DYNAMIC PROGRAMMING

LCS AND MCM

ELEMENTS OF DYNAMIC PROGRAMMING

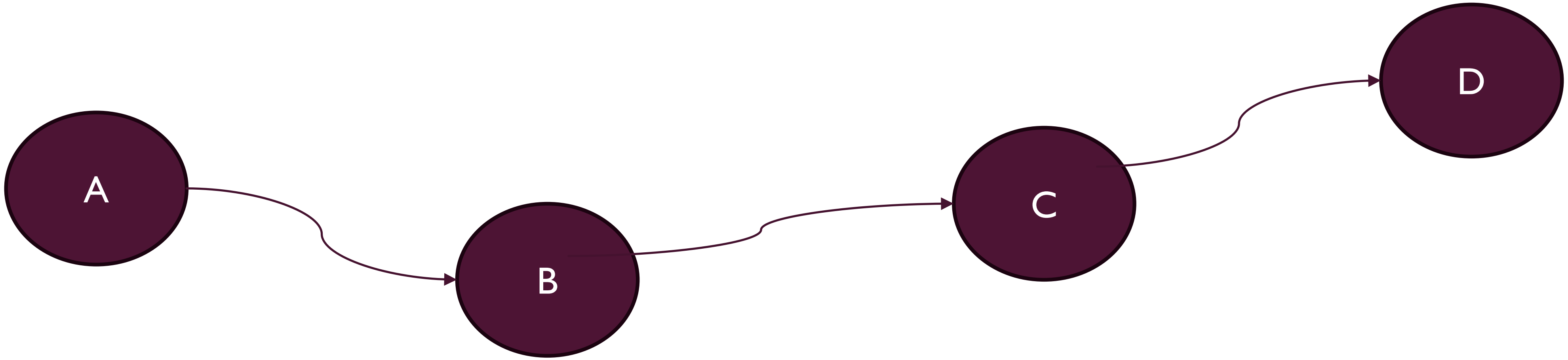
- Optimal substructure
- Overlapping subproblems

OPTIMAL SUBSTRUCTURE

A problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems.

OPTIMAL SUBSTRUCTURE

A problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems.



**If the shortest route from A to D passes through B and then C, then the shortest route from B to D must pass through C too. That is, the problem of how to get from B to D is nested inside the problem of how to get from A to D.

OPTIMAL SUBSTRUCTURE

- Show that a solution to a problem consists of making a choice, which leaves one or more subproblems to solve.

Imagine you're trying to find the shortest path from your home to your college. The entire journey can be seen as a problem. When you choose a particular road to start your journey, you're making a choice. This choice then leaves you with a subproblem: finding the shortest path from the end of this road to your college.

OPTIMAL SUBSTRUCTURE

- Show that a solution to a problem consists of making a choice, which leaves one or more subproblems to solve.
- Suppose that you are given this last choice that leads to an optimal solution.
- Given this choice, determine which subproblems arise and how to characterize the resulting space of subproblems.

Each subproblem is essentially a smaller instance of the original problem: finding the shortest path between two points.

OPTIMAL SUBSTRUCTURE

- Show that a solution to a problem consists of making a choice, which leaves one or more subproblems to solve.
- Suppose that you are given this last choice that leads to an optimal solution.
- Given this choice, determine which subproblems arise and how to characterize the resulting space of subproblems.
- Show that the solutions to the subproblems used within the optimal solution must themselves be optimal. Usually use cut-and-paste.
- Need to ensure that a wide enough range of choices and subproblems are considered.

OPTIMAL SUBSTRUCTURE

- Show that the solutions to the subproblems used within the optimal solution must themselves be optimal. Usually use cut-and-paste.

OPTIMAL SUBSTRUCTURE

- Show that a solution to a problem consists of making a choice, which leaves one or more subproblems to solve.
- Suppose that you are given this last choice that leads to an optimal solution.
- Given this choice, determine which subproblems arise and how to characterize the resulting space of subproblems.
- Show that the solutions to the subproblems used within the optimal solution must themselves be optimal. Usually use cut-and-paste.
- Need to ensure that a wide enough range of choices and subproblems are considered.

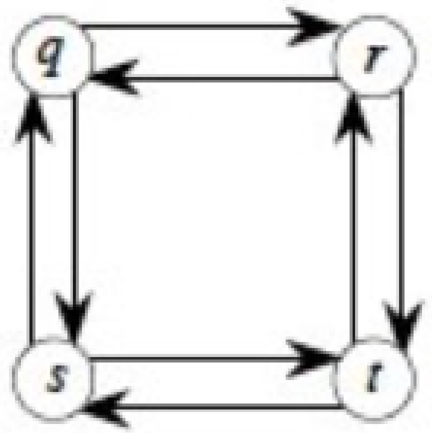
  This means not just looking at the immediate next step, but considering all possible paths that could lead from your starting point to your destination

OPTIMAL SUBSTRUCTURE

- Optimal substructure varies across problem domains:
  - 1. *How many subproblems* are used in an optimal solution.
  - 2. *How many choices* in determining which subproblem(s) to use.
- Informally, running time depends on (# of subproblems overall) × (# of choices).
- Dynamic programming uses optimal substructure **bottom up**.
  - *First* find optimal solutions to subproblems.
  - *Then* choose which to use in optimal solution to the problem.

OPTIMAL SUBSTUCTURE

- Does optimal substructure apply to all optimization problems? Yes/No.
- Applies to determining the shortest path
- Does it apply to determining the longest simple path of an unweighted directed graph.



$$q \rightarrow s \rightarrow t \rightarrow r$$

$$r \rightarrow q \rightarrow s \rightarrow t$$

OPTIMAL SUBSTUCTURE

- Does optimal substructure apply to all optimization problems? Yes/No.
- Applies to determining the shortest path but NOT the longest simple path of an unweighted directed graph.
- Why?
  - Shortest path has independent subproblems.
  - Solution to one subproblem does not affect solution to another subproblem of the same problem.
  - Subproblems are not independent in longest simple path.
    - Solution to one subproblem affects the solutions to other subproblems.

Dynamic programming requires *overlapping* yet *independently solveable* subproblems.

OVERLAPPING SUBPROBLEMS

- The space of subproblems must be "small".
- The total number of distinct subproblems is a polynomial in the input size.
  - A recursive algorithm is exponential because it solves the same problems repeatedly.
  - If divide-and-conquer is applicable, then each problem solved will be brand new.

OPTIMAL SUBSTRUCTURE PROPERTY

- If S is an optimal solution to a problem, then the components of S are optimal solutions to subproblems
- Examples:
  - True for knapsack
  - True for coin-changing
  - True for single-source shortest path
  - Not true for longest-simple-path

GENERAL STRATEGY OF DYN. PROG.

1. Structure: What's the structure of an optimal solution in terms of solutions to its subproblems?
2. Give a recursive definition of an optimal solution in terms of optimal solutions to smaller problems
   - Usually using min or max
3. Use a data structure (often a table) to store smaller solutions in a <u>bottom-up</u> fashion
   - Optimal value found in the table
4. (If needed) Reconstruct the optimal solution
   - I.e. what produced the optimal value

LONGEST COMMON SUBSEQUENCE (LCS)

Problem: Given 2 sequences, $X = \langle x_1,\ldots,x_m \rangle$ and $Y = \langle y_1,\ldots,y_n \rangle$, find a common subsequence whose length is maximum.

springtime	ncaa tournament	basketball
printing	north carolina	krzyzewski

Subsequence need not be consecutive, but must be in order.

STEPS IN DYNAMIC PROGRAMMING

1. Characterize structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution values either top-down with caching or bottom-up in a table.
4. Construct an optimal solution from computed values.

LONGEST COMMON SUBSEQUENCE (LCS)

Problem: Given 2 sequences, $X = \langle x_1,\ldots,x_m \rangle$ and $Y = \langle y_1,\ldots,y_n \rangle$, find a common subsequence whose length is maximum.

Subsequence need not be consecutive, but must be in order.

LONGEST COMMON SUBSEQUENCE (LCS)

Application: comparison of two DNA strings

Ex: X= {A B C B D A B }, Y= {B D C A B A}

Longest Common Subsequence:

X = A **B** **C** **B** D **A** B

Y = **B** D **C** A **B** **A**

Brute force algorithm would compare each subsequence of X with the symbols in Y

LCS ALGORITHM

- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.
- Define X_i, Y_j to be the prefixes of X and Y of length *i* and *j* respectively
- Define $c[i,j]$ to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be $c[m,n]$

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

LCS RECURSIVE SOLUTION

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- We start with $i = j = 0$ (empty substrings of x and y)
- Since X_0 and Y_0 are empty strings, their LCS is always empty (i.e. $c[0,0] = 0$)
- LCS of empty string and any other string is empty, so for every i and j: $c[0, j] = c[i,0] = 0$

LCS RECURSIVE SOLUTION

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

- When we calculate *c[i,j]*, we consider two cases:
- **First case:** *x[i]=y[j]*: one more symbol in strings X and Y matches, so the length of LCS X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{i-1} , plus 1

LCS RECURSIVE SOLUTION

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max(c[i,j-1],c[i-1,j]) & \text{otherwise} \end{cases}$$

- **Second case:** *x[i] != y[j]*
- As symbols don't match, our solution is not improved, and the length of LCS(X_i , Y_j) is the same as before (i.e. maximum of LCS(X_i, Y_{j-1}) and LCS(X_{i-1},Y_j)

LCS LENGTH ALGORITHM

```
LCS-Length(X, Y)
m = length(X)  // get the # of symbols in X
n  = length(Y) // get the # of symbols in Y
for i = 1 to m    c[i,0] = 0     // special case: Y_0
for j = 1 to n    c[0,j] = 0     // special case: X_0
for i = 1 to m                   // for all X_i
  for j = 1 to n                 // for all Y_j
      if ( X_i == Y_j )
          c[i,j] = c[i-1,j-1] + 1
      else c[i,j] = max( c[i-1,j], c[i,j-1] )
return c[m,n]   // return LCS length for X and Y
```

LCS EXAMPLE

We'll see how LCS algorithm works on the following example:

- X = ABCB
- Y = BDCAB

What is the Longest Common Subsequence of X and Y?

LCS(X, Y) = BCB

X = A **B**   **C**   **B**

Y =   **B** D **C** A **B**

ABCB

BDCAB

LCS EXAMPLE

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,4]

ABCB

BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

for i = 1 to m   c[i,0] = 0

for j = 1 to n   c[0,j] = 0

ABCB

BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	A	**0**	**0**				
2	B	**0**					
3	C	**0**					
4	B	**0**					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB

BDCAB

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**		
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB

BDCAB

		j					
		0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB

BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB

BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**				
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB

BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB

BDCAB

i	j	0	1	2	3	4	5
	Yj		B	D	C	A	B
	Xi						
0		0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0					
4	B	0					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB

BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**			
4	**B**	**0**					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB

BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**		
4	**B**	**0**					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB

BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**	**1**				

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB

BDCAB

		j	0	1	2	3	4	5
i			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1	1	2	2	

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB

BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	3

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS ALGORITHM RUNNING TIME

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?

O(m*n)

since each c[i,j] is calculated in constant time, and there are m*n elements in the array

HOW TO FIND ACTUAL LCS

- So far, we have just found the *length* of LCS, but not LCS itself.
- We want to modify this algorithm to make it output Longest Common Subsequence of X and Y

Each *c[i,j]* depends on *c[i-1,j]* and *c[i,j-1]*

or *c[i-1, j-1]*

For each c[i,j] we can say how it was acquired:

2	2
2	3

For example, here
c[i,j] = c[i-1,j-1] +1 = 2+1=3

HOW TO FIND ACTUAL LCS - CONTINUED

- Remember that

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- So we can start from *c[m,n]* and go backwards
- Look first to see if 2nd case above was true
- If not, then *c[i,j]* = *c[i-1, j-1]+1*, so remember *x[i]* (because *x[i]* is a part of LCS)
- When i=0 or j=0 (i.e. we reached the beginning), output remembered letters in reverse order

ALGORITHM TO FIND ACTUAL LCS

- Here's a recursive algorithm to do this:

```
LCS_print(x, m, n, c) {
   if (c[m][n] == c[m-1][n]) // go up?
      LCS_print(x, m-1, n, c);
   else if (c[m][n] == c[m][n-1] // go left?
      LCS_print(x, m, n-1, c);
   else { // it was a match!
      LCS_print(x, m-1, n-1, c);
      print(x[m]); // print after recursive call
    }
}
```

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

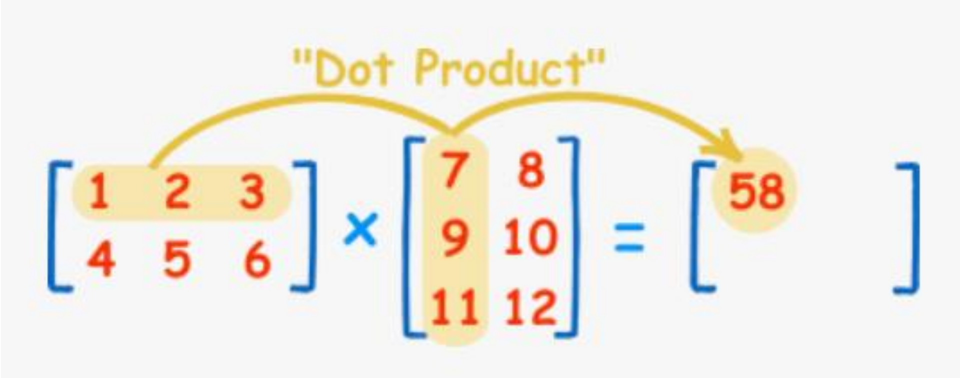
LCS (reversed order): **B C B**

LCS (straight order): **B C B**

(this string turned out to be a palindrome)

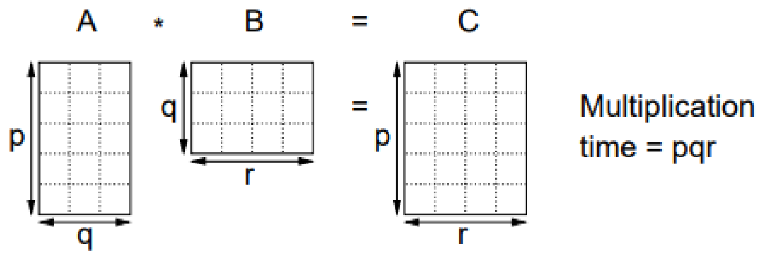
MATRIX-CHAIN MULTIPLICATION (MCM)

- Problem: given $\langle A_1, A_2, \ldots, A_n \rangle$, compute the product: $A_1 \times A_2 \times \ldots \times A_n$, find the fastest way (i.e., minimum number of multiplications) to compute it.
- Suppose two matrices $A(p,q)$ and $B(q,r)$, compute their product $C(p,r)$ in $p \times q \times r$ multiplications
  - **for** $a=1$ **to** p **for** $b=1$ **to** r $C[a,b]=0$
  - **for** $a=1$ **to** p
    - **for** $b=1$ **to** r
      - **for** $c=1$ **to** q $C[a,b] = C[a,b]+ A[a,c]B[c,b]$



MATRIX-CHAIN MULTIPLICATION

- Different parenthesizations will have different number of multiplications for product of multiple matrices
- Example: A(10,100), B(100,5), C(5,50)
  - If ((A ×B) ×C), 10 ×100 ×5 +10 ×5 ×50 =7500
  - If (A ×(B ×C)), 10 ×100 ×50+100 ×5 ×50=75000



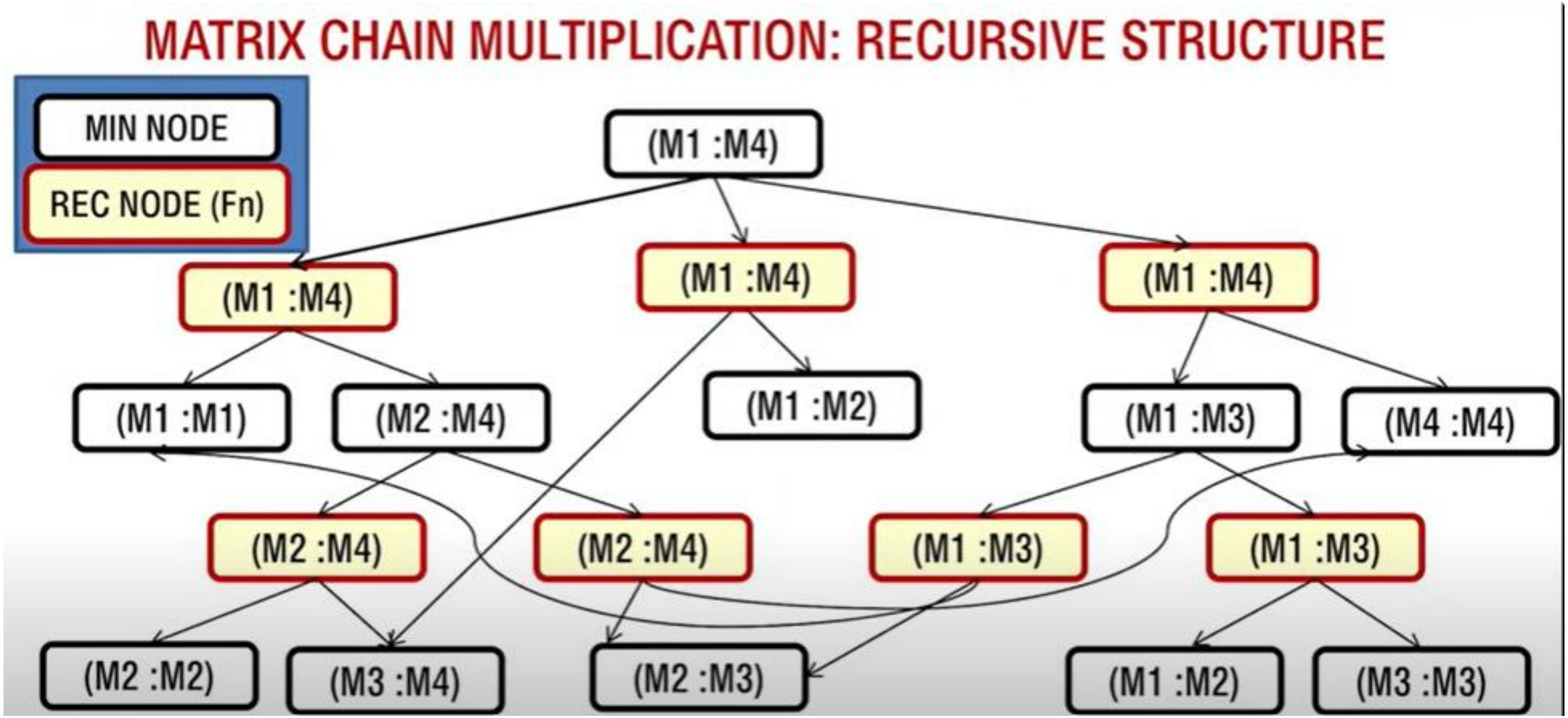
- The first way is ten times faster than the second !!!
- Denote <$A_1, A_2, \ldots, A_n$> by < $p_0, p_1, p_2, \ldots, p_n$>
  - i.e, $A_1(p_0,p_1), A_2(p_1,p_2), \ldots, A_i(p_{i-1},p_i), \ldots A_n(p_{n-1},p_n)$

MATRIX-CHAIN MULTIPLICATION

- Intuitive brute-force solution: Counting the number of parenthesizations by exhaustively checking all possible parenthesizations.
- Let P(*n*) denote the number of alternative parenthesizations of a sequence of *n* matrices:

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

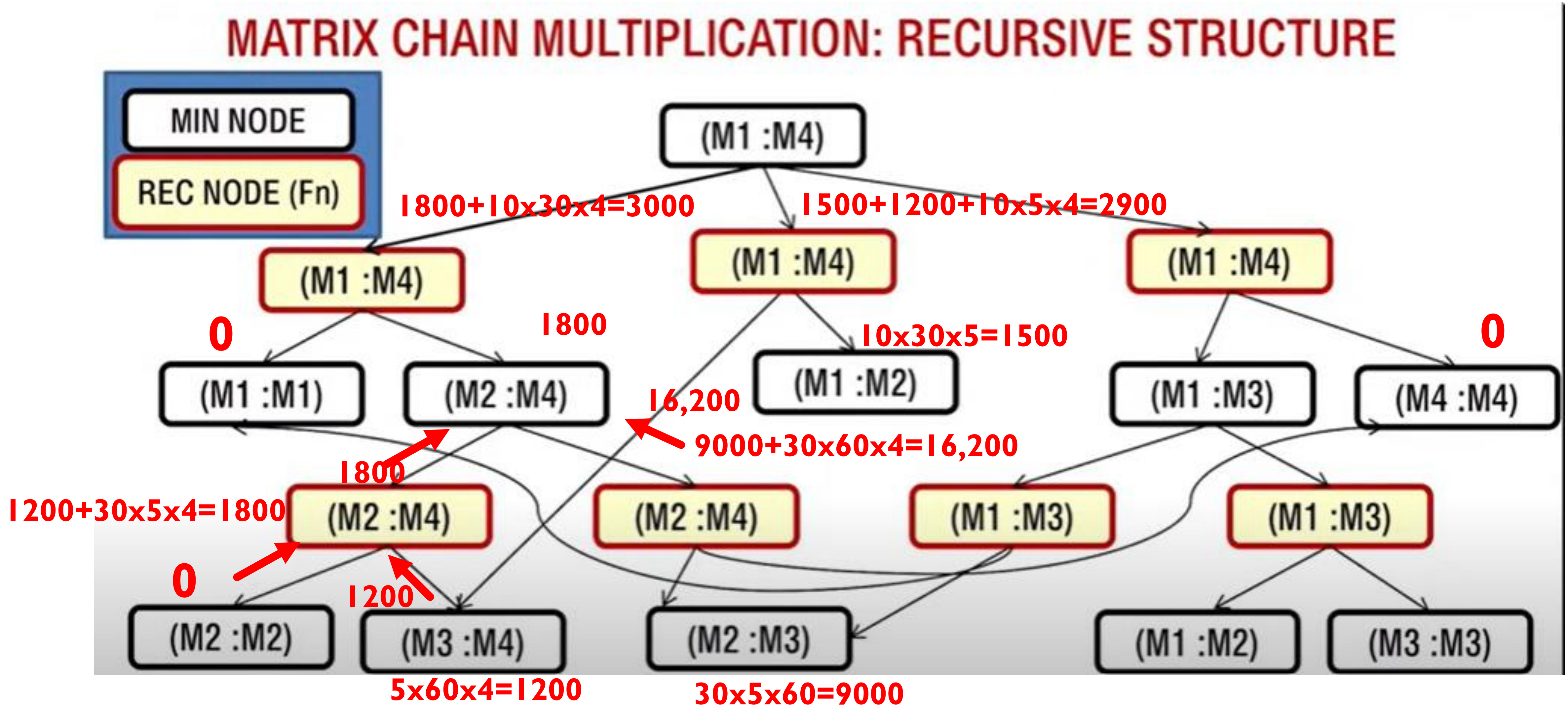
- The solution to the recursion is $\Omega(2^n)$.


MATRIX CHAIN MULTIPLICATION: RECURSIVE STRUCTURE
MIN NODE
REC NODE (Fn)
(M1 :M4)
(M1 :M4)
(M1 :M4)
(M1 :M4)
(M1 :M1)
(M2 :M4)
(M1 :M2)
(M1 :M3)
(M4 :M4)
(M2 :M4)
(M2 :M4)
(M1 :M3)
(M1 :M3)
(M2 :M2)
(M3 :M4)
(M2 :M3)
(M1 :M2)
(M3 :M3)

MCM: TOP DOWN EVALUATION

```
M[i, j], Done[i, j]=0

Eval-m(i, j)
{
        if (Done[i, j] =1) return (M[i, j])
        if (i=j) {
                Done[i, j]=1;
                M[i, j] = 0;
                return (M[i, j])
            }
        val=MAX_INT
        for(k=i to j-1){
                v_k= Eval-m(i, k)+Eval-m(k+1,j)+p[i-1]*p[k]*p[j]
                 if(v_k< val) val=v_k
         }
        Done[i,j]=1
        M[i, j]=val
        Return (M[i, j])
}
```



M1 [10 by 30], M2 [30 by 5], M3 [5 by 60], M4 [60 by 4]

MATRIX-CHAIN-ORDER(p)

```
n ← length[p] − 1
for i ← 1 to n
     do m[i, i] ← 0
for l ← 2 to n                    ▷ l is the chain length.
     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 and s
```