Dynamic Programming

Dr Dayal Kumar Behera

A method for solving complex problems by breaking them into smaller, easier, sub problems.

Term *Dynamic Programming* coined by mathematician Richard E. Bellman in early 1950s.

"I thought ***dynamic programming*** was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities"

- Richard E. Bellman

What is Dynamic Programming?

- Dynamic programming solves *optimization problems* by combining solutions to subproblems

- "Programming" refers to a tabular method with a series of choices, not "coding"

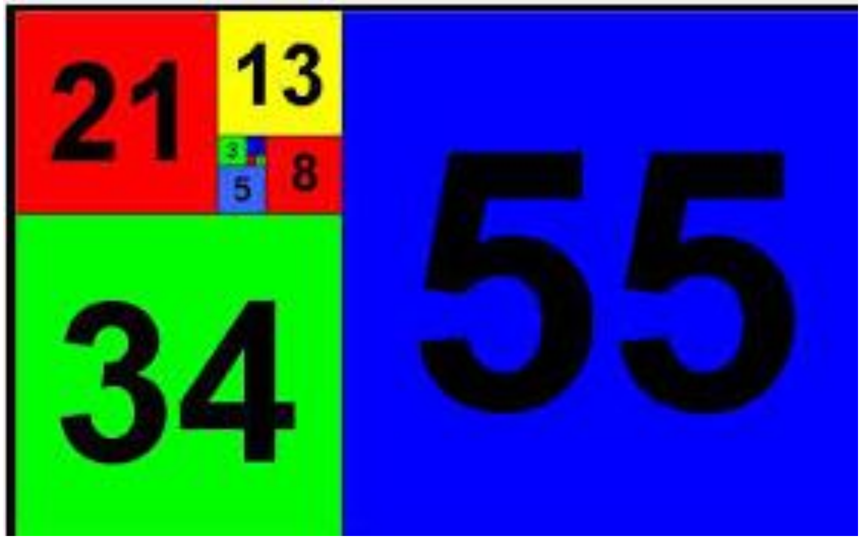
Break big problem up into smaller problems ...

Sound familiar?

Recursion?
Problems with Recursion...

Fibonacci Series: An Example

Using Recursion



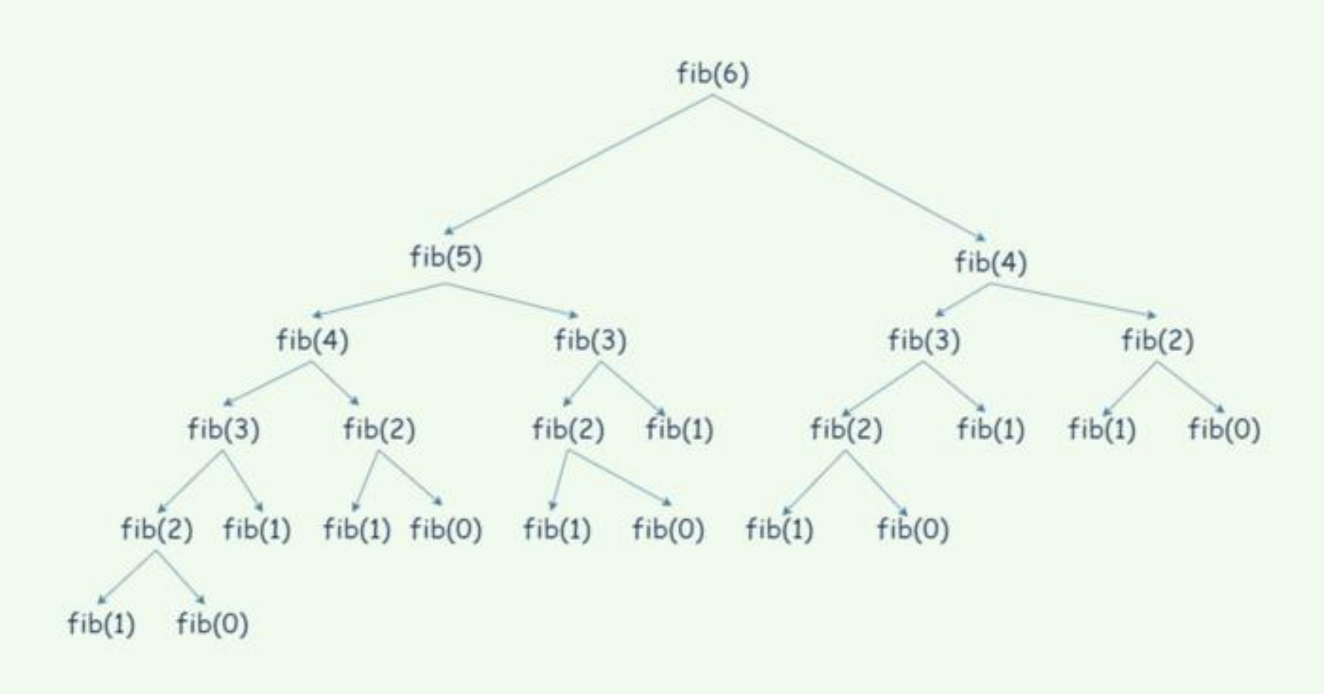
$$fib(0) = 0$$

$$fib(1) = 1$$

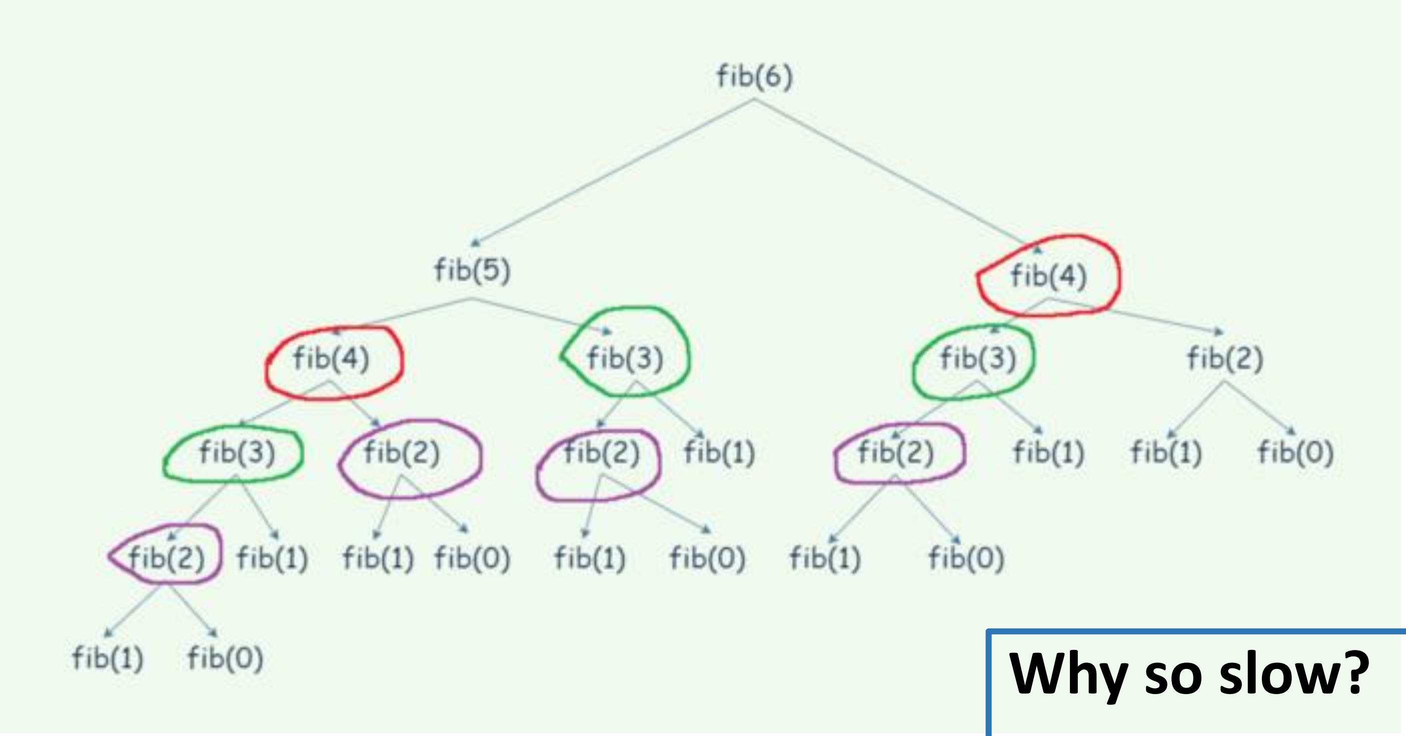
$$fib(n) = fib(n-1) + fib(n-2) \text{ if } n > 1$$

Problem With Recursion

```
1th fibonnaci number: 1 -  Time: 4.467E-6
2th fibonnaci number: 1 -  Time: 4.47E-7
3th fibonnaci number: 2 -  Time: 4.46E-7
4th fibonnaci number: 3 -  Time: 4.46E-7
5th fibonnaci number: 5 -  Time: 4.47E-7
6th fibonnaci number: 8 -  Time: 4.47E-7
7th fibonnaci number: 13 -  Time: 1.34E-6
8th fibonnaci number: 21 -  Time: 1.787E-6
9th fibonnaci number: 34 -  Time: 2.233E-6
10th fibonnaci number: 55 -  Time: 3.573E-6
11th fibonnaci number: 89 -  Time: 1.2953E-5
12th fibonnaci number: 144 -  Time: 8.934E-6
13th fibonnaci number: 233 -  Time: 2.9033E-5
14th fibonnaci number: 377 -  Time: 3.7966E-5
15th fibonnaci number: 610 -  Time: 5.0919E-5
16th fibonnaci number: 987 -  Time: 7.1464E-5
17th fibonnaci number: 1597 -  Time: 1.08984E-4
36th fibonnaci number: 14930352 -  Time: 0.045372057
37th fibonnaci number: 24157817 -  Time: 0.071195386
38th fibonnaci number: 39088169 -  Time: 0.116922086
39th fibonnaci number: 63245986 -  Time: 0.186926245
40th fibonnaci number: 102334155 -  Time: 0.308602967
41th fibonnaci number: 165580141 -  Time: 0.498588795
42th fibonnaci number: 267914296 -  Time: 0.793824734
43th fibonnaci number: 433494437 -  Time: 1.323325593
44th fibonnaci number: 701408733 -  Time: 2.098209943
45th fibonnaci number: 1134903170 -  Time: 3.392917489
46th fibonnaci number: 1836311903 -  Time: 5.506675921
47th fibonnaci number: -1323752223 -  Time: 8.803592621
48th fibonnaci number: 512559680 -  Time: 14.295023778
49th fibonnaci number: -811192543 -  Time: 23.030062974
50th fibonnaci number: -298632863 -  Time: 37.217244704
51th fibonnaci number: -1109825406 -  Time: 60.224418869
```



Slow Fibonacci



Why so slow?

- Algorithm keeps calculating the same value over and over
- When calculating the 40^{th} Fibonacci number the algorithm calculates the 4^{th} Fibonacci number **24,157,817** times!!!

DP vs. D-n-C

- Divide-and-Conquer(D-n-C) algorithms partition, the problem into independent sub-problems. Solve the sub-problems recursively and then combine their solutions to solve the original problem.

- In contrast, Dynamic Programming(DP) is applicable when the sub-problems are not independent i.e. when sub-problems share sub-sub-problems.

Dynamic Programming: Approach

- Given Problem is divided into number of interrelated over lapping Sub-problems

- DP algorithm solves every sub-sub-problem just once and saves the answer in a table, thereby avoiding the work of re-computing the answer every time the sub-sub-problem is encountered .

- The solution of the sub-problem are combined in a bottom –up approach to obtain the final solution.

Multistage Optimization

- Dynamic programming is useful in case of multi stage optimization problems
- Each Optimization Problem has an objective function and a set of constraints/ restrictions.
- Optimization problem deals with the maximization or minimization of the objective function.
- In multistage optimization problem, decisions are taken at multiple stages to obtain a global solution.

Components of Dynamic Programming

Stages : Given problem can be divided into a number of sub problems called stages.

- division of problem into number of sub-problems should be done in polynomial time.
- Its also referred as polynomial breakup.

Decision: In each stage there can be multiple decisions, out of which the best decision should be taken.

- A decision taken at every stage should be optimal.

State: A state indicates the sub problem for which decision needs to be taken.

- The variables that are used to take decision at every stage are called state variables
- Number of state variables should be as small as possible.

Components of Dynamic Programming

Policy: Policy is a rule that determines the decision at each stage

- A policy is called optimal, if it is globally optimal
- This is called the Bellmann's Principle of Optimality

Principle of Optimality

- The core principle of Dynamic Programming is 'Principle of Optimality'

  It states that the optimal sequence of decisions in a multistage decision problem is feasible if and only if its sub-sequences are optimal.

Steps of Dynamic Programming

Step 1: Characterize the **structure** of an optimal solution

Step 2: Recursively define (iterative evaluation) the value of an optimal solution.

Step 3: Compute the value of an optimal solution in a **bottom-up** fashion

Step 4: Construct an **optimal solution** from computed information.

Characteristic/Elements of Dynamic Programming

1. Overlapping Sub-problem:

- One of the main characteristics of dynamic programming is to spilt the problem into sub-problems.
- But unlike divide and conquer approach here many subproblems overlap and can not be treated distinctly

Characteristic/Elements of Dynamic Programming

Two ways of handling overlapping problems

1.1. **Memoization Technique:** This method looks into a table to check whether the table has any entry or not.

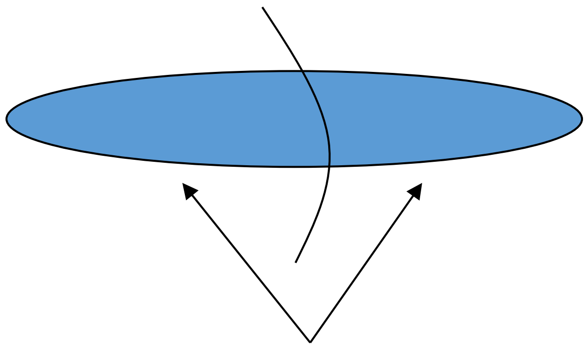
- Initially all entries are filled with NIL or undefined
- If no value is present then it is computed
- Here the computation flows in a top-down method.

1.2. **Tabulation Method:** Here is the problem is solved from scratch

- The smallest subproblem is solved and it is stored in the table.
- Its value is used in the table. Its value is used later for solving larger problem
- Computation follows a Bottom-up method.

Characteristic/Elements of Dynamic Programming

2. Optimal Substructures:



Each substructure is optimal.

(Principle of optimality)

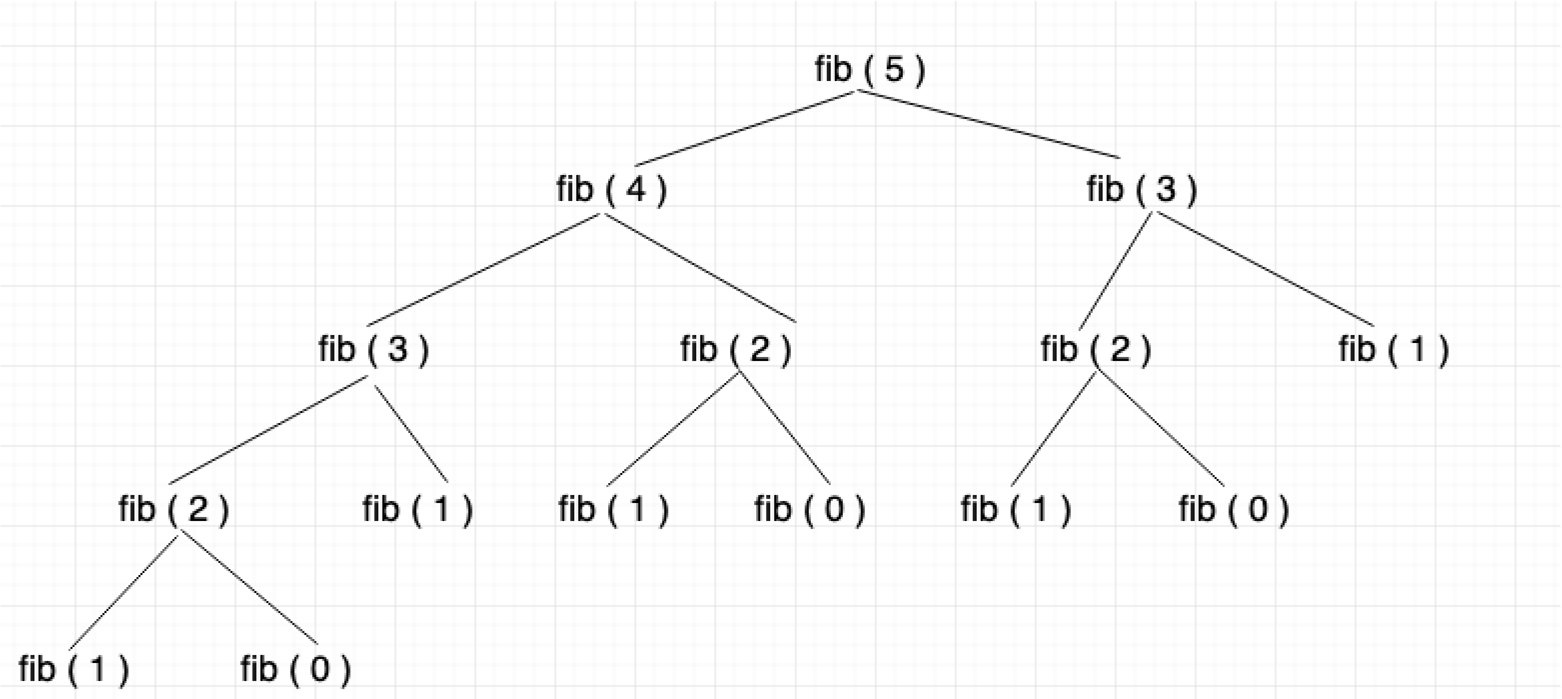
- An optimal solution to a problem contains optimal solution to each of its sub-problems.
- Optimal solution to the entire problem is build in a bottom-up manner from optimal solutions to sub-problems

Algorithm Fibonacci(n): D-n-C

Divide and Conquer Approach

```
Algorithm fib(n)
Begin
        if((n==0) or (n==1)) then
                return n
        else
                return fib(n-1) + fib(n-2)
        end if
End
```

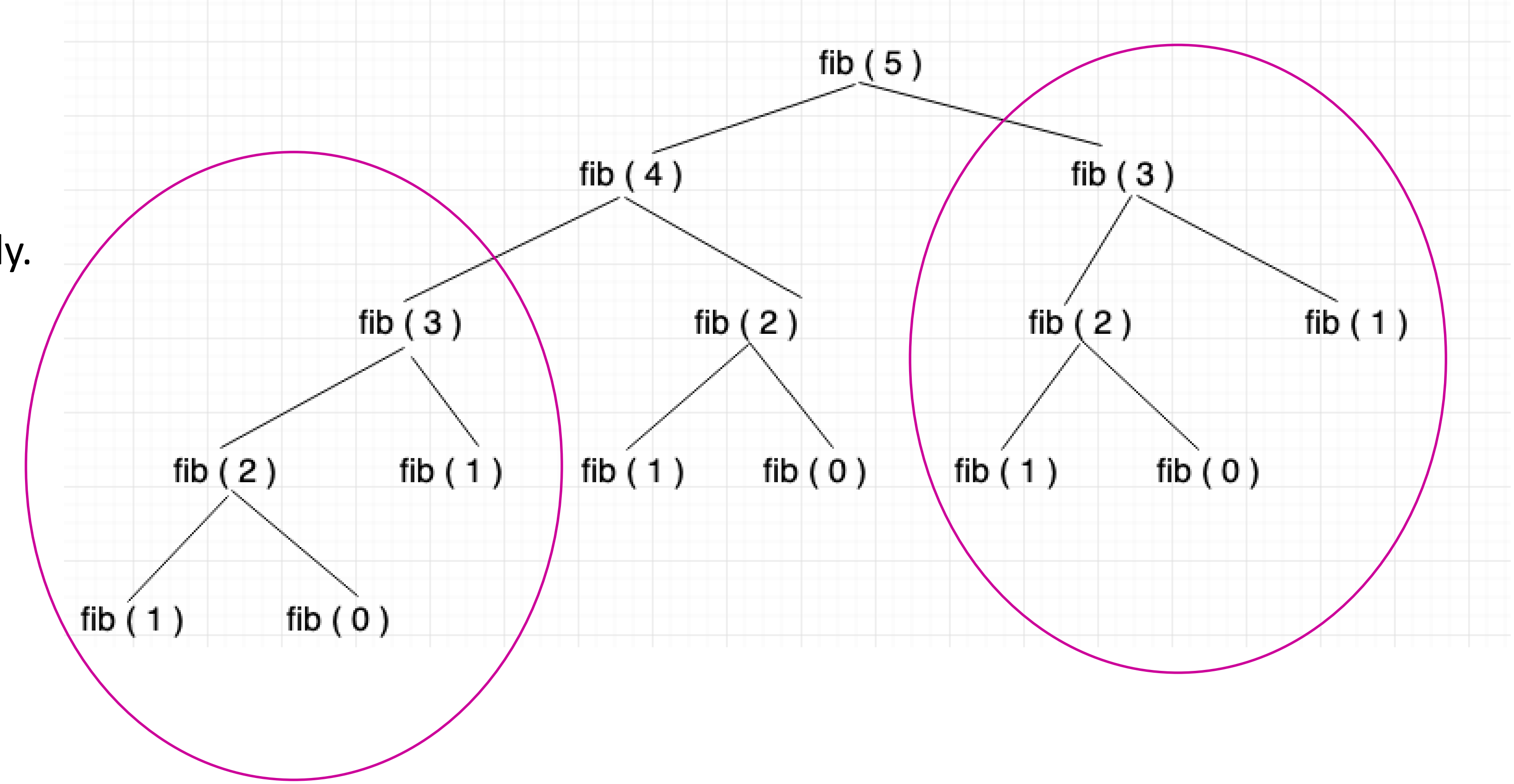
Computing fib(5): D-n-C



Computing fib(5): D-n-C

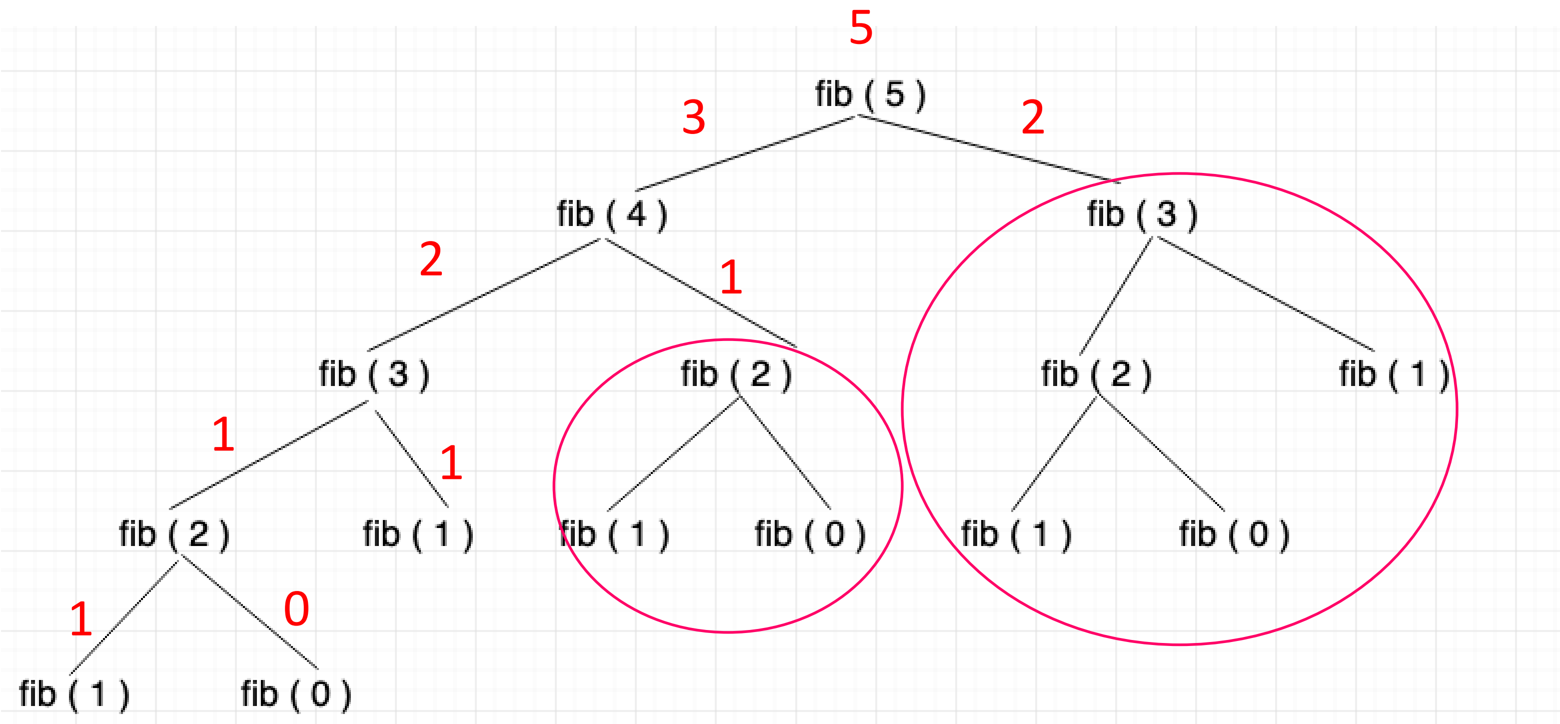
Overlapping Sub-problems:

- Wasteful re-computation
- Computation trees grows exponentially.



Never re-evaluate a sub problem.

Computing fib(5): Memoization



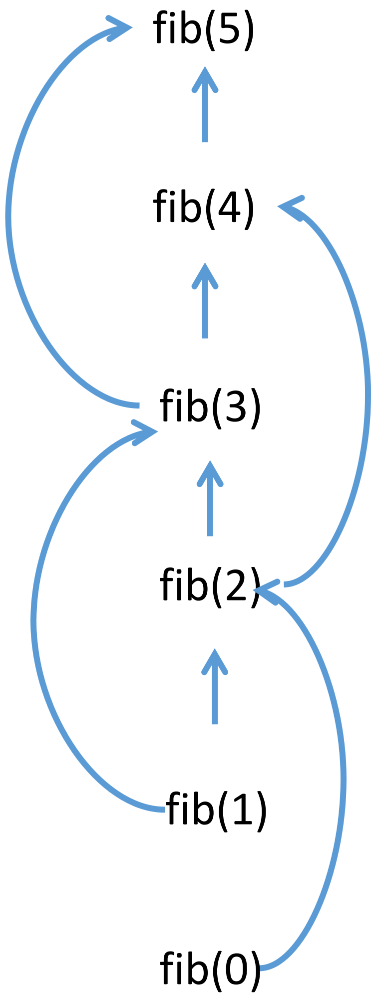
k	fib(k)
0	0
1	1
2	1
3	2
4	3
5	5

Algorithm Fibonacci(n): Memoization

```
Algorithm mem_fib(n)            // top-down approach
Begin
    if fibtable[n] then
        return fibtable[n]
    if n==0 or n==1
        value = n
    else
        value = mem_fib(n-1) +  mem_fib(n-2)
        fibtable[n] = value
    return value
End
```

Computing fib(5): DP

- Anticipate what the memory table look like
  - Sub-problems are known from problem structure
  - Dependencies form a DAG.
- Solve sub problems in topological order.



k	fib(k)
0	0
1	1
2	1
3	2
4	3
5	5

Algorithm Fibonacci(n): DP or Tabulation

```
Algorithm dp_fib(n)                    // bottom-up approach
Begin
        fibtable[0] = 0
        fibtable[1] = 1
        for i=2 to n
                fibtable[i] = fibtable[i-1]+fibtable[i-2]

        return fibtable[n]
End
```

Memoization vs. DP

- Memoization:
    - Store values of sub-problems in a table.
    - Look up the table before making a recursive call.
    - Recursive Evaluation

- Dynamic Programming:
    - Solve sub-problems in topological order of dependency.
    - Dependencies must form a DAG
    - Iterative Evaluation

Matrix-Chain Multiplication/Product

Given a sequence $\langle A_1, A_2, \ldots, A_n \rangle$, compute the product:

$$A_1 \cdot A_2 \cdots A_n$$

- Matrix Multiplications are not Commutative but Associative.
- In what order should we multiply the matrices?
- Parenthesize the product to get the order in which matrices are multiplied

E.g.: $A_1 \cdot A_2 \cdot A_3 = ((A_1 \cdot A_2) \cdot A_3) = (A_1 \cdot (A_2 \cdot A_3))$

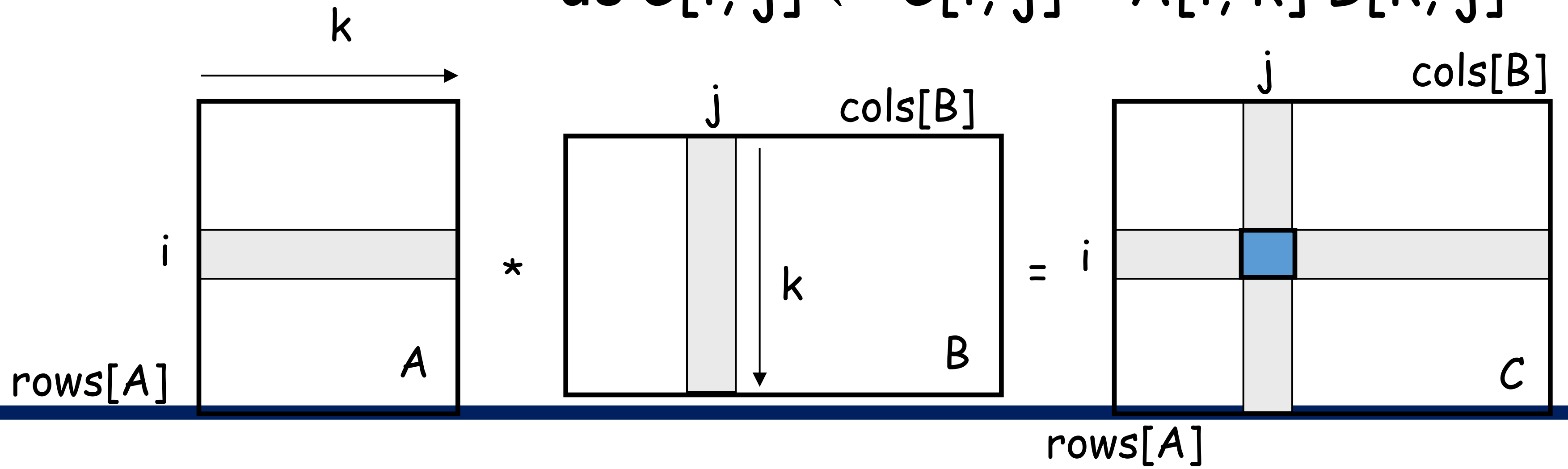
  - The order in which we multiply the matrices has a significant impact on the cost of evaluating the product

Matrix-chain Multiplication Problem

- Given a chain of “n” matrices $\langle A_1, A_2, \ldots, A_n \rangle$,
- where for i = 1, 2, …, n, matrix A_i has dimension $P_{i-1} x\ P_i$
- Fully parenthesize the product $A_1 \cdot A_2 \cdots A_n$ in a way that minimizes the cost (number of scalar multiplication).

MATRIX-MULTIPLY(A, B)

if columns[A] ≠ rows[B]
    then error "incompatible dimensions"
else for i ← 1 to rows[A]
    do for j ← 1 to columns[B]
        do C[i, j] = 0
          for k ← 1 to columns[A]
            do $C[i, j] \leftarrow C[i, j] + \overbrace{A[i, k]\, B[k, j]}^{\text{rows}[A] \cdot \text{cols}[A] \cdot \text{cols}[B] \text{ multiplications}}$



Matrix Compatibility for Multiplication

$$\{ A_1 \}_{pxq} \text{ X } \{ A_2 \}_{qxr} = \{ A \}_{pxr}$$

No of scalar multiplications = p x q x r

Dimension of resultant matrix = q x r

$$A_1 \cdot A_2 \cdots A_i \cdot A_{i+1} \cdots A_n$$

$$\{P_0 x P_1,\ P_1 x P_2,\ \ldots\ P_{i-1} x P_i,\ \ P_i x P_{i+1}\ \ldots P_{n-1} x P_n\}$$

$$\{P_0\ P_1\ P_2\ \ \ldots\ \ P_{i-1}\ P_i\ \ P_{i+1}\ \ \ldots\ \ P_{n-1}\ P_n\}$$

No of Multiplication $\underset{P_{i-1}P_i}{A_i} \cdot \underset{P_i\ P_{i+1}}{A_{i+1}}$

$$P_{i-1} \cdot P_i \cdot P_{i+1}$$

What is the dimension(size) of matrix for the chain : $A_i \cdot A_{i+1} \cdots A_n$

$P_{i-1} \text{ x } P_n$

Matrix Chain Multiplication

A : 2 x 3

B : 3 x 4

C : 4 x 5

A X B X C

Two Possible Ordering

((A B) C)
(A (B C))

[(A B) C] = (2x3x4) + (2x4x5) = 24 + 40 = 64

[A (B C)] = (3x4x5) + (2x3x5) = 60 + 30 = 90

So, the optimal order is [(A B) C]

Matrix Chain Multiplication

A X B X C X D

A : 2 x 3

B : 3 x 4

C : 4 x 3

D: 3 x 2

Five Possible Ordering

A(B(CD))

A((BC)D)

(AB)(CD)

(A(BC))D

((AB)C)D

<

[A(B(CD))] = (4x3x2) + (3x4x2) + (2x3x2) = 24 + 24 + 12 = 60

[A((BC)D)] = (3x4x3) + (3x3x2) + (2x3x2) = 36 + 18 + 12 = 66

[(AB)(CD)] = (2x3x4) + (4x3x2) + (2x4x2) = 24 + 24 + 16 = 64

[(A(BC))D] = (3x4x3) + (2x3x3) + (2x3x2) = 36 + 18 + 12 = 66

[((AB)C)D] = (2x3x4) + (2x4x3) + (2x3x2) = 24 + 24 + 12 = 60

Optimal Ordering

The Structure of an Optimal Parenthesization

Notation:

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

For i < j:

Suppose that an optimal parenthesization of $A_{i\ldots j}$ splits the product between A_k and A_{k+1}, where $i \leq k < j$

$$\begin{aligned} A_{i\ldots j} &= A_i\ A_{i+1} \cdots A_j \\ &= A_i\ A_{i+1} \cdots A_k\ A_{k+1} \cdots A_j \\ &= A_{i\ldots k}\ A_{k+1\ldots j} \end{aligned}$$

MCP Dynamic Programming Steps

Step 1: structure of an optimal parenthesization

- Let $\mathbf{A}_{i..j}$ ($i \le j$) denote the matrix resulting from $A_i \times A_{i+1} \times \ldots \times A_j$
- Any parenthesization of $A_i \times A_{i+1} \times \ldots \times A_j$ must split the product between A_k and A_{k+1} for some k, ($i \le k < j$).
- The cost = # of computing $A_{i..k}$ + # of computing $A_{k+1..j}$ + # $A_{i..k} \times A_{k+1..j}$.
- If k is the position for an optimal parenthesization, the parenthesization of "prefix" subchain $A_i \times A_{i+1} \times \ldots \times A_k$ within this optimal parenthesization of $A_i \times A_{i+1} \times \ldots \times A_j$ must be an optimal parenthesization

$$\underbrace{\underbrace{A_i \times A_{i+1} \times \ldots \times A_k}_{} \times \underbrace{A_{k+1} \times \ldots \times A_j}_{}}_{}$$

Optimal Substructure

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

- The parenthesization of the "prefix" $A_{i\ldots k}$ must be an optimal parenthesization
- If there were a less costly way to parenthesize $A_{i\ldots k}$, we could substitute that one in the parenthesization of $A_{i\ldots j}$ and produce a parenthesization with a lower cost than the optimum.
- *An optimal solution to an instance of the matrix-chain multiplication contains within it optimal solutions to subproblems*

A Recursive Formula/Solution

Subproblem:

determine the minimum cost of parenthesizing $A_{i\ldots j} = A_i A_{i+1} \cdots A_j$

for $1 \le i \le j \le n$

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

- Full problem ($A_{1..n}$): m[1, n]
- $i = j$: $A_{i\ldots i} = A_i \Rightarrow m[i, i] = 0$, for i = 1, 2, …, n

A Recursive Formula/Solution

- Consider the subproblem of parenthesizing

- $A_{i\ldots j} = A_i \, A_{i+1} \cdots A_j$ for $1 \leq i \leq j \leq n$

  $p_{i-1}p_kp_j$

  $= A_{i\ldots k} \, A_{k+1\ldots j}$ for $i \leq k < j$

  m[i, k] m[k+1,j]

- Assume that the optimal parenthesization splits the product $A_i \, A_{i+1} \cdots A_j$ at k ($i \leq k < j$)

$$m[i, j] = \underbrace{m[i, k]}_{\text{min \# of multiplications to compute } A_{i\ldots k}} + \underbrace{m[k+1, j]}_{\text{min \# of multiplications to compute } A_{k+1\ldots j}} + \underbrace{p_{i-1}p_kp_j}_{\text{\# of multiplications to compute } A_{i\ldots k}A_{k\ldots j}}$$

A Recursive Formula/Solution

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

- We do not know the value of k
  - There are j – i possible values for k: **k = i, i+1, …, j-1**
- Minimizing the cost of parenthesizing the product $A_i\, A_{i+1} \cdots A_j$ becomes:

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

Reconstructing the Optimal Solution

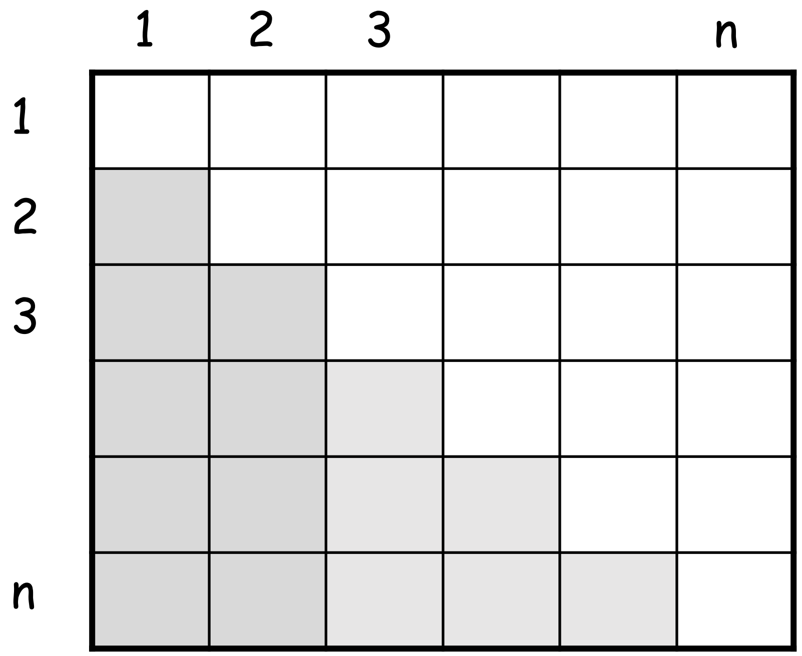
Additional information to maintain:

s[i, j] = value of k for which the cost of parenthesizing $A_i\ A_{i+1} \cdots A_j$ is minimum.

Computing the Optimal Costs

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

- How many subproblems do we have?
  - Parenthesize $A_{i \ldots j}$ for $1 \le i \le j \le n$
  - One problem for each choice of i and j



Computing the Optimal Costs (cont.)

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

- How do we fill in the tables m[1..n, 1..n] and s[1..n, 1..n] ?
  - Determine which entries of the table are used in computing $m[i, j]$

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

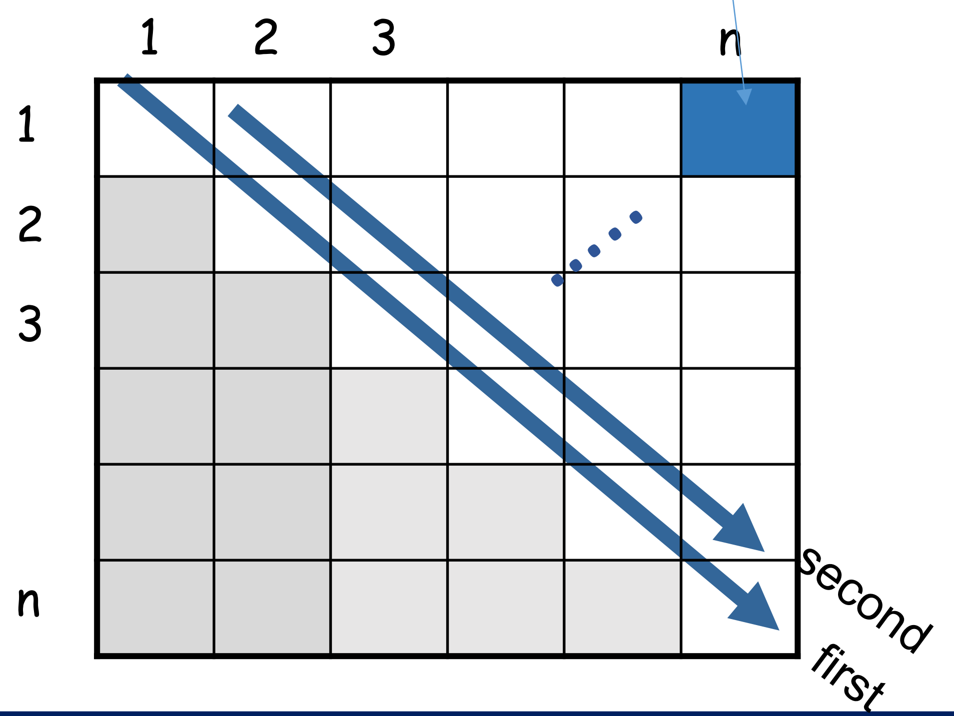
  - Fill in m such that it corresponds to solving problems of increasing length

Computing the Optimal Costs (cont.)

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

- Length = 1: $j = i$, $i = 1, 2, \ldots, n$
- Length = 2: $j = i + 1$, $i = 1, 2, \ldots, n-1$

$m[1, n]$ gives the optimal solution to the problem



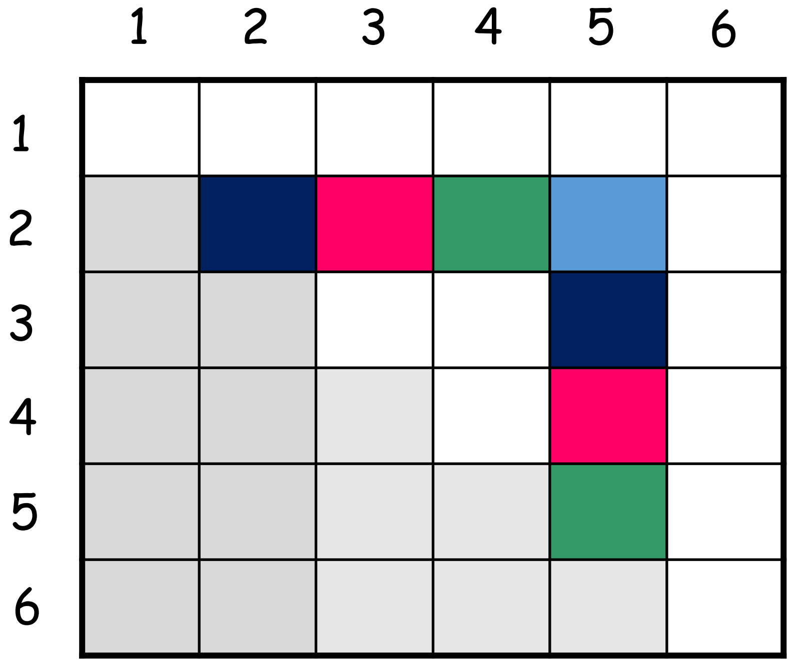
Compute rows from top to bottom and from left to right

In a similar matrix s we keep the optimal values of k

Example: $\min \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\}$

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1p_2p_5 & k = 2 \\ m[2,3] + m[4,5] + p_1p_3p_5 & k = 3 \\ m[2,4] + m[5,5] + p_1p_4p_5 & k = 4 \end{cases}$$

- Values $m[i, j]$ depend only on values that have been previously computed



i

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

A_1	A_2	A_3	A_4
4x5	5x3	3x2	2x7
P_0P_1	P_1P_2	P_2P_3	P_3P_4

Length = 1: $j = i$, $i = 1, 2, \ldots, n$

When i==j then M[i, j] =0

$\Rightarrow$ M[1,1] =0

$\Rightarrow$ M[2,2] =0

$\Rightarrow$ M[3,3] =0

$\Rightarrow$ M[4,4] =0

M

	1	2	3	4
1	0			
2		0		
3			0	
4				0

S

	1	2	3	4
1	0			
2		0		
3			0	
4				0

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

A_1	A_2	A_3	A_4
4x5	5x3	3x2	2x7
P_0P_1	P_1P_2	P_2P_3	P_3P_4

For the Second Super Diagonal

Length = 2: $j = i+1$, $i = 1, 2, \ldots, n-1$

$\Rightarrow$ $M[1,2] = M[1,1] + M[2,2] + P_0P_1P_2 = 0+0+60=60$ (k=1)

$\Rightarrow$ $M[2,3] = M[2,2] + M[3,3] + P_1P_2P_3 = 0+0+30=30$ (k=2)

$\Rightarrow$ $M[3,4] = M[3,3] + M[4,4] + P_2P_3P_4 = 0+0+42=42$ (k=3)

M

	1	2	3	4
1	0	60		
2		0	30	
3			0	42
4				0

S

	1	2	3	4
1	0	1		
2		0	2	
3			0	3
4				0

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

A_1	A_2	A_3	A_4
4x5	5x3	3x2	2x7
P_0P_1	P_1P_2	P_2P_3	P_3P_4

For the Third Super Diagonal

Length = 3: $j = i+2$, $i = 1, 2, \ldots, n-2$

M

	1	2	3	4
1	0	60	70	
2		0	30	100
3			0	42
4				0

$\Rightarrow$ $M[1,3] = M[1,1] + M[2,3] + P_0P_1P_3 = 0+30+40=70$ (k=1)

Or $M[1,3] = M[1,2] + M[3,3] + P_0P_2P_3 = 60+0+24=84$ (k=2)

S

	1	2	3	4
1	0	1	1	
2		0	2	3
3			0	3
4				0

$\Rightarrow$ $M[2,4] = M[2,2] + M[3,4] + P_1P_2P_4 = 0+42+105=147$ (k=2)

Or $M[2,4] = M[2,3] + M[4,4] + P_1P_3P_4 = 30+0+70=100$ (k=3)

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

A_1	A_2	A_3	A_4
4x5	5x3	3x2	2x7
P_0P_1	P_1P_2	P_2P_3	P_3P_4

For the Forth Super Diagonal

Length = 4: $j = i+3$, $i = 1, 2, \ldots, n-3$

$\Rightarrow$ $M[1,4] = M[1,1] + M[2,4] + P_0P_1P_4$
$= 0+100+140=240$ (k=1)

Or $M[1,4] = M[1,2] + M[3,4] + P_0P_2P_4$
$= 60+42+84=186$ (k=2)

Or $M[1,4] = M[1,3] + M[4,4] + P_0P_3P_4$
$= 74+0+56=126$ (k=3)

M

	1	2	3	4
1	0	60	74	126
2		0	30	100
3			0	42
4				0

S

	1	2	3	4
1	0	1	1	3
2		0	2	3
3			0	3
4				0

MATRIX-CHAIN-ORDER(p)

```
n ← length[p] - 1
for i ← 1 to n
    do m[i, i] ← 0
for l ← 2 to n
    do for i ← 1 to n - l + 1
        do j ← i + l - 1
            m[i, j] ← ∞
            for k ← i to j - 1
                do q ← m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
                    if q < m[i, j]
                        then m[i, j] ← q
                             s[i, j] ← k
return m, s
```

Running time: $\Theta(n^3)$

Chains of length one have cost 0

l is the length of the chain

For a particular m[i, j], look at all possible choices for k and choose the one that gives the minimum cost

Construct the Optimal Solution

Store the optimal choice made at each subproblem

$s[i, j]$ = a value of k such that an optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1}

$s[1, n]$ is associated with the entire product $A_{1..n}$

- The final matrix multiplication will be split at k = $s[1, n]$

  $$A_{1..n} = A_{1..s[1, n]} \cdot A_{s[1, n]+1..n}$$

- For each subproduct recursively find the corresponding value of k that results in an optimal parenthesization

Construct the Optimal Solution

- $s[i, j]$ = value of k such that the optimal parenthesization of $A_i\ A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}

	1	2	3	4
1	0	1	1	3
2		0	2	3
3			0	3
4				0

- $s[1, n] = 3 \Rightarrow A_{1..4} = A_{1..3}\ A_{4..4}$
- $s[1, 3] = 1 \Rightarrow A_{1..3} = A_{1..1}\ A_{2..3}$

Final Parenthesis: $((\ A_1(A_2\ A_3)\ )A_4)$

Construct the Optimal Solution (cont.)

PRINT-OPT-PARENS(s, i, j)

if i = j

then print "A $_i$"

else print "("

PRINT-OPT-PARENS(s, i, $s[i, j]$)

PRINT-OPT-PARENS(s, $s[i, j] + 1$, j)

print ")"

	1	2	3	4
1	0	1	1	3
2		0	2	3
3			0	3
4				0

Initial Call is PRINT-OPT-PARENS(s, 1, 4)

Matrix-chain Multiplication: DP

```
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
```

Matrix-chain Multiplication: D-n-C

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
if i == j
    return 0
m[i, j] = ∞
for k = i to j − 1
    q = RECURSIVE-MATRIX-CHAIN(p, i, k)
        + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
        + p_{i−1} p_k p_j
    if q < m[i, j]
        m[i, j] = q
return m[i, j]
```

Matrix-chain Multiplication: Memoization

```
MEMOIZED-MATRIX-CHAIN(p)
n = p.length − 1
let m[1..n, 1..n] be a new table
for i = 1 to n
    for j = i to n
        m[i, j] = ∞
return LOOKUP-CHAIN(m, p, 1, n)

LOOKUP-CHAIN(m, p, i, j)
if m[i, j] < ∞                    //If solved earlier lookup from the table.
    return m[i, j]
if i == j
    m[i, j] = 0
else for k = i to j − 1
        q = LOOKUP-CHAIN(m, p, i, k)
            + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
        if q < m[i, j]
            m[i, j] = q
return m[i, j]
```

Thank You