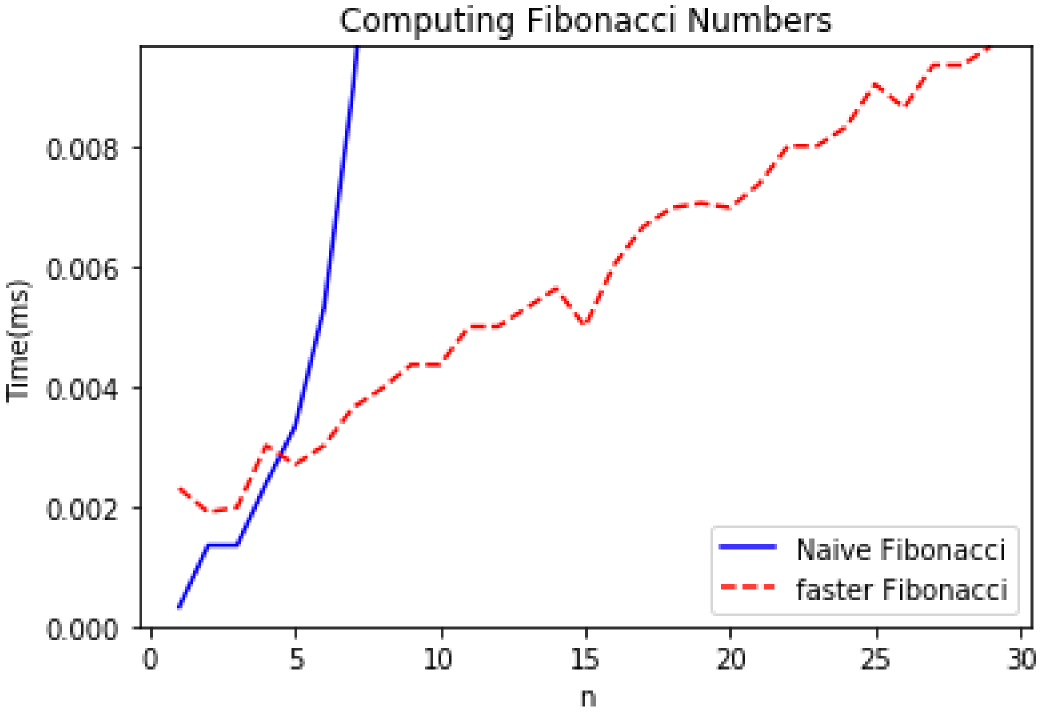


Maybe this would be better:



```
def fasterFibonacci(n):
    • F = [0, 1, None, None, …, None ]
        • \\ F has length n + 1
    • for i = 2, …, n:
        • F[i] = F[i-1] + F[i-2]
    • return F[n]
```

Much better running time!



This was an example of…

Dynamic Programming!

What is *dynamic programming*?

- It is an algorithm design paradigm
  - like divide-and-conquer is an algorithm design paradigm.
- Usually, it is for solving **optimization problems**
  - eg, ***shortest*** path (Bellman Ford's Algorithm)
  - (Fibonacci numbers aren't an optimization problem, but they are a good example of DP anyway…)

Elements of dynamic programming

1. Optimal sub-structure:

- Big problems break up into sub-problems.
  - Fibonacci: F(i) for i ≤ n

- The optimal solution to a problem can be expressed in terms of optimal solutions to smaller sub-problems.
  - Fibonacci:

$$F(i+1) = F(i) + F(i-1)$$

Elements of dynamic programming

2. Overlapping sub-problems:

- The sub-problems overlap.
  - Fibonacci:
    - Both F[i+1] and F[i+2] directly use F[i].
    - And lots of different F[i+x] indirectly use F[i].
  - This means that we can save time by solving a sub-problem just once and storing the answer.

Elements of dynamic programming

- Optimal substructure.
  - Optimal solutions to sub-problems can be used to find the optimal solution to the original problem.
- Overlapping subproblems.
  - The subproblems show up again and again

- Using these properties, we can design a ***dynamic programming*** algorithm:
  - Keep a table of solutions to the smaller problems.
  - Use the solutions in the table to solve bigger problems.
  - Ultimately, we can use the information we collected to find the solution to the whole thing.

- Construct the optimal solution

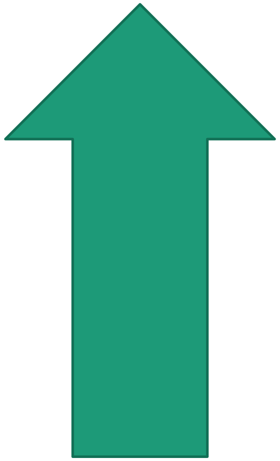
Two ways to think about and/or implement DP algorithms

- Top down

- Bottom up

Bottom up approach

what we just saw.



- For Fibonacci:
- Solve the small problems first
  - fill in F[0],F[1]
- Then bigger problems
  - fill in F[2]
- …
- Then bigger problems
  - fill in F[n-1]
- Then finally solve the real problem.
  - fill in F[n]

Top down approach

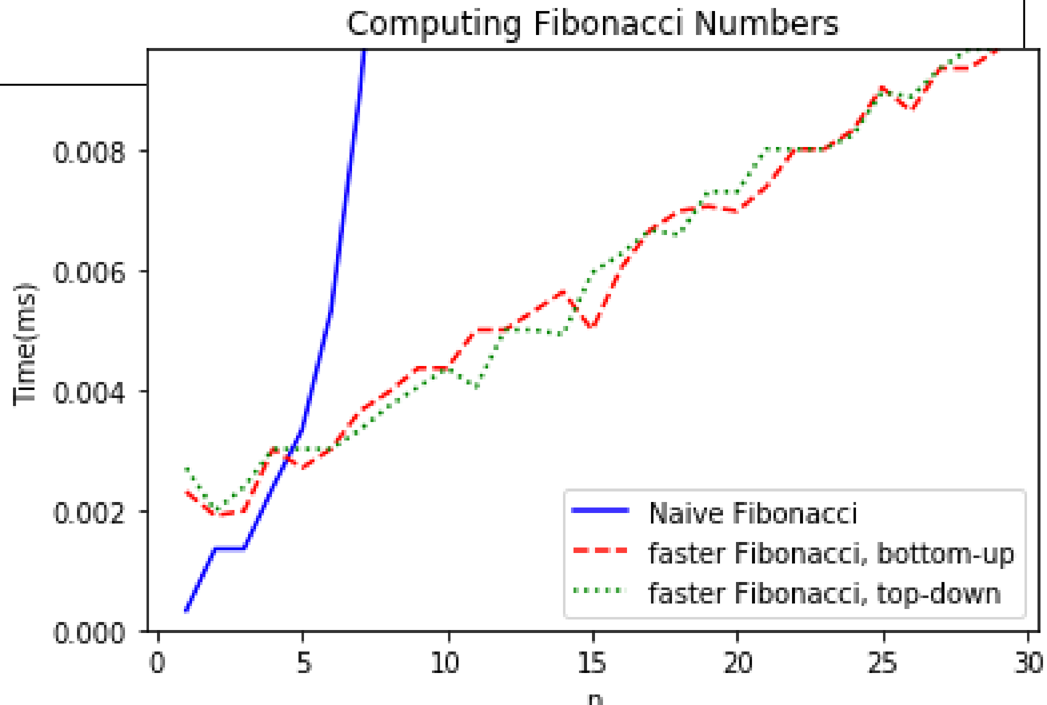
- Think of it like a recursive algorithm.
- To solve the big problem:
    - Recurse to solve smaller problems
        - Those recurse to solve smaller problems
            - etc..

- The difference from divide and conquer:
    - Keep track of what small problems you've already solved to prevent re-solving the same problem twice.
    - Aka, "**memo-ization**"

Example of top-down Fibonacci

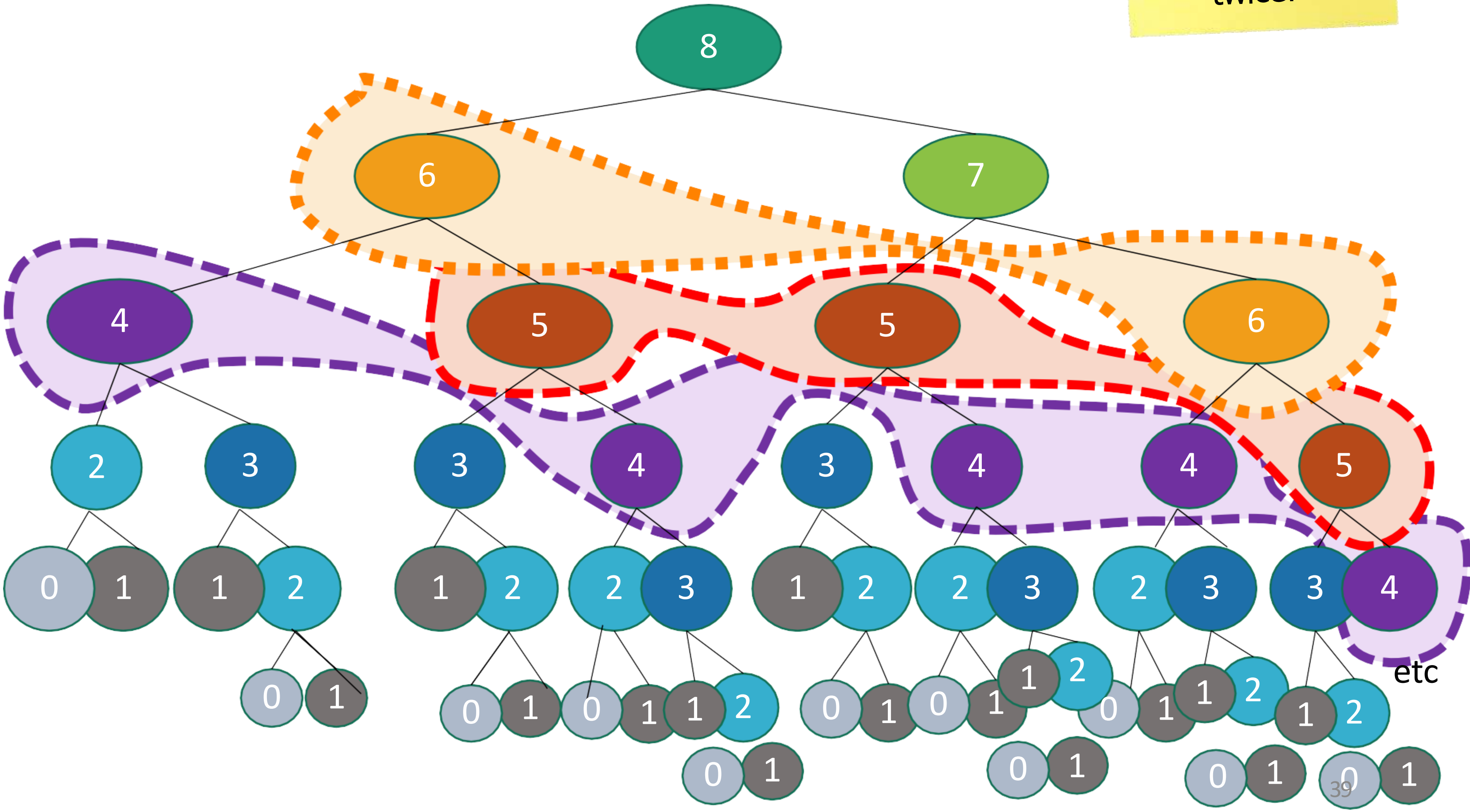
```
define a global list F = [0,1,None, None, …, None]
def Fibonacci(n):
    if F[n] != None:
        return F[n]
    else:
        F[n] = Fibonacci(n-1) + Fibonacci(n-2)
    return F[n]
```

Memo-ization:
Keeps track (in F)
of the stuff you've
already done.



Memo-ization visualization

Collapse repeated nodes and don't do the same work twice!



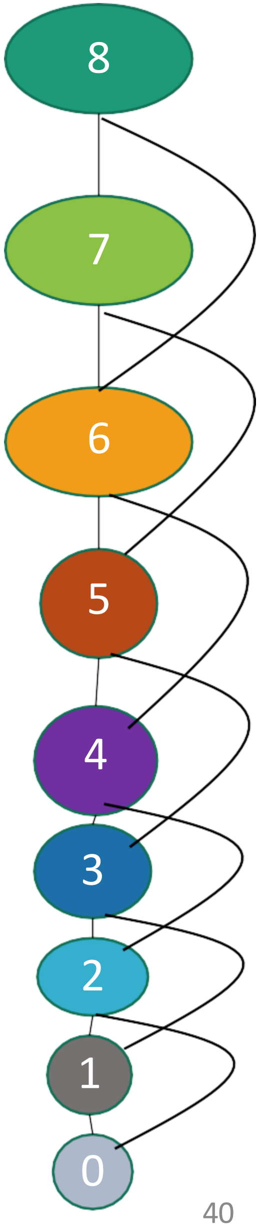
Memo-ization Visualization

ctd

Collapse repeated nodes and don't do the same work twice!

But otherwise treat it like the same old recursive algorithm.

```
• define a global list F = [0,1,None, None, …, None]
• def Fibonacci(n):
    • if F[n] != None:
        • return F[n]
    • else:
        • F[n] = Fibonacci(n-1) + Fibonacci(n-2)
    • return F[n]
```



What have we learned?

- ***Dynamic programming:***
  - Paradigm in algorithm design.
  - Uses **optimal substructure**
  - Uses **overlapping subproblems**
  - Can be implemented **bottom-up** or **top-down.**
  - It's a fancy name for a pretty common-sense idea:

Don't duplicate work if you don't have to!

Steps for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.

- **Step 2:** Find a recursive formulation for the thing you want.

- **Step 3:** Use dynamic programming to find the thing you want.
  - Fill in a table, starting with the smallest sub-problems and building up.

- **Step 4:** construct an optimal solution (will discuss with applications…)

Applications of
Dynamic Programming

Matrix-chain Multiplication

Let’s discuss Matrix Multiplication First

Matrix Multiplication

Matrix: A $n \times m$ matrix $A = [a[i,j]]$ is a two-dimensional array

$$A = \begin{bmatrix} a[1,1] & a[1,2] & \cdots & a[1,m-1] & a[1,m] \\ a[2,1] & a[2,2] & \cdots & a[2,m-1] & a[2,m] \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ a[n,1] & \cdots & \cdots & a[n,m-1] & a[n,m] \end{bmatrix}$$

Which has n rows and m columns

Example: The following is a 2×3 matrix:

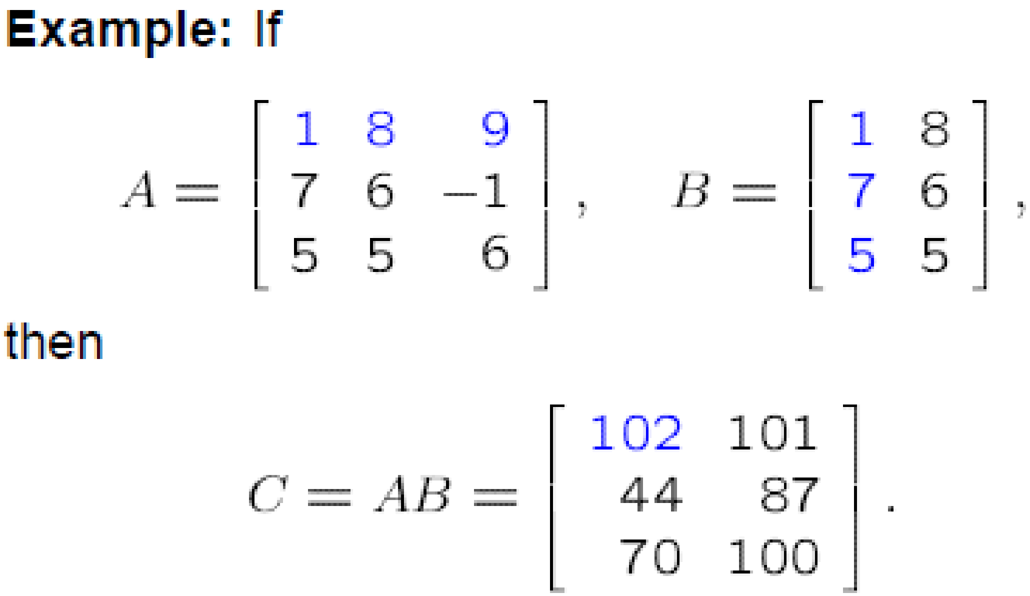
$$\begin{bmatrix} 5 & 2 & 3 \\ -3 & 1 & 4 \end{bmatrix}$$

Matrix Multiplication

The product $C = AB$ of a $p \times q$ matrix A and a $q \times r$ matrix $\mathbf{B}$ is a $p \times r$ matrix given by

$$c[i,j] = \sum_{k=1}^{q} a[i,k]\, b[k,j]$$

For $1 \le i \le p$ and $1 \le j \le r$



Example: If

$$A = \begin{bmatrix} 1 & 8 & 9 \\ 7 & 6 & -1 \\ 5 & 5 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 8 \\ 7 & 6 \\ 5 & 5 \end{bmatrix},$$

then

$$C = AB = \begin{bmatrix} 102 & 101 \\ 44 & 87 \\ 70 & 100 \end{bmatrix}.$$

Matrix Multiplication

- If AB is defined, BA may not be defined

- Quite possible that $AB \neq BA$

- Multiplication is recursively defined by

$$A_1 A_2 A_3 \ldots A_{n\,\_\,1} A_n = A_1(A_2 (A_3 \ldots (An\,\_\,{}_1 A_n)))$$

- Matrix multiplication is **associative**, i.e.,

$$A_1 A_2 A_3 =(A_1 A_2)A_3 = A_1(A_2 A_3)$$

so parenthesis does not change the result.

Matrix Multiplication

Given a $p \times q$ matrix A and a $q \times r$ matrix B, the direct way of multiplying $C = AB$ is to compute a $p \times r$ matrix by

$$c[i,j] = \sum_{k=1}^{q} a[i,k]\, b[k,j]$$

For $1 \leq i \leq p$ and $1 \leq j \leq r$.

Complexity of Direct Matrix Multiplication

Note that C has pr entries and each entry takes $O(q)$ time to compute so the total procedure takes $\boldsymbol{O(pqr) time}$

Matrix Multiplication of ABC

Given a $p \times q$ matrix A and a $q \times r$ matrix B and a $r \times s$ matrix C, then ABC can be computed in two ways $(AB)C$ and $\boldsymbol{A(BC)}$:

The number of multiplications needed are:

$$mult[\,(AB)C\,] = pqr + prs$$
$$mult[\,A(BC)\,] = qrs + pqs$$

When p = 5, q = 4, $r = 6$ and $s = 2$, then

$$mult[\,(AB)C\,] = 180$$
$$mult[\,A(BC)\,] = 88$$

A big difference!

Implication: The multiplication "sequence" (parenthesis) is important

Matrix-chain Multiplication

Given a sequence $A_1, A_2, \ldots, An$ of n matrices to be multiplied with the dimensions of $p_0, p_1, \ldots, p_n$ respectively

A_i, has a dimension of $p_{i\ \_\ 1} \times\ p_i$,

- determine the multiplication sequence that minimizes the number of scalar multiplications in computing $A_1, A_2, \ldots, An$
- determine how to parenthesize the multiplications to compute the product of $A_1, A_2, \ldots, An$ with minimum multiplications

$$
\begin{aligned}
A_1 A_2 A_3\ A_4 &= (A_1 A_2)(A_3, A_4) \\
&= A_1 (A_2(A_3 A_4)) \ = A_1((A_2 A_3)A_4) \\
&= ((A_1 A_2)A_3)A_4 \ = (A_1\ (A_2 A_3))A_4
\end{aligned}
$$

Exhaustive search: Exponential

DP a better approach…

Steps for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the matrix-chain multiplication
- **Step 3:** Use dynamic programming to find the minimum product in matrix-chain multiplication.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual parenthesis multiplication.

Step 1: Optimal substructure

Decompose the problem into subproblems:

for each pair $1 \leq i \leq j \leq n$,

determine the multiplication sequence for

$$A_{i\,\ldots\,j} = A_i, A_{i\,+\,1}, \ldots, A_j$$

that minimizes the number of multiplications.

$A_{i\,\ldots\,j}$ is a $p_{i\,-\,1} \times p_j$ matrix

Original problem: determine a sequence of multiplication for $A_{1\,\ldots\,n}$

Step 1: Optimal substructure

Split the original structure into substructures

Suppose $A_i, A_{i+1}, \ldots, A_j$ is split between two substructures around k

$$A_i, A_{i+1}, \ldots, A_j$$

Split into 2 substructures

$$\boldsymbol{A_i, A_{i+1}, \ldots, A_k} \qquad \boldsymbol{A_{k+1}, \ldots, A_j}$$

- Find the optimal solution for both substructures
- And then combine them to get the optimal solution of the original structure

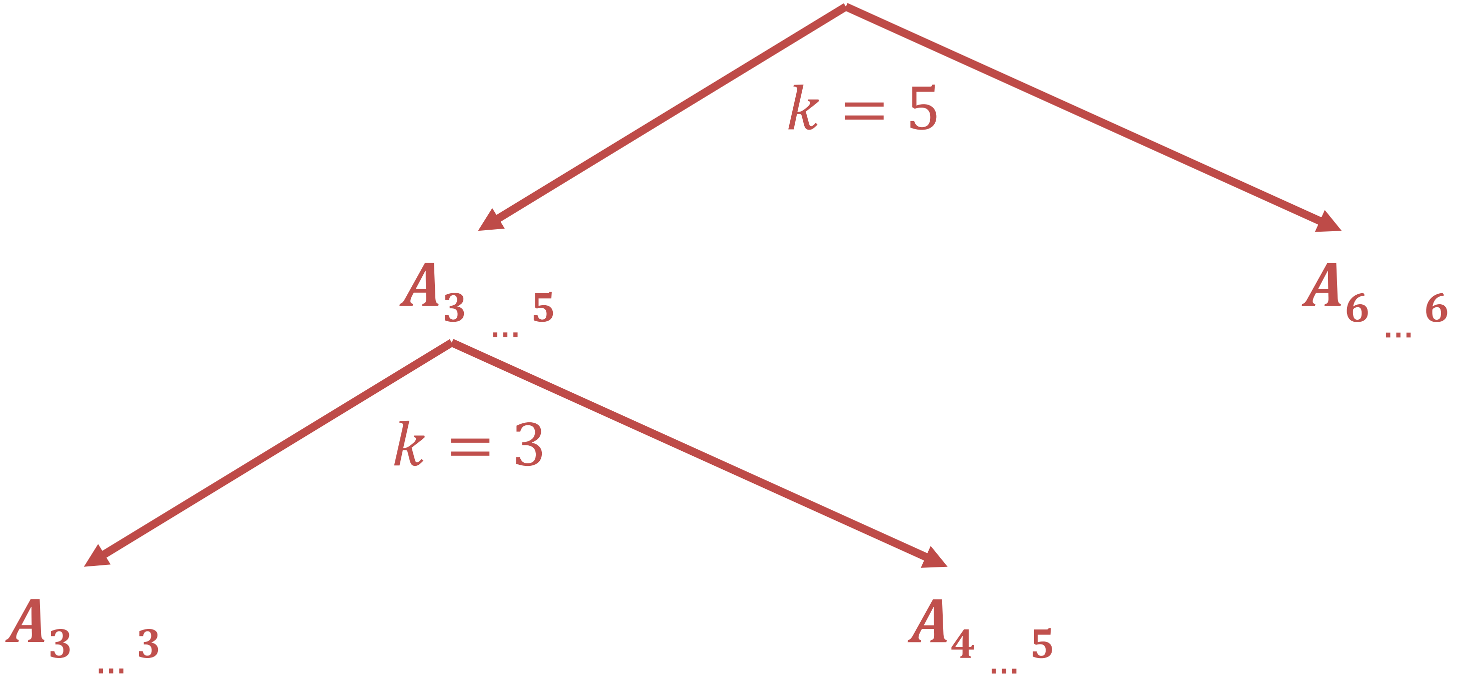
NOTE: Need to ensure the correct place (*i.e.*, k) to split the product

Step 1: Optimal substructure

Split the original structure into substructures

Suppose A_i, A_{i+1}, …, A_j is split between two substructures around k

$$A_{3\ldots6} = (A_3(A_4A_5))(A_6)$$



$k = 5$

$\boldsymbol{A_{3\ldots5}}$

$\boldsymbol{A_{6\ldots6}}$

$k = 3$

$\boldsymbol{A_{3\ldots3}}$

$\boldsymbol{A_{4\ldots5}}$

Step 1: Optimal substructure

Split the original structure into substructures

Suppose A_i, A_{i+1}, …, A_j is split between two substructures around k

- How do we decide where to split the chain (what is k)?

(Search all possible values of k)

- How do we parenthesize the substructures $\boldsymbol{A_{i \ldots k}}$ and $\boldsymbol{A_{k+1 \ldots j}}$?

(Problem has optimal substructure property that $\boldsymbol{A_{i \ldots k}}$ and $\boldsymbol{A_{k+1 \ldots j}}$ must be optimal so we can apply the same procedure recursively (STEP 2))

Steps for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the matrix-chain multiplication
- **Step 3:** Use dynamic programming to find the minimum product in matrix-chain multiplication.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual parenthesis multiplication.

Step 2: Recursive Formulation

Find the cost of the optimal solution of the original problem recursively in terms of the optimal solution to the subproblems

Objective: Minimum cost parenthesis

$A_i, A_{i+1}, \ldots, A_j$ where $1 \leq i \leq j \leq n$

Let $m[i,j]$ be the minimum number of scalar multiplication needed to compute matrix $A_{i \ldots j}$

Step 2: Recursive Formulation

Objective: Minimum cost parenthesis

$A_i, A_{i+1}, \ldots, A_j$ where $1 \leq i \leq j \leq n$

CASE 1 (trivial case):

$i == j$ means only a single matrix

$$A_{i \ldots i} = Ai$$

Thus, $\boldsymbol{m[i,i]} = \mathbf{0}$, for $\boldsymbol{i = 1, 2, \ldots, n}$

CASE 2: when $i < j$

Split $A_{i \ldots j}$ into $\boldsymbol{A_{i \ldots k}}$ and $\boldsymbol{A_{k+1 \ldots j}}$ substructures

Thus

$$\boldsymbol{m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j}$$

for $\boldsymbol{k = i, i+1, i+2, \ldots, j-1}$

Step 2: Recursive Formulation

CASE 2: when $i < j$

Split $A_{i \dots j}$ into $A_{i \dots k}$ and $A_{k+1 \dots j}$ substructures

Thus

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}\, p_k\, p_j$$

$$\text{for } k = i, i+1, i+2, \dots, j-1$$

Check all possible values k and pick the best one

Recursive Formulation:

$$\boldsymbol{m[i,j] = \begin{cases} 0 & if\ i == j \\ \min\limits_{i \le k < j}\left(m[i,k] + m[k+1,j] + p_{i-1}\, p_k\, p_j\right) & \\ & if\ i < j \end{cases}}$$

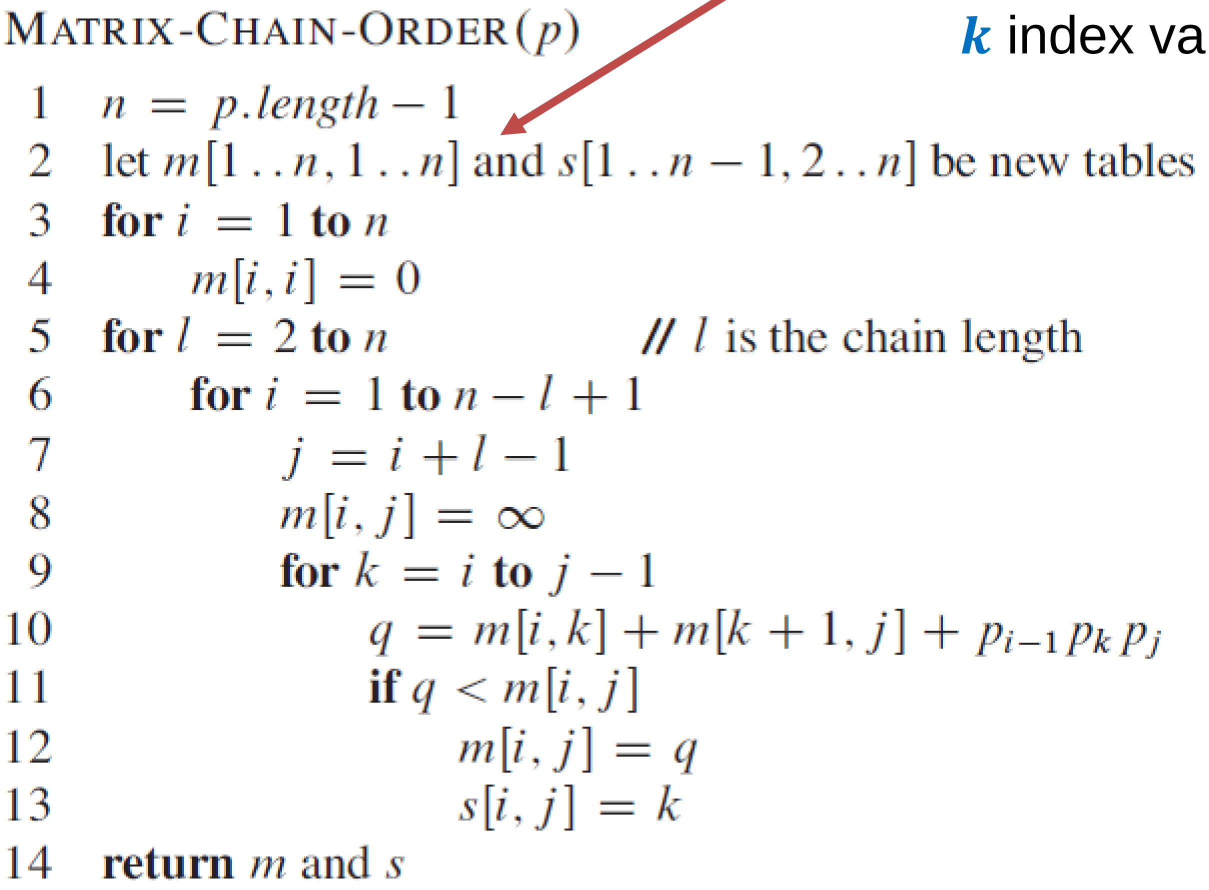
Steps for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the matrix-chain multiplication
- **Step 3:** Use dynamic programming to find the minimum product in matrix-chain multiplication.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual parenthesis multiplication.

Step 2: Dynamic Programming (Bottom-Up)

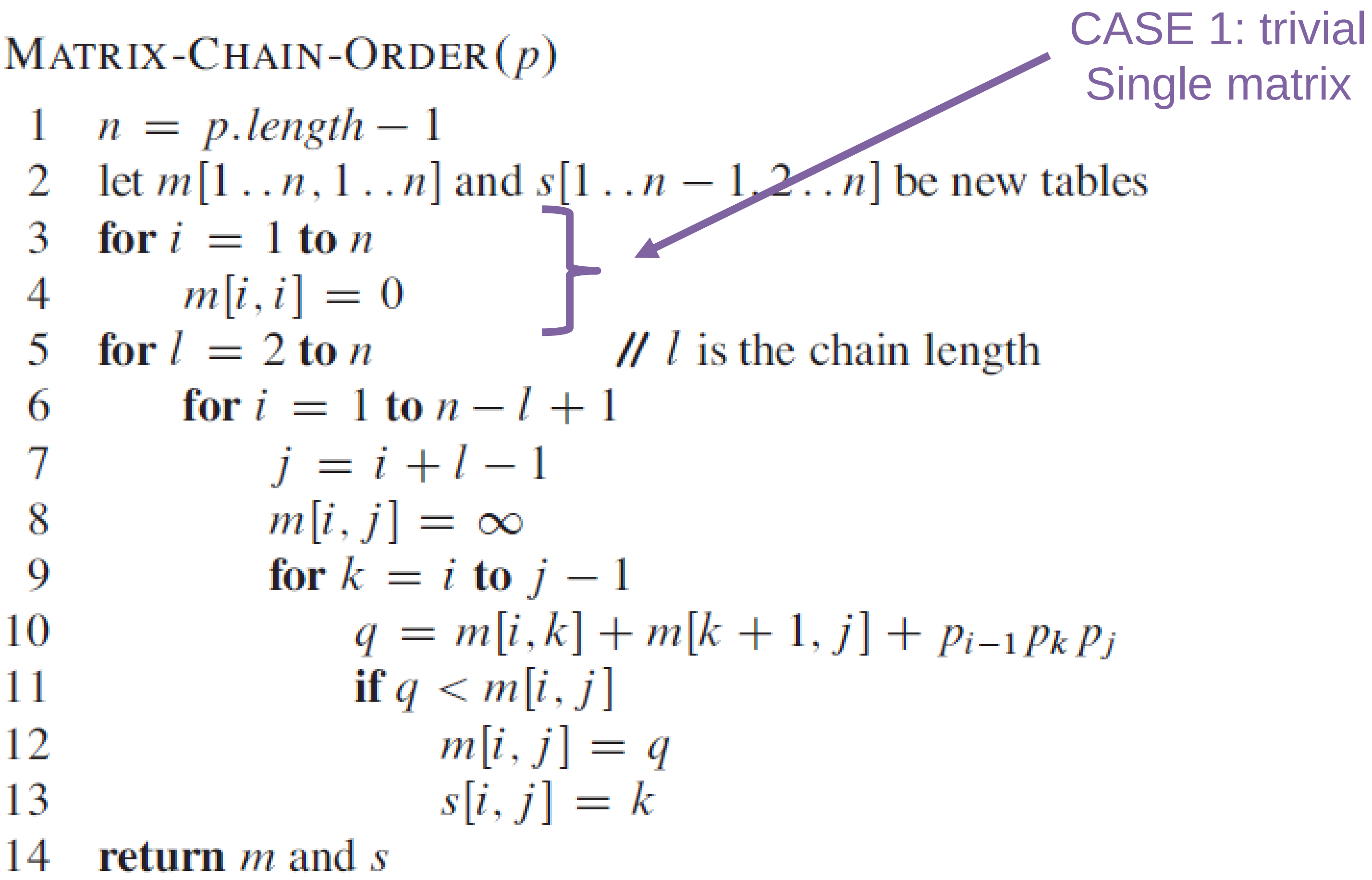
Uses two auxiliary tables

- $\boldsymbol{m[1\ \dots n, 1 \dots n]}$ for storing $\boldsymbol{m[i, j]}$
- $\boldsymbol{s\ \ [1\ \dots n, -1\ 2 \dots n]}$ for storing $\boldsymbol{k}$ index value



```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                    // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + $p_{i-1} p_k p_j$
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

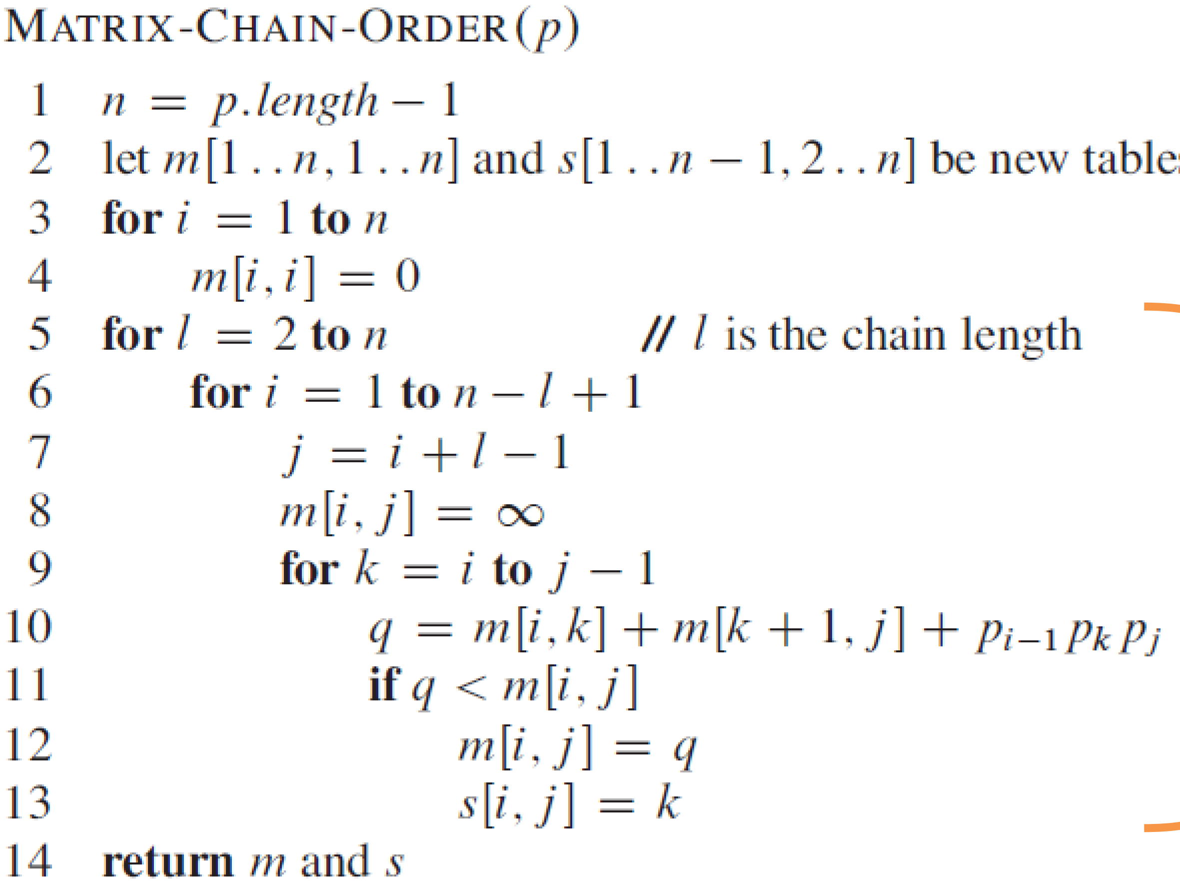
Step 2: Dynamic Programming (Bottom-Up)



```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                  // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + $p_{i-1} p_k p_j$
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

CASE 1: trivial
Single matrix

Step 2: Dynamic Programming (Bottom-Up)



```
MATRIX-CHAIN-ORDER($p$)
$n = p.length - 1$
let $m[1..n, 1..n]$ and $s[1..n-1, 2..n]$ be new tables
for $i = 1$ to $n$
    $m[i, i] = 0$
for $l = 2$ to $n$                    // $l$ is the chain length
    for $i = 1$ to $n - l + 1$
        $j = i + l - 1$
        $m[i, j] = \infty$
        for $k = i$ to $j - 1$
            $q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$
            if $q < m[i, j]$
                $m[i, j] = q$
                $s[i, j] = k$
return $m$ and $s$
```

CASE 2: $i < j$

Steps for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the matrix-chain multiplication
- **Step 3:** Use dynamic programming to find the minimum product in matrix-chain multiplication.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual parenthesis multiplication from the auxiliary table $\boldsymbol{s}$.

Matrix-chain Multiplication

Example: Given a sequence of four matrices A_1, A_2, A_3, A_4 with $p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$.

Matrix-chain Multiplication

Example: Given a sequence of four matrices A_1, A_2, A_3, A_4 with $p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$.

The Optimal Solution $\left((A_1)(A_2\,A_3)\right)(A_4)$:

$$(A_2\,A_3) = 4 \times 6 \times 2 = 48$$
$$(A_1)(A_2\,A_3) = 5 \times 4 \times 2 = 40$$
$$\left((A_1)(A_2\,A_3)\right)(A_4) = 5 \times 2 \times 7 = 70$$

$$\text{Total Multiplication } \left((A_1)(A_2\,A_3)\right)(A_4) = 48 + 40 + 70 = 158$$

Matrix-chain Multiplication

Example: Given a sequence of four matrices A_1, A_2, A_3, A_4 with $p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$.

		i			
		1	2	3	4
	4				0
	3			0	
j	2		0		
	1	0			

$\boldsymbol{m}$

		i		
		1	2	3
	4			
j	3			
	2			

$\boldsymbol{s}$

Matrix-chain Multiplication

$p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$

$$m[1,2] = \min_{1 \le k < 2} (m[1,k] + m[k+1,2] + p_0\, p_k\, p_2)$$
$$= m[1,1] + m[2,2] + p_0\, p_1\, p_2 = 120$$

i

		1	2	3	4
	4				0
	3			0	
j	2	120	0		
	1	0			

m

i

		1	2	3
	4			
j	3			
	2	1		

s

Matrix-chain Multiplication

$p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$.

$$m[2,3] = \min_{2 \le k < 3} (m[2,k] + m[k+1,3] + p_1 \, p_k \, p_3)$$
$$= m[2,2] + m[3,3] + p_1 \, p_2 \, p_3 = 48$$

i

j	1	2	3	4
4				0
3		48	0	
2	120	0		
1	0			

m

i

j	1	2	3
4			
3		2	
2	1		

s

Matrix-chain Multiplication

$p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$.

$$\begin{aligned} m[3,4] &= \min_{3 \le k < 4} (m[3,k] + m[k+1,4] + p_2\, p_k\, p_4) \\ &= m[3,3] + m[4\;4] + p_2\, p_3\, p_4 = 84 \end{aligned}$$

i

		1	2	3	4
	4			84	0
	3		48	0	
j	2	120	0		
	1	0			

m

i

		1	2	3
	4			3
j	3		2	
	2	1		

s

Matrix-chain Multiplication

$p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$.

$$m[1,3] = \min_{1\le k<3}(m[1,k] + m[k+1,3] + p_0\,p_k\,p_3)$$

$$= \min\begin{cases} m[1,1] + m[2,3] + p_0\,p_1\,p_3 = 88 \\ m[1,2] + m[3,3] + p_0\,p_2\,p_3 = 180 \end{cases}$$

$$= 88$$

i

j	1	2	3	4
4			84	0
3	88	48	0	
2	120	0		
1	0			

m

i

j	1	2	3
4			3
3	1	2	
2	1		

s

Matrix-chain Multiplication

$p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$.

$$m[2,4] = \min_{2 \le k < 4} (m[2,k] + m[k+1,4] + p_1\, p_k\, p_4)$$

$$= min \begin{cases} m[2,2] + m[3\ 4] + p_1\, p_2\, p_4 = 252 \\ m[2,3] + m[4,4] + p_1\, p_3\, p_4 = 104 \end{cases}$$

$$= 104$$

i

j	1	2	3	4
4		104	84	0
3	88	48	0	
2	120	0		
1	0			

m

i

j	1	2	3
4		3	3
3	1	2	
2	1		

s

Matrix-chain Multiplication

$p_0 = 5, p_1 = 4, p_2 = 6, p_3 = 2$ and $p_4 = 7$.

$$m[1,4] = \min_{1 \le k < 4} (m[1,k] + m[k+1,4] + p_0\, p_k\, p_4)$$

$$= \min \begin{cases} m[1,1] + m[2,4] + p_0\, p_1\, p_4 = 244 \\ m[1,2] + m[3,4] + p_0\, p_2\, p_4 = 294 \\ m[1,3] + m[4,4] + p_0\, p_3\, p_4 = 158 \end{cases} = 158$$

j	1	2	3	4
4	158	104	84	0
3	88	48	0	
2	120	0		
1	0			

$\boldsymbol{m}$

j	1	2	3
4	3	3	3
3	1	2	
2	1		

$\boldsymbol{s}$

Steps for applying Dynamic Programming

Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual parenthesis multiplication from the auxiliary table $\boldsymbol{s}$.

$\boldsymbol{s}\,[\mathbf{1},\mathbf{4}] = 3$ $\quad (\boldsymbol{A_1 A_2 A_3})(\boldsymbol{A_4})$

$\boldsymbol{s}\,[\mathbf{1},\mathbf{3}] = 1$ $\quad ((\boldsymbol{A_1})(\boldsymbol{A_2 A_3}))(\boldsymbol{A_4})$

		1	2	3
	4	3	3	3
j	3	1	2	
	2	1		

$\boldsymbol{s}$

Solution:

$\text{Total Multiplication} = \boldsymbol{m}[\mathbf{1},\mathbf{4}] = 158$

$\big((\boldsymbol{A_1})(\boldsymbol{A_2 A_3})\big)(\boldsymbol{A_4})$

Steps for applying Dynamic Programming

Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual parenthesis multiplication from the auxiliary table $\boldsymbol{s}$.

$\boldsymbol{s}\,[\mathbf{1},\mathbf{4}] = 3$ $\quad (\boldsymbol{A_1 A_2 A_3})(\boldsymbol{A_4})$

$\boldsymbol{s}\,[\mathbf{1},\mathbf{3}] = 1$ $\quad ((\boldsymbol{A_1})(\boldsymbol{A_2 A_3}))(\boldsymbol{A_4})$

		1	2	3
	4	3	2	3
j	3	1	2	
	2	1		

$\boldsymbol{s}$

$p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$

$$(\boldsymbol{A_2 A_3}) = \mathbf{4 \times 6 \times 2 = 48}$$

$$(\boldsymbol{A_1})(\boldsymbol{A_2 A_3}) = \mathbf{5 \times 4 \times 2 = 40}$$

$$\big((\boldsymbol{A_1})(\boldsymbol{A_2 A_3})\big)(\boldsymbol{A_4}) = \mathbf{5 \times 2 \times 7 = 70}$$

$\text{Total Multiplication} = 48 + 40 + 70 = 158$

Steps for applying Dynamic Programming

Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual parenthesis multiplication from the auxiliary table $\boldsymbol{s}$.

Algorithm (refer Cormen book for detail)

```
PRINT-OPTIMAL-PARENS($s, i, j$)
if $i$ == j
    print “$A$” $_i$
else print “(”
    PRINT-OPTIMAL-PARENS($s, i, s[i, j]$)
    PRINT-OPTIMAL-PARENS ($s, s[i, j] + 1, j$)
print “)”
```