

# Dynamic Programming

Liu Ziwei

**Reference:** Computer Algorithms: Introduction to Design and Analysis, 3<sup>rd</sup> Ed, by Sara Basse and Allen Van Gelder. Sections 10.1, 10.2 & 10.3

# Outline

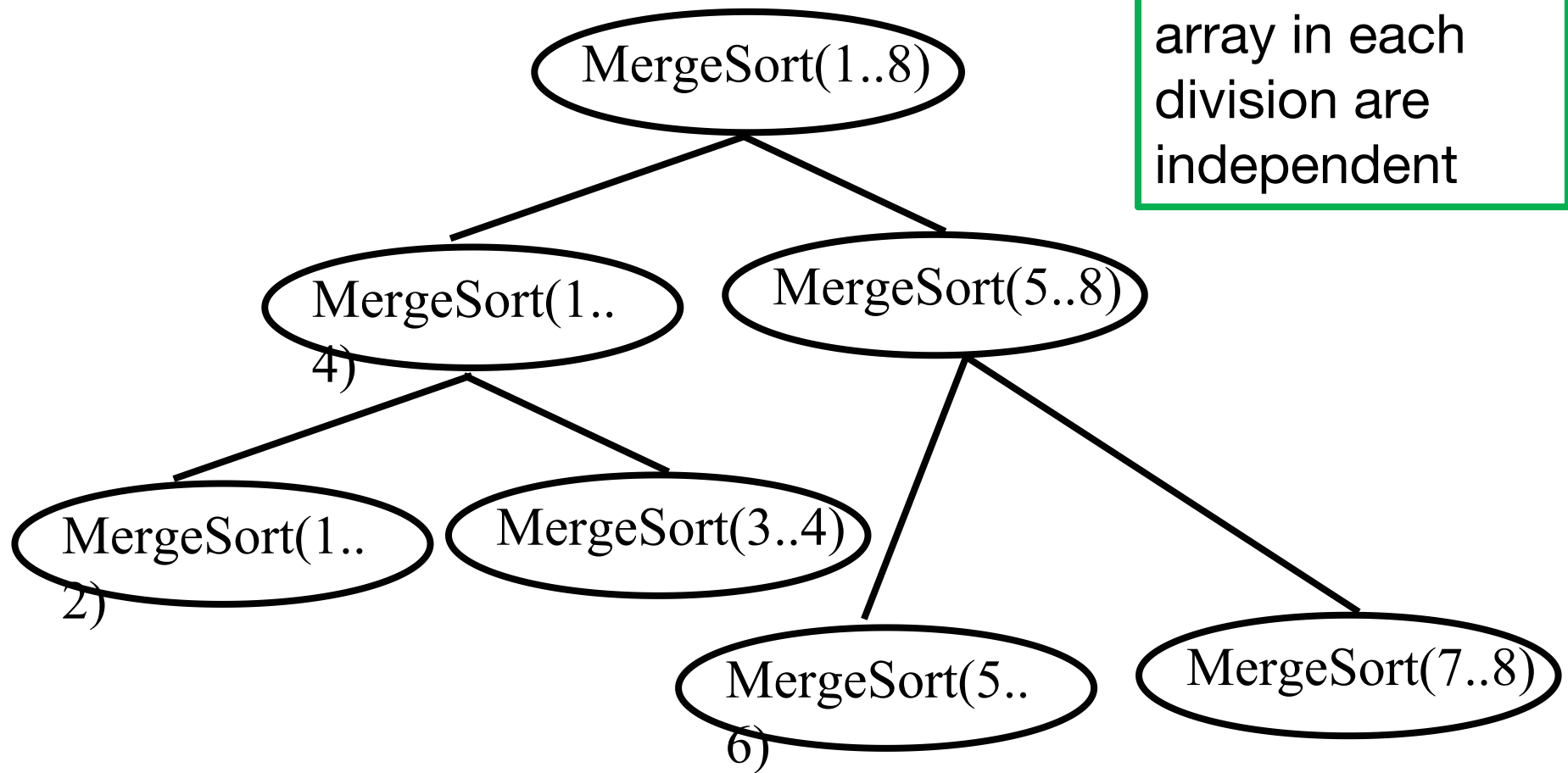
1. Concepts of dynamic programming
2. Longest common subsequence
3. Chain matrix multiplication
4. 0/1 Knapsack problem

# What is Dynamic Programming?

- It is a problem solving paradigm
- To a certain extent, it is similar to divide-and-conquer
- What do we do in divide-and-conquer?
  - Divide a problem into independent subproblems
  - Solve each subproblem recursively
  - Combine the solutions to subproblems into a solution for the given problem
  - Example: MergeSort

# Example: MergeSort(8)

The 2 half sections of the array in each division are independent



# What is Dynamic Programming?

- Dynamic programming:
  - Divide a problem into overlapping subproblems
  - Solve each subproblem recursively
  - Combine the solutions to subproblems into a solution for the given problem
  - Do not compute the answer to the same subproblem more than once
  - Example: computing Fibonacci numbers

## Fibonacci sequence

The Fibonacci sequence is defined recursively as:

$$F_n = F_{n-1} + F_{n-2} \quad \text{for } n \geq 2$$

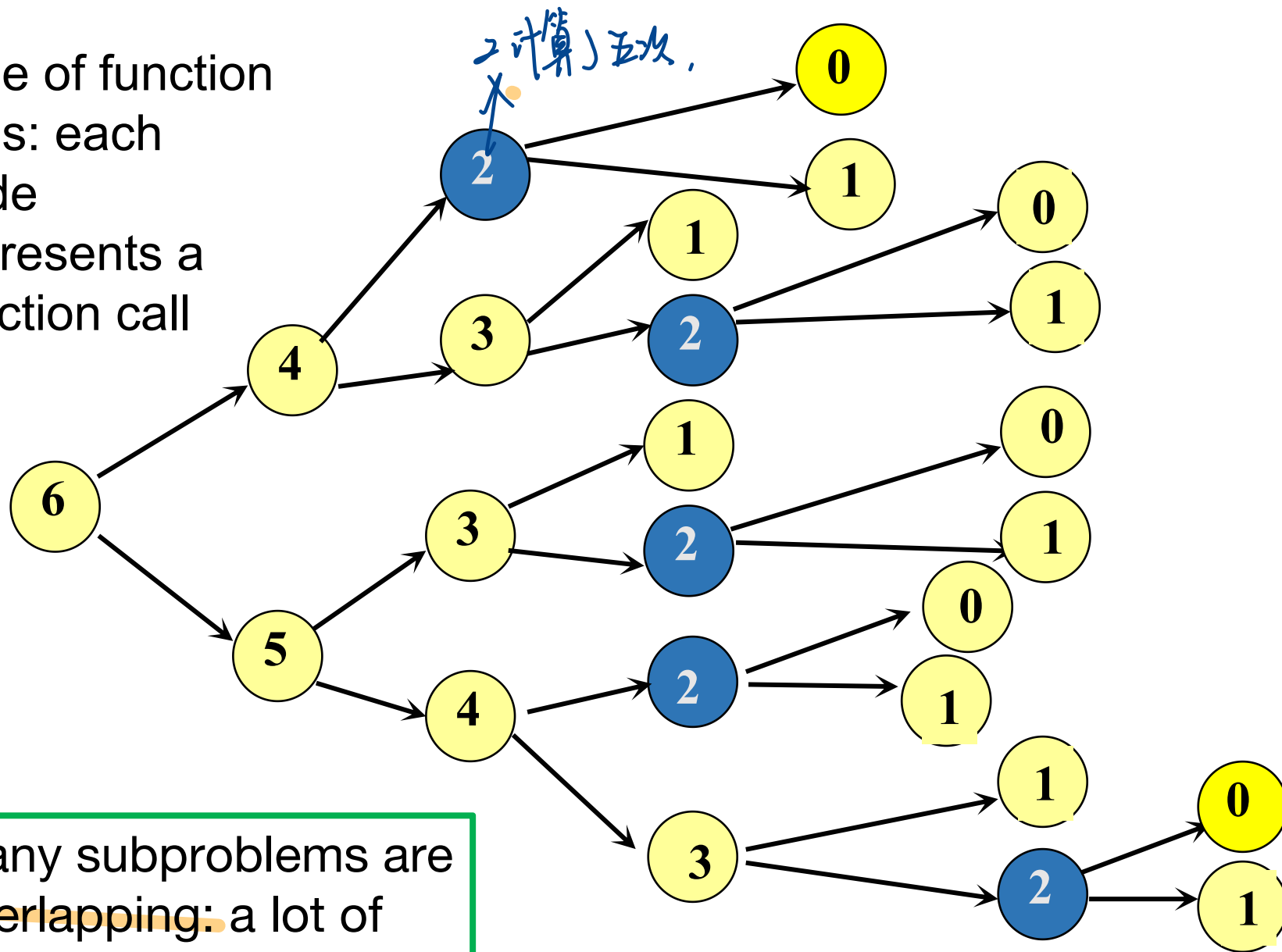
$$F_0 = 0, F_1 = 1$$

This series occurs frequently in algorithm analysis.

Divide-and-conquer: Recursive Fibonacci function

```
int fib(n)
{
    if (n == 0 || n == 1) return n;
    else return fib(n - 1) + fib(n - 2);
}
```

Tree of function calls: each node represents a function call



Many subproblems are overlapping: a lot of recomputation

- Example of repetition is given in the shaded nodes
- Notice that this is a full binary tree up to depth 3 (i.e.  $n/2$ )
- The deepest level is 5 (i.e.  $n-1$ )
- The number of recursive calls  $R$  is such that

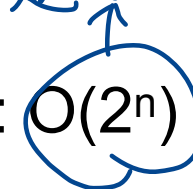
$$2^4 - 1 < R < 2^6 - 1$$

In general,

$$2^{\frac{n}{2}+1} - 1 < R < 2^n - 1$$

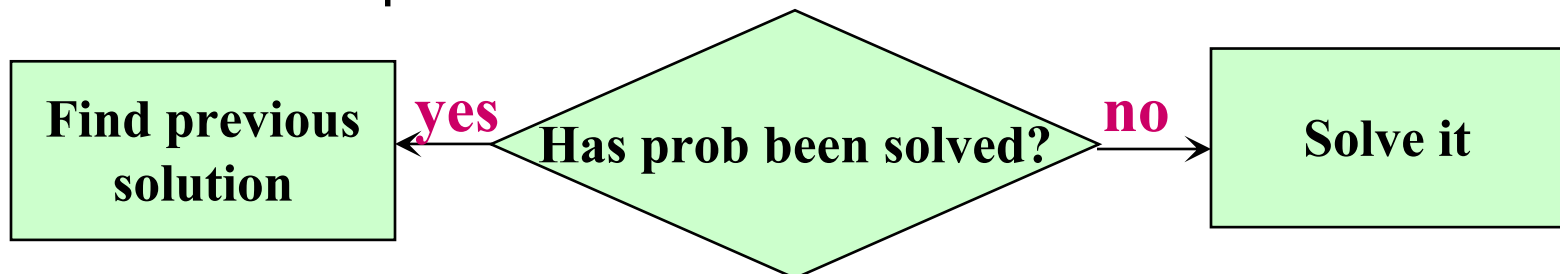
So this is an exponential time algorithm:  $O(2^n)$

超级大.  
↑





- The main feature of **dynamic programming** is that it replaces an exponential-time computation by a **polynomial-time computation**
- It is done by this: **Memorize the solutions and do not recompute**
- Recall that a DFS on a graph only explores edges to undiscovered vertices, and it checks the other edges.
- This strategy may be applied to only solve unsolved subproblems, and checks and retrieves solutions to the solved subproblems



# Dynamic programming (Top Down)

1. Formulate the problem  $P$  in terms of smaller versions of the problem (recursively), say,  $Q_1, Q_2, \dots$
2. Turn this formulation into a recursive function to solve problem  $P$
3. *Avoid redundancy*  
~~Use a dictionary to store solutions to subproblems~~
4. In the recursive function to solve  $P$ 
  - ❖ Before any recursive call, say on subproblem  $Q_i$ , check the dictionary to see if a solution for  $Q_i$  has been stored
    - If no solution has been stored, make the recursive call
    - Otherwise, retrieve the stored solution
  - ❖ Just before returning the solution for  $P$  store the

# The top-down approach

A dynamic programming version of fib(n)

```
int fibDP(n)
{
    int f1, f2;
    if (n == 0 || n == 1) {
        store(Soln, n, n);
        return n;
    }
    else {
        if (not member(Soln, n - 1))
            f1 = fibDP(n - 1);
        else f1 = retrieve(Soln, n - 1);
    }
}
```

*template*

*$f(0)=0, f(1)=1$ .*

*recursive call.*

**Store,**  
**member,**  
**retrieve** are all  
methods of the  
Dictionary

```

if (not member(Soln, n - 2))
    f2 = fibDP(n - 2);
else f2 = retrieve(Soln, n - 2);

f1 += f2;
store(Soln, n, f1);
return f1; }

```

*return the result.*

- Before calling fibDP, the dictionary Soln has to be initialized. E.g.

	0	1	2	3	4	5	6
Soln	-1	-1	-1	-1	-1	-1	-1
n							

- member(Dictionary, j):  
    return Dictionary[j] <> -1
- store(Dictionary, j, s):  
    Dictionary[j] = s
- retrieve(Dictionary, j):  
    return Dictionary[j]

```

int fibDP(n)
{ int f1, f2;
  if (n == 0 || n == 1) {
    store(Soln, n, n);
    return n;  }
  else {
    if (not member(Soln, n -
1))
      f1 = fibDP(n - 1);

```

```

    if (not member(Soln, n -
2))
      f2 = fibDP(n - 2);
    else f2 = retrieve(Soln, n
- 2);

    f1 += f2;
    store(Soln, n, f1);
    return f1;  }
}
Complexity: O(n)

```

else f1 = retrieve(Soln, n

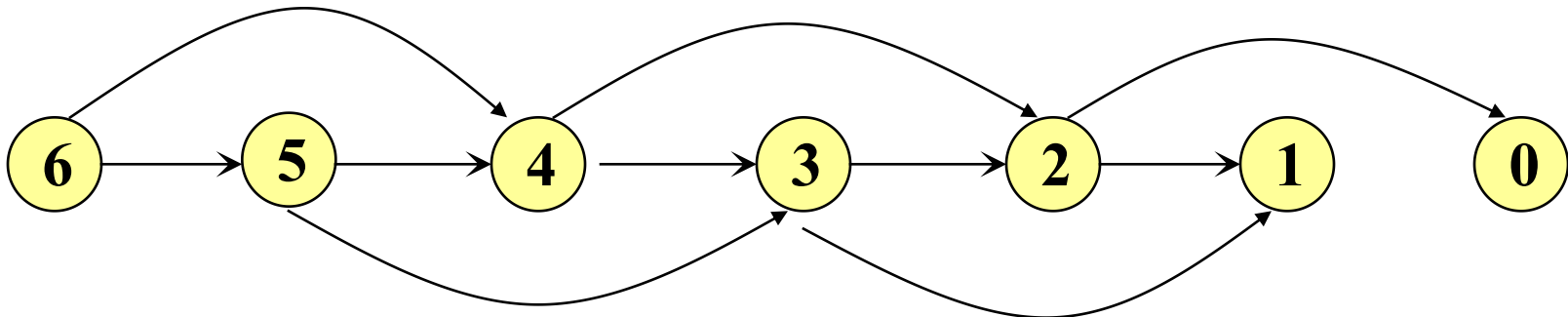
The total computational time in each function call, excluding that of the calls to fibDP() on subproblems is bounded by a constant.

The total computational cost is thus proportional to the number of calls to fibDP() when solving fibDP(n) – n+1 times.

# Dynamic programming (Bottom Up)

## Subproblem graphs

- For a recursive algorithm A, the subproblem graph for A is a directed graph whose vertices are the instances for this problem. The directed edges (I, J) for all pairs that indicate: if A is invoked on problem I, it makes a recursive call directly on instance J.
- E.g. the subproblem graph for fib(6):



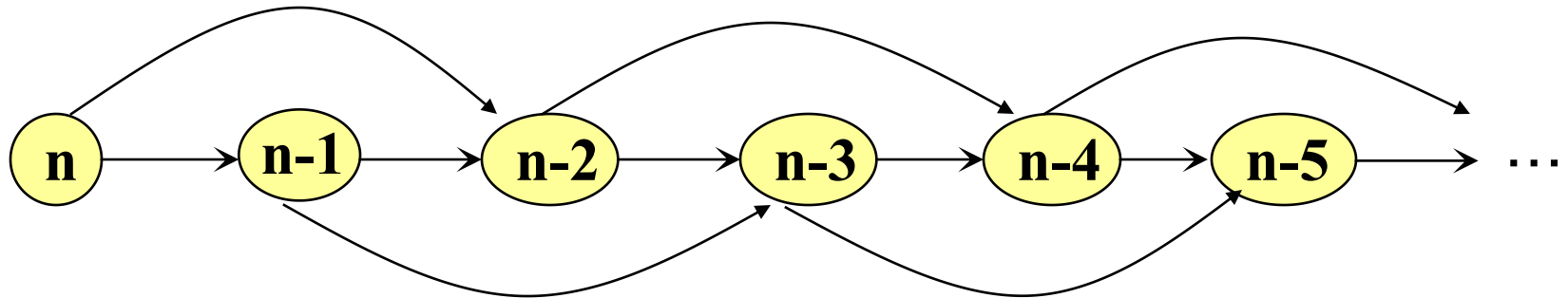
# Dynamic programming (Bottom Up)

先拆分, 再自下而上解决.

1. Formulate the problem  $P$  in terms of smaller versions of the problem (recursively), say,  $Q_1, Q_2, \dots$
2. Turn this formulation into a recursive function to solve problem  $P$
3. Draw the subproblem graph and find the dependencies among subproblems
4. Use a dictionary to store solutions to subproblems
5. In the iterative function to solve  $P$ 
  - ❖ compute the solutions of subproblems of a problem first
  - ❖ The solution to  $P$  is computed based on the solutions to its subproblems and is stored into the dictionary



The subproblem graph of  $\text{fib}(n)$



- Observation 1: Since we have a sequence of subproblems for  $\text{fib}(n)$ , we will use an one-dimensional array to memorize the solutions of subproblems. E.g.  $\text{fib}(6)$

	0	1	2	3	4	5	6
Soln							

- Observation 2: Seeing the dependencies among the solutions –  $\text{fib}(n)$  needs the solutions of  $\text{fib}(n-1)$  and  $\text{fib}(n-2)$ , we can compute the elements in this array in a correct order.

```
soln[0] = 0;  
soln[1] = 1;  
for j = 2 to n  
    soln[j] = soln[j-1] + soln[j-2];
```

Complexity:  $O(n)$

	0	1	2	3	4	5	6
Soln	0	1	1	2	3	5	8

# Longest Common Subsequence

- Given a sequence  $s = \langle s_1, s_2, \dots, s_n \rangle$ , a subsequence is any sequence  $\langle s_{i_1}, s_{i_2}, \dots, s_{i_m} \rangle$ , with  $i_j$  strictly increasing.

Example:  $s = \text{ACTTGCG}$

$\text{ACT}, \text{AG}, \text{ATTC}, \text{T}, \text{ACTTGC}$  are all subsequences.

$\text{TTA}, \text{AGGC}$  are not subsequences.

- Given two sequences  $x = \langle x_1, x_2, \dots, x_n \rangle$ ,  $y = \langle y_1, y_2, \dots, y_m \rangle$ , a common subsequence is a subsequence of both  $x$  and  $y$ .

- A longest common subsequence (LCS) is a common subsequence of maximum length

Example:  $x =$

AAACCGTGAGTTATTCTAGAA

$y =$  CACCCCTAAGGTACCTTTGGTTC

Common subsequences: ACGG, CAGTTTC

LCS = ACCTAGTACTTTG (LCS may not be unique)

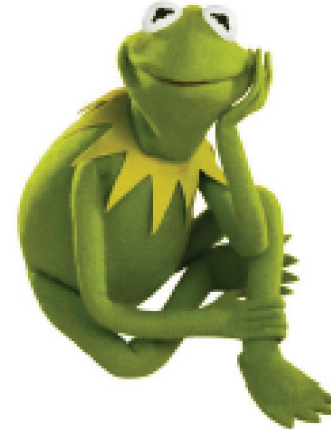
- LCS has many applications including document analysis and computational biology – the similarity between two sequences is measured by the length of LCS

- How similar are these two species?



DNA:

AGCCCTAAGGGCTACCTAGCTT



DNA:

GACAGCCTACAAGCGTTAGCTTG

Problem definition: Given two sequences  $x = \langle x_1, x_2, \dots, x_n \rangle$ ,  $y = \langle y_1, y_2, \dots, y_m \rangle$ , compute  $\text{LCS}(n, m)$  that gives the length of the longest common subsequence

- A trivial algorithm: find all subsequences of  $x$  (there are up to  $2^n$  of them) and check whether they are subsequences of  $y$ .
- What is the complexity of this trivial algorithm?
- This is an optimization problem  $2^n$
- Dynamic programming is a powerful tool to solve optimization problems that satisfy the Principle of Optimality
- A problem is said to satisfy the principle of optimality if the subsolutions of an optimal solution of the problem are themselves optimal solutions for their subproblems.

- Does the LCS problem satisfy this principle?

Step 1: Formulate the problem P in terms of smaller versions of the problem

- Consider two sequences  $x = \langle x_1, x_2, \dots, x_i \rangle$ ,  $y = \langle y_1, y_2, \dots, y_j \rangle$ . Take them as character strings.

E.g.

		2	3	4	5
x	A	C	G	A	G
y		A	C	T	G

LCS(5,4)



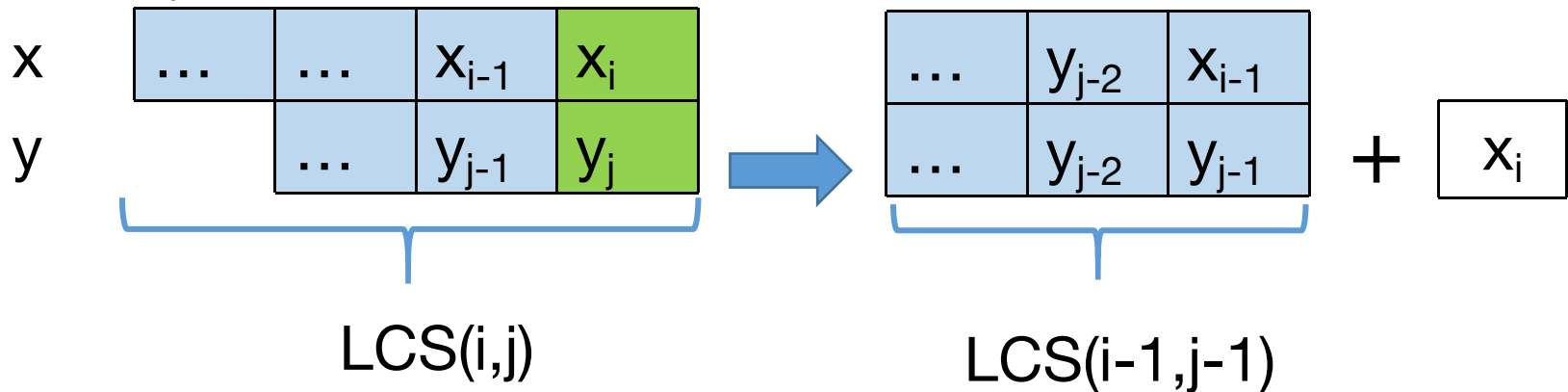
when  $x_5 = y_4$

	1	2	3	4
x	A	C	G	A
y		A	C	T

LCS(4,3)

+ G

- If  $x_i = y_j$ , then this character is the last character in the longest common subsequence. The longest common subsequence is the longest common subsequence of  $\langle x_1, x_2, \dots, x_{i-1} \rangle$ , and  $\langle y_1, y_2, \dots, y_{j-1} \rangle$  followed by  $x_i$ .



$LCS(i-1, j-1)$  is a subsolution of  $LCS(i, j)$  when  $x_i = y_j$ .

$LCS(i-1, j-1)$  is an optimal solution.

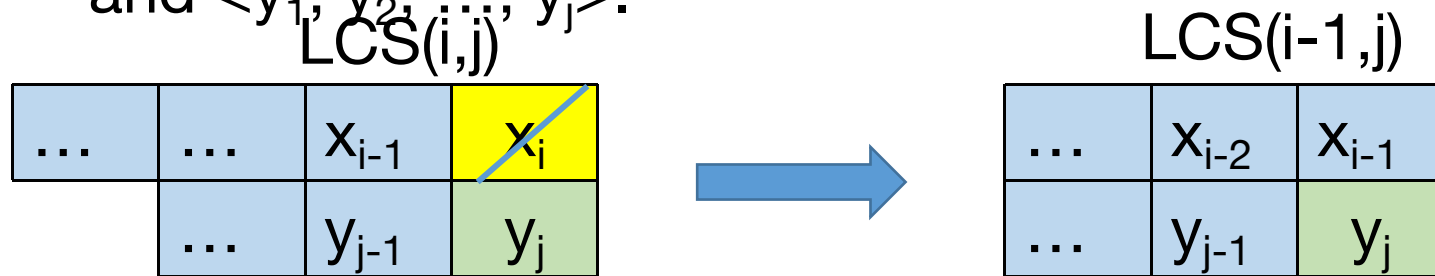
Otherwise  $LCS(i, j)$  cannot be an optimal solution



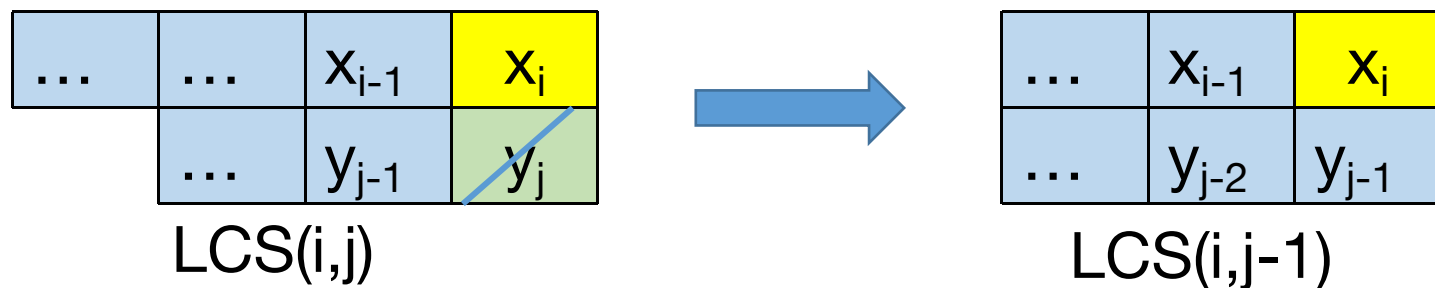
*Next question: How to determine whether in*

- If  $x_i \neq y_j$ , then either  $x_i$  is not in the LCS or  $y_j$  is not in the LCS (or both of them are not in the LCS). *LCS or not?*

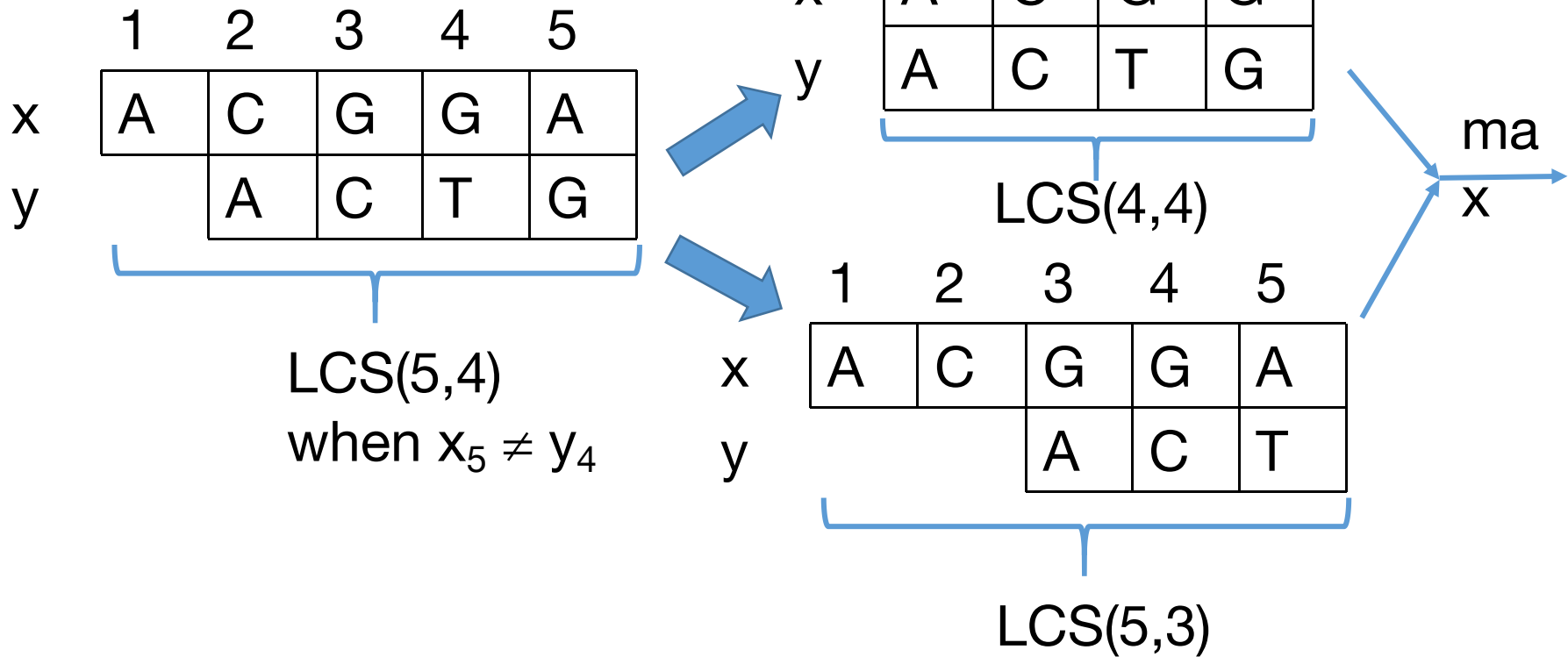
If  $x_i$  is not in the LCS, we just need to find the longest common subsequence of  $\langle x_1, x_2, \dots, x_{i-1} \rangle$  and  $\langle y_1, y_2, \dots, y_j \rangle$ .



If  $y_j$  is not in the LCS, we just need to find the longest common subsequence of  $\langle x_1, x_2, \dots, x_i \rangle$  and  $\langle y_1, y_2, \dots, y_{j-1} \rangle$ .

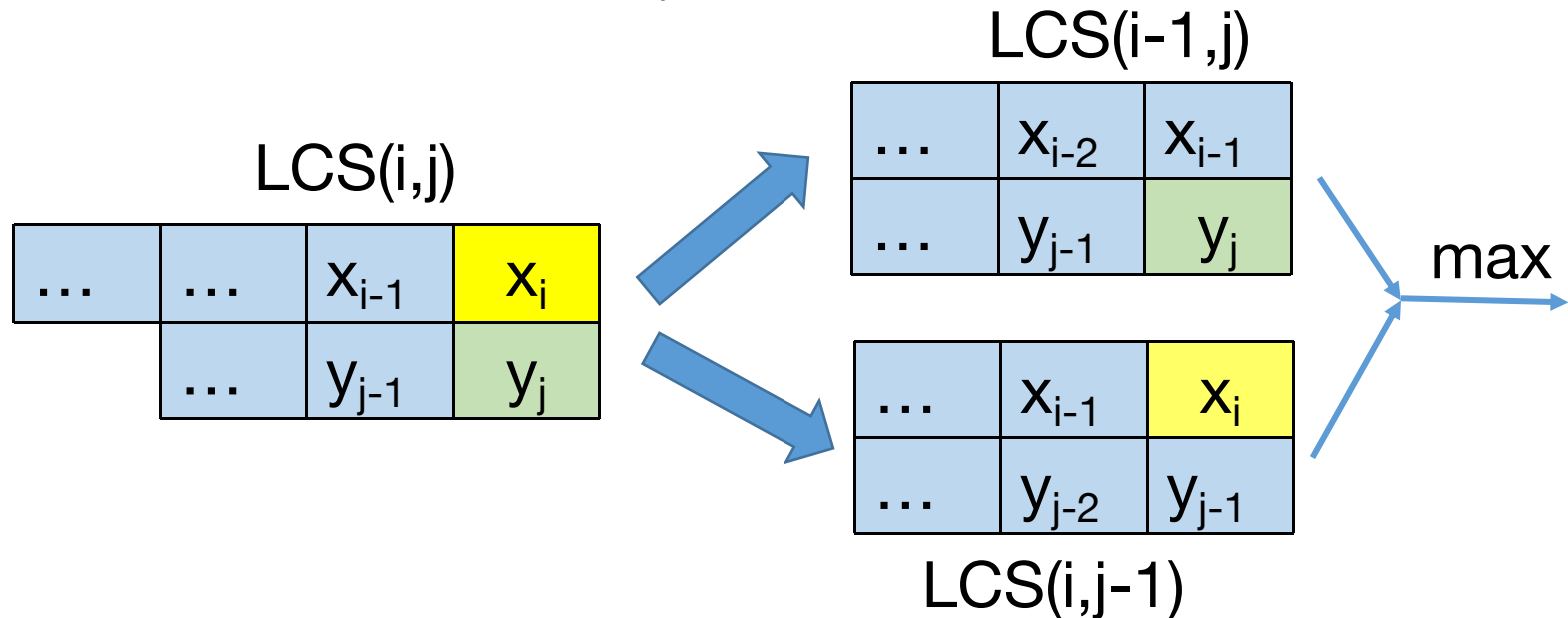


○ E.g.,



- The dynamic programming selection rule: **when given a number of possibilities, compute all and take the best.**

Therefore, when  $x_i \neq y_j$ ,



- $x_i = y_i$ 
 $x_{i-1} = y_j$ 
 $x_i = y_{j-1}$ 
 LCS(i-1,j-1), LCS(i-1,j) and LCS(i,j-1) are the optimal solutions for the respective subproblems. Otherwise LCS(i,j) cannot be optimal – principle of optimality.

Step 2: Turn this formulation into a recursive function to solve the longest common subsequence problem:

$$\text{LCS}(i,j) = 0 \quad \text{if } i=0$$

or  $j=0$

$$\text{LCS}(i,j) = \text{LCS}(i-1,j-1) + 1 \quad \text{if } i,j > 0,$$

$x_i = y_j$

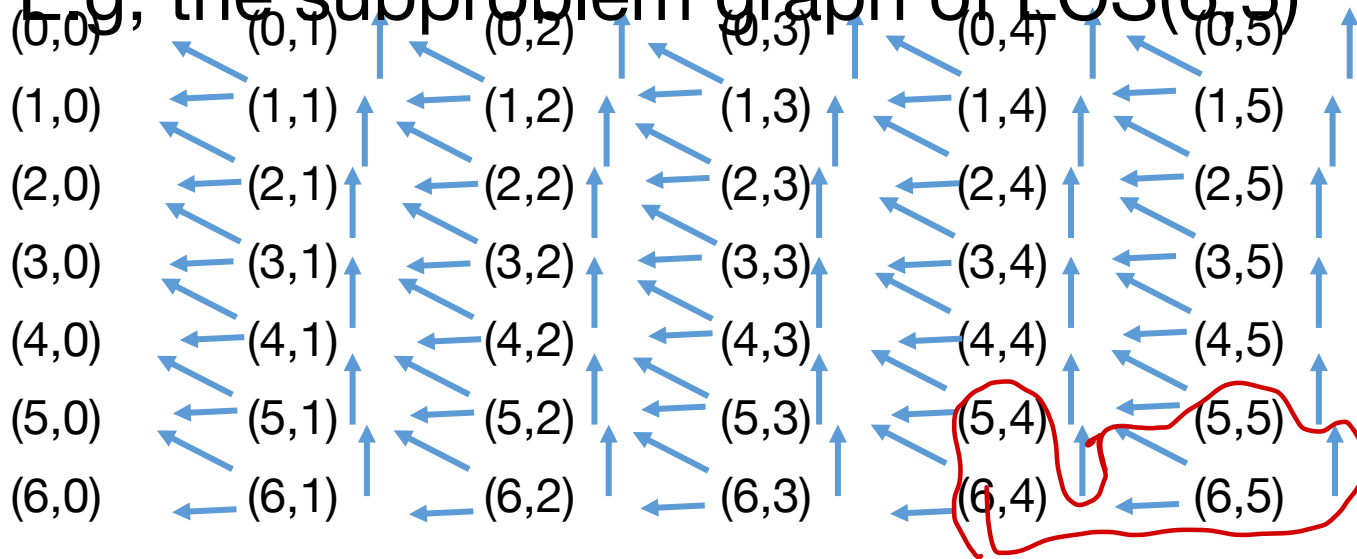
$$\text{LCS}(i,j) = \max(\text{LCS}(i-1,j), \text{LCS}(i,j-1)) \quad \text{if } i,j > 0,$$

$x_i \neq y_j$

- The top down approach *repeated solving sub problems* using a recursive function will be very inefficient.

Step 3 (bottom up approach): Draw the subproblem graph and find the dependencies among subproblems

E.g, the subproblem graph of LCS(6,5)



Step 4: the dictionary is a  $n+1$  by  $m+1$  array.

Initialise row 0, column 0.

Compute from row 1 to row  $n$ , column 1 to column  $m$  within each row.

## Step 5

```
Int LCS(n, m)
{
    for i = 0 to n    c[i][0] = 0;
    for j = 1 to m    c[0][j] = 0;
    for i = 1 to n
        for j = 1 to m
            if x[i] == y[j]
                c[i][j] = c[i-1][j-1] + 1;
            else if c[i-1][j] >= c[i][j-1]
                c[i][j] = c[i-1][j];
            else c[i][j] = c[i][j-1];
    return c[n][m];
}
```

```
Int LCS(n, m)
```

```
{
```

```
    for i = 0 to n    c[i][0] = 0;
```

```
    for j = 1 to m    c[0][j] = 0;
```

```
    for i = 1 to n
```

```
        for j = 1 to m
```

```
            if x[i] == y[j]
```

```
                c[i][j] = c[i-1][j-1] + 1;
```

```
            else if c[i-1][j] >= c[i][j-1]
```

```
                c[i][j] = c[i-1][j];
```

```
            else c[i][j] = c[i][j-1];
```

```
    return c[n][m];
```

```
}
```

Space Complexity:  
(n+1)x(m+1) array  
 $O(nm)$

Total no. of  
iterations:  $nm$

Bounded by a  
constant time

Time Complexity:  
 $O(nm)$

# Example 1

for i = 0 to n c[i][0] = 0;  
for j = 1 to m c[0][j] = 0;

	1	2	3	4	5
x	A	C	G	G	A
y	A	C	T	G	

		j	0	1	2	3	4
i			A	C	T	G	
0	p	0	0	0	0	0	
1	A	0	1	1	1	1	
2	C	0	1	2	2	2	
3	G	0	1	2	2	3	
4	G	0	1	2	2	3	
5	A	0	1	2	2	3	

Handwritten annotations on the DP table:

- Red arrows indicating the path of maximum matches from (5,5) to (0,0).
- Red circle around the value 2 at (3,3).
- Red text annotations:  $c_i = c$ ,  $a_i = G$ ,  $a_i = G$ ,  $a_i = G$ .
- Red text annotation:  $[1,1]$  near the cell (1,1).



# Example 1

	1	2	3	4	5
x	A	C	G	G	A
y	A	C	T	G	

		A	C	T	G
A	0	0	0	0	0
C	0	1	1	1	1
G	0	1	2	2	2
G	0	1	2	2	3
A	0	1	2	2	<b>3</b>

$$\text{LCM}(5,4) = 3$$

```

for i = 1 to n
  for j = 1 to m
    if x[i] == y[j]
      c[i][j] = c[i-1][j-1] + 1;
    else if c[i-1][j] >= c[i][j-1]
      c[i][j] = c[i-1][j];
    else c[i][j] = c[i][j-1];
  ]

```

- To find the longest common subsequence, a hint array is used to indicate for  $LCS(i,j)$  where the optimal subsolution is from :  $LCS(i-1, j-1)$ ,  $LCS(i-1, j)$  or  $LCS(i, j-1)$ .
  - First column of the hint array will be filled with '|'.
    - First row of the hint array will be filled '—'.
    - For the remaining cells  $h[i][j]$ ,
      - If  $LCS(i,j) = LCS(i-1, j-1) + 1$ ,  $h[i][j] = \backslash$  加一个新字符
      - If  $LCS(i,j) = LCS(i-1, j)$ ,  $h[i][j] = |$
      - If  $LCS(i,j) = LCS(i, j-1)$ ,  $h[i][j] = —$

Int LCS(n, m)     // with hints to find the sequence

{

  for i = 0 to n { c[i][0] = 0; h[i][0] = '|'; }

  for j = 1 to m { c[0][j] = 0; h[0][j] = '—'; }

  for i = 1 to n

    for j = 1 to m

      if x[i] == y[j]

        { c[i][j] = c[i-1][j-1] + 1; h[i][j] = '\'; }

      else if c[i-1][j] >= c[i][j-1]

        { c[i][j] = c[i-1][j]; h[i][j] = '|'; }

      else { c[i][j] = c[i][j-1]; h[i][j] = '—'; }

  return c[n][m];

}

Time Complexity:  
O(nm)

- To obtain the longest common subsequence computed, we start from  $h[n][m]$ .
- For each element of the hint array,  $h[i][j]$ ,
  - If  $h[i][j] = '\backslash'$ , it means  $x_i = y_j$  and this character is the last character of the longest common subsequence of  $x_1..x_i$  and  $y_1..y_j$ . This character is preceded by the longest common subsequence of  $x_1..x_{i-1}$  and  $y_1..y_{j-1}$ .
  - If  $h[i][j] = '|'$ , it means the longest common subsequence of  $x_1..x_i$  and  $y_1..y_j$  is the longest common subsequence of  $x_1..x_{i-1}$  and  $y_1..y_j$ .
  - If  $h[i][j] = '-'$ , it means the longest common subsequence of  $x_1..x_i$  and  $y_1..y_j$  is the longest common subsequence of  $x_1..x_i$  and  $y_1..y_{j-1}$ .
- After reaching the 1<sup>st</sup> row/column of the hint array,  
end

```
getSequence(n m)    // get the LCS from hint  
array
```

```
{ s = empty stack;    // s stores the characters  
in LCS
```

```
    i = n;
```

```
    j = m;
```

```
    while (i  $\neq$  0 and j  $\neq$  0)
```

```
        if (h[i][j] == '\')
```

```
            { s.push(x[i]); i--; j--; }
```

```
        else if (h[i][j] == '|')
```

```
            i--;
```

```
        else j--;
```

```
    pop and output from s;
```

Maximum no. of  
iterations:  $n+m$ .  
Complexity:  $O(n+m)$

Bounded by a  
constant time

# Example 1

	A	C	T	G
A	0	0	0	0
C	0	1	1	1
G	0	1	2	2
G	0	1	2	3
A	0	1	2	3

	A	C	T	G
A				
C				
G				
G				
A				

	1	2	3	4	5
x	A	C	G	G	A
y	A	C	T	G	

$h(5,4) = '|'$

$h(4,4) = '\backslash' \rightarrow$  G

$h(3,3) = '|'$

$h(2,3) = '-'$

$h(2,2) = '\backslash' \rightarrow$  C

$h(1,1) = '\backslash' \rightarrow$  A

end

The sub sequence: A C G

Example  
2:

x	C	G	G	T	A	T
y	A	G	T	T	G	C

refer to the upper first  
second if:  $[C[i-1][j]] \neq C[i][j]$ ...

	A	G	T	T	G	C
C	0	0	0	0	0	0
G	0	0	0	0	0	1
G	0	0	1	1	2	2
T	0	0	1	2	2	2
A	0	1	1	2	2	2
T	0	1	1	2	3	3

	A	G	T	T	G	C
C						
G						
G						
T						
A						
T						

$$\text{LCS}(6,6) = 3$$

Example  
2:

x	C	G	G	T	A	T
y	A	G	T	T	G	C

	A	G	T	T	G	C
C	0	0	0	0	0	0
G	0	0	1	1	1	1
G	0	0	1	1	2	2
T	0	0	1	2	2	2
A	0	1	1	2	2	2
T	0	1	1	2	3	3

$$\text{LCS}(6,6) = 3$$

*necessary to have index 0.*

	A	G	T	T	G	C
	—	—	—	—	—	—
C						\
G			\	—	\	
G			\		\	—
T				\		
A		\				
T				\	\	—

*push x[3]*  
*push x[4]*  
*push x[6]*



Example  
2:

x	C	G	G	T	A	T
y	A	G	T	T	G	C

		A	G	T	T	G	C
	—	—	—	—	—	—	—
C							\
G			\	—	—	\	
G			\			\	—
T				\	\		
A		\					
T				\	\	—	—

$LCS(6,6) = 3$

$h(6,6) = \text{—}$

$h(6,5) = \text{—}$

$h(6,4) = \backslash \rightarrow \boxed{T}$

$h(5,3) = |$

$h(4,3) = \backslash \rightarrow \boxed{T}$

$h(3,2) = \backslash \rightarrow \boxed{G}$

$h(2,1) = |$

$h(1,1) = |$

$h(0,1) = \text{—}$

end

The subsequence:  $\boxed{G} \boxed{T} \boxed{T}$

# Chain Matrix Multiplication

- The Order problem

Consider  $A_1 \times A_2 \times A_3 \times A_4$   
 $30 \times 1 \quad 1 \times 40 \quad 40 \times 10 \quad 10 \times 25$

$$\begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

$2 \times 3 \times 4$

Many possibilities. For examples,

$$((A_1 A_2) A_3) A_4 \longrightarrow 30 \times 1 \times 40 + 30 \times 40 \times 10 + 30 \times 10 \times 25 = 20,700 \text{ multiplications}$$

$$A_1 (A_2 (A_3 A_4)) \longrightarrow 40 \times 10 \times 25 + 1 \times 40 \times 25 + 30 \times 1 \times 25 = 11,750 \text{ multiplications}$$

$$(A_1 A_2) (A_3 A_4) \longrightarrow 30 \times 1 \times 40 + 40 \times 10 \times 25 + 30 \times 40 \times 25 = 41,200 \text{ multiplications}$$

$$A_1 ((A_2 A_3) A_4) \longrightarrow 1 \times 40 \times 10 + 1 \times 10 \times 25 + 30 \times 1 \times 25 = 1,400 \text{ multiplications}$$

Problem definition: given matrices  $A_1, A_2, \dots, A_n$  where dimensions of  $A_i$  are  $d_{i-1} \times d_i$  (for  $1 \leq i \leq n$ ), what order should the matrix multiplications be computed in order to incur minimum cost? Cost is the number of multiplications.

$d_0 \quad d_1 \quad d_2 \quad d_3 \quad \dots \quad d_{n-1} \quad d_n$

- There are  $(n-1)!$  ways for  $n$  matrices
- Matrix multiplication is associative:  $(AB)C = A(BC)$ . So different ways give the same result
- This is an optimization problem

Step 1: formulate the matrix multiplication cost problem in terms of smaller versions of the same problem

Consider a sequence of 6 matrices:

$A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6$   
matrices

$d_0 \times d_1 \quad d_1 \times d_2 \quad d_2 \times d_3 \quad d_3 \times d_4 \quad d_4 \times d_5 \quad d_5 \times d_6$  dimensions  
 $B_1 \quad B_2$

Suppose the last matrix multiplication were at  $A_3$ ; then

- 1) We need to multiply  $A_1 \times A_2 \times A_3$  to create  $B_1$ , a  $d_0 \times d_3$  matrix
- 2) We need to multiply  $A_4 \times A_5 \times A_6$  to create  $B_2$ , a  $d_3 \times d_6$  matrix

Cost would be the cost of (1)+(2)+ cost of( $B_1 \times B_2$ )

$A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6$  matrices

$d_0 \times d_1 \quad d_1 \times d_2 \quad d_2 \times d_3 \quad d_3 \times d_4 \quad d_4 \times d_5 \quad d_5 \times d_6$  dimensions

The last multiplication may be at each of the 5 matrices.

$$\text{Cost}((A_1 A_2 A_3 A_4 A_5)(A_6)) = \text{Cost}(A_1 A_2 A_3 A_4 A_5) + \text{Cost}(A_6) \\ + d_0 \times d_5 \times d_6$$

$$\text{Cost}((A_1 A_2 A_3 A_4)(A_5 A_6)) = \text{Cost}(A_1 A_2 A_3 A_4) + \text{Cost}(A_5 A_6) \\ + d_0 \times d_4 \times d_6$$

$$\text{Cost}((A_1 A_2 A_3)(A_4 A_5 A_6)) = \text{Cost}(A_1 A_2 A_3) + \text{Cost}(A_4 A_5 A_6) \\ + d_0 \times d_3 \times d_6$$

$$\text{Cost}((A_1 A_2)(A_3 A_4 A_5 A_6)) = \text{Cost}(A_1 A_2) + \text{Cost}(A_3 A_4 A_5 A_6) \\ + d_0 \times d_2 \times d_6$$

$$\text{Cost}((A_1)(A_2 A_3 A_4 A_5 A_6)) = \text{Cost}(A_1) + \text{Cost}(A_2 A_3 A_4 A_5 A_6) \\ + d_0 \times d_1 \times d_6$$

$A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6$  matrices

$d_0 \times d_1 \quad d_1 \times d_2 \quad d_2 \times d_3 \quad d_3 \times d_4 \quad d_4 \times d_5 \quad d_5 \times d_6$  dimensions

The dynamic programming selection rule: **when given a number of possibilities, compute all and take the best.**

The optimal cost of multiplying the 6 matrices:

$$\begin{aligned} \text{OptCost}(A_1 A_2 A_3 A_4 A_5 A_6) = \text{Min}(\quad & \\ \text{OptCost}(A_1 A_2 A_3 A_4 A_5) + \text{OptCost}(A_6) + d_0 \times d_5 \times d_6, & \\ \text{OptCost}(A_1 A_2 A_3 A_4) + \text{OptCost}(A_5 A_6) + d_0 \times d_4 \times d_6, & \\ \text{OptCost}(A_1 A_2 A_3) + \text{OptCost}(A_4 A_5 A_6) + d_0 \times d_3 \times d_6, & \\ \text{OptCost}(A_1 A_2) + \text{OptCost}(A_3 A_4 A_5 A_6) + d_0 \times d_2 \times d_6, & \\ \text{OptCost}(A_1) + \text{OptCost}(A_2 A_3 A_4 A_5 A_6) + d_0 \times d_1 \times d_6 ) & \end{aligned}$$

$$\text{OptCost}(A) = 0$$

Step 2: Turn this formulation into a recursive function to solve the chain matrix multiplication problem.

Suppose we use array **d** to store the dimensions of the matrices.

$d_0$	$d_1$	$d_2$	$\dots$		
-------	-------	-------	---------	--	--

Let  $\text{OptCost}(i,j)$  be the optimal cost of multiplying matrices with dimensions  $d_i \times d_{i+1}$ ,  $d_{i+1} \times d_{i+2}$ , ...,  $d_{j-1} \times d_j$ .

$$\text{OptCost}(i, j) = 0 \quad \text{if } j-i=1$$

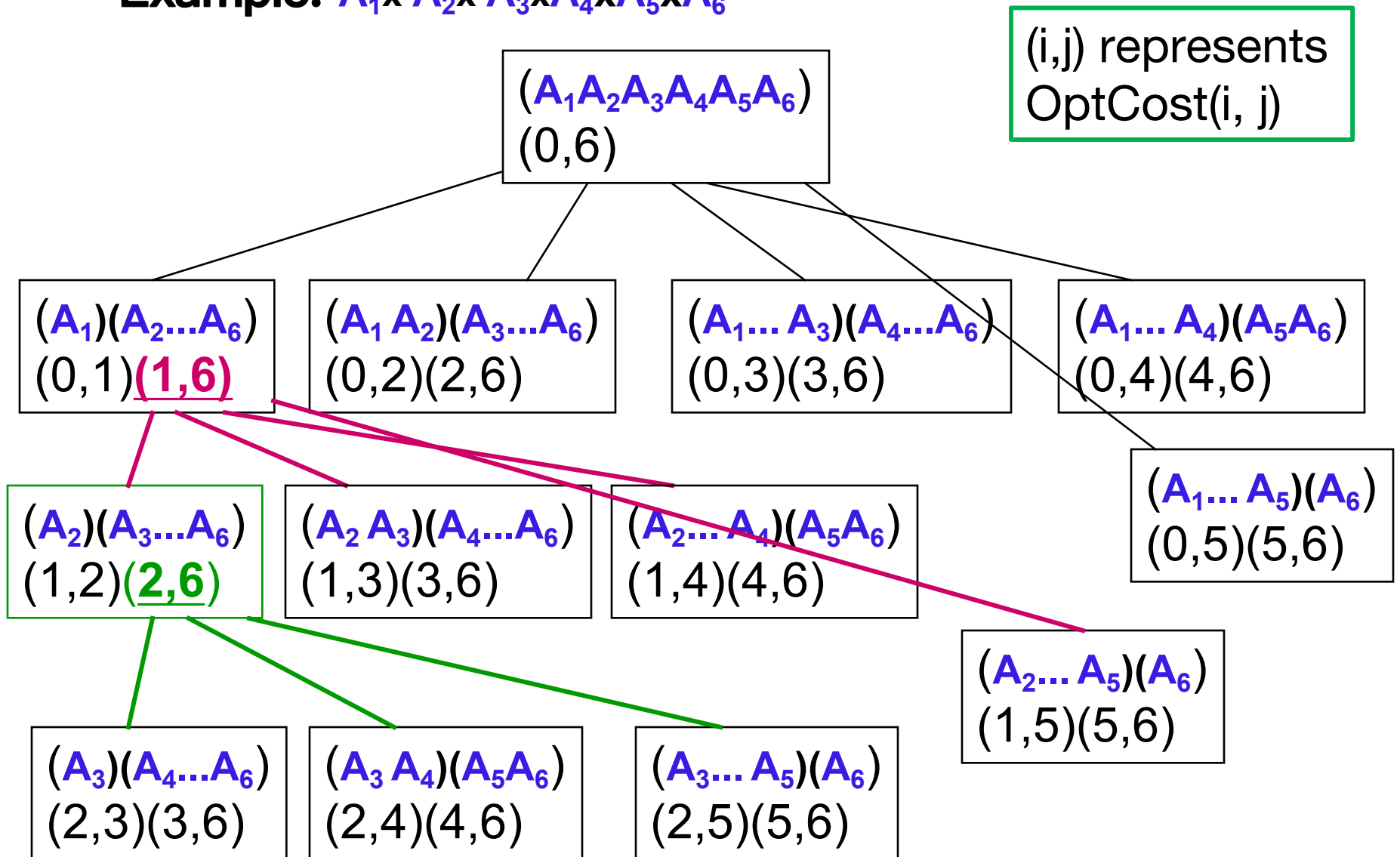
$$\begin{aligned} \text{OptCost}(i, j) \\ = \min_{i+1 \leq k \leq j-1} (\text{OptCost}(i,k) + \text{OptCost}(k, j) + d_i \times d_k \times d_j) \end{aligned} \quad \text{if } j-i > 1$$

The optimal cost of multiplying  $n$  matrices is  $\text{OptCost}(0, n)$ .

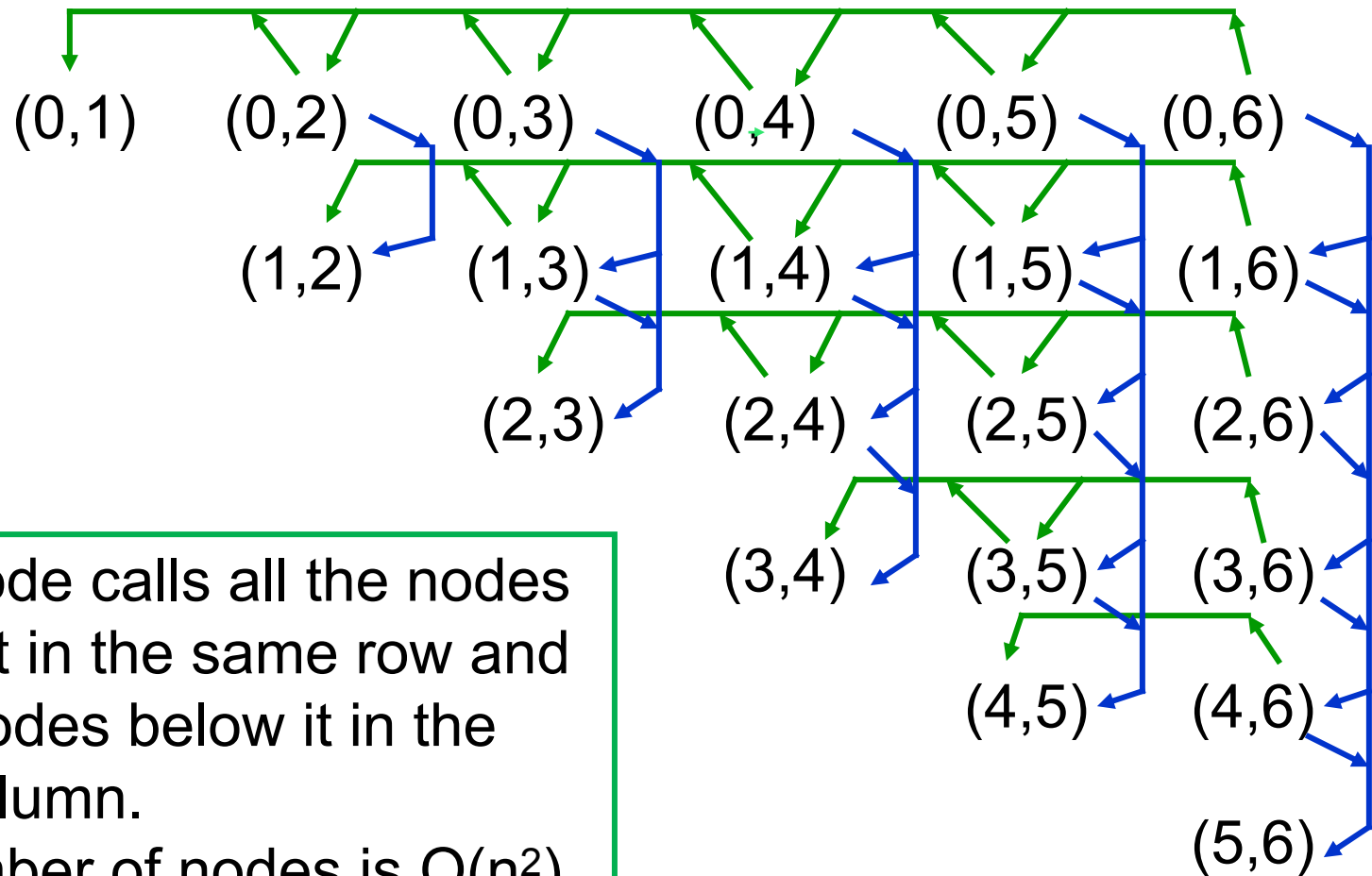
- The chain matrix multiplication problem satisfies the principle of optimality
  - $\text{OptCost}(i, k)$  and  $\text{OptCost}(k, j)$  for  $k = i+1, \dots, j-1$  are the subsolutions of  $\text{OptCost}(i, j)$
  - They are the optimal solutions for the subproblems
  - Proof by contradiction



# Example: $A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6$

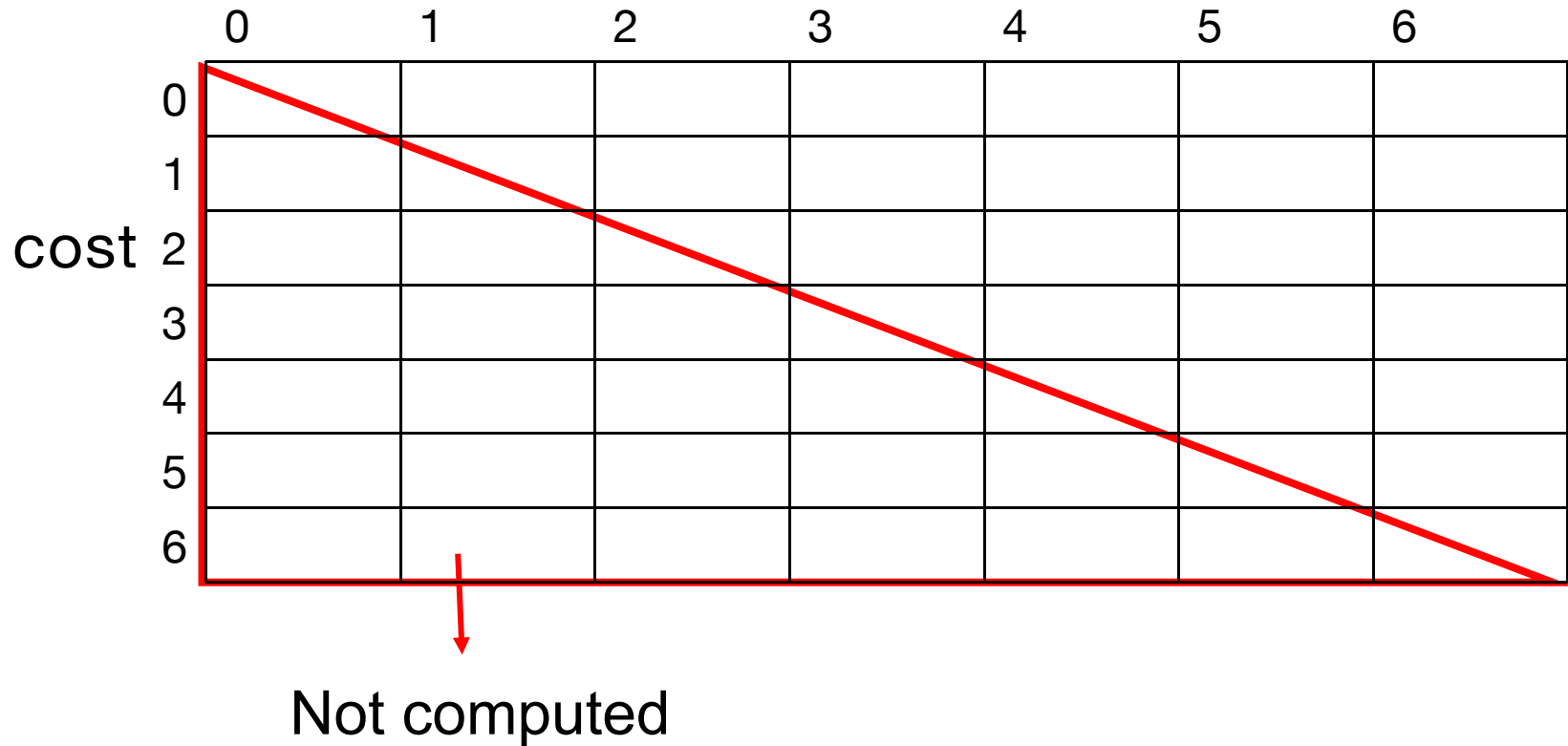


Step 3: Draw the subproblem graph and find the dependencies among subproblems



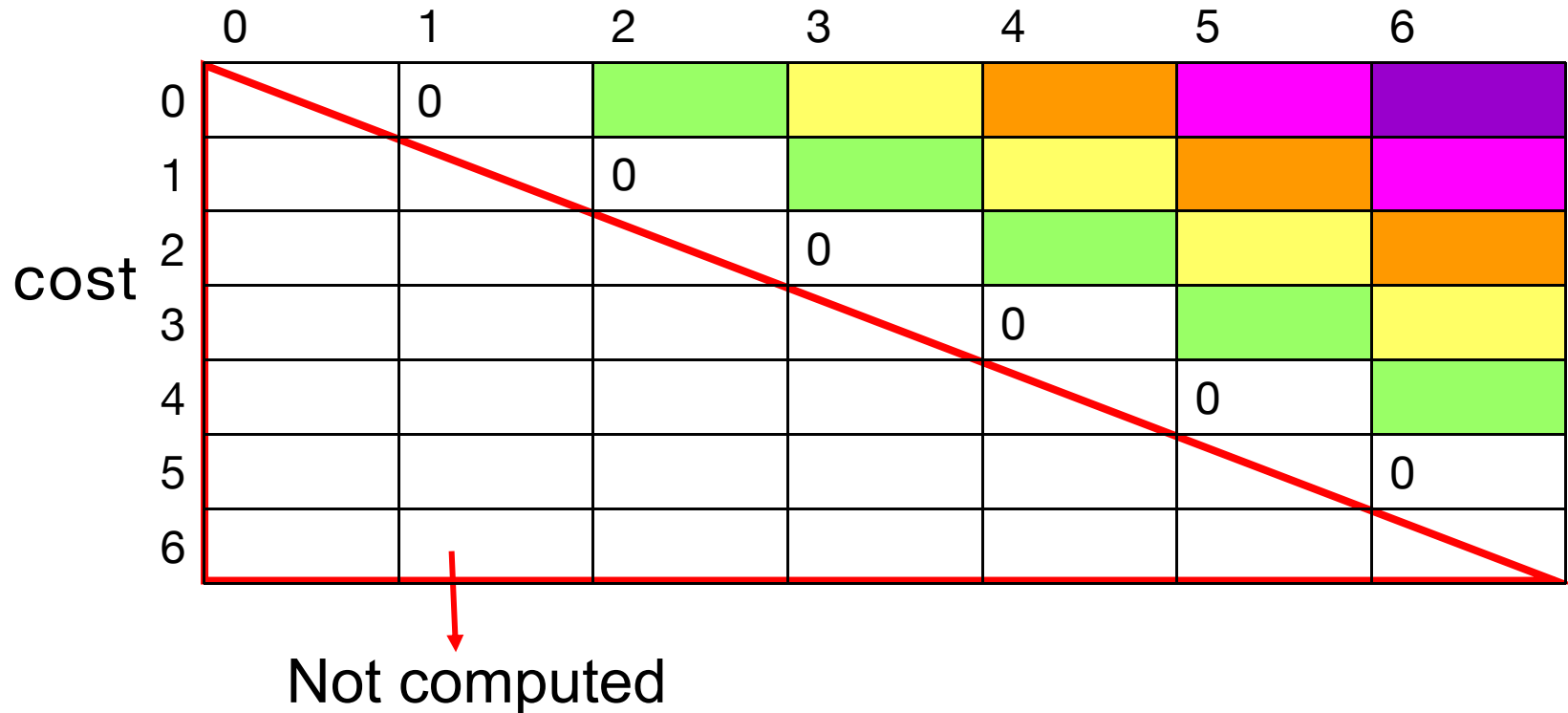
Every node calls all the nodes on its left in the same row and all the nodes below it in the same column.  
The number of nodes is  $O(n^2)$ .

## Step 4: Dictionary: $\text{cost}[n+1][n+1]$



## Step 5

Order to solve the subproblems

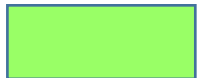


Use another array,  $last[n+1][n+1]$  to represent the index of the last multiplication to be done for a subproblem

## Find the pattern

	0	1	2	3	4	5	6
0		0					
1			0				
2				0			
3					0		
4						0	
5							0
6							

cost



Row number and column number differ by 2, row goes from 0 to 4



Row number and column number differ by 3, row goes from 0 to 3



Row number and column number differ by 4, row goes from 0 to 2

## Find the pattern

	0	1	2	3	4	5	6
0		0					
1			0				
2				0			
3					0		
4						0	
5							0
6							



Row number and column number differ by 5, row goes from 0 to 1



Row number and column number differ by 6, row goes from 0 to 0

Thus, row number and column number differ by 2 to 6, within each difference, row goes from 0 to n minus this difference

```

int matrixOrder(int [] d, int n)
{
    for i = 0 to n-1      cost[i][i+1] = 0;

    for l = 2 to n
        for i = 0 to n-l
            j = i + l;
            cost[i][j] = ∞;
            for k = i+1 to j-1

                c = cost[i][k] + cost[k][j] + d[i]*d[k]*d[j];
                if (c < cost[i][j])
                    cost[i][j] = c; last[i][j] = k;

            }
}

```

$$\min_{i+1 \leq k \leq j-1} (\text{OptCost}(i, k) + \text{OptCost}(k, j) + d_i \times d_k \times d_j)$$

$c = \text{cost}[i][k] + \text{cost}[k][j] + d[i] \times d[k] \times d[j];$   
 if ( $c < \text{cost}[i][j]$ )  
      $\text{cost}[i][j] = c; \text{last}[i][j] = k;$

```

int matrixOrder(int [] d, int n)
{
    for i = 0 to n-1      cost[i][i+1] = 0;
    for l = 2 to n
        for i = 0 to n-l

            j = i + l;
            cost[i][j] = ∞;

            for k = i+1 to j-1

                c = cost[i][k] + cost[k][j] + d[i]*d[k]*d[j];
                if (c < cost[i][j])
                    cost[i][j] = c; last[i][j] = k;

}

```

Complexity  
of computing  
the optimal  
order:  $O(n^3)$

Repeated  $O(n^2)$  times

Repeated  $O(n^3)$  times



# Example

**$A_1 \times A_2 \times A_3 \times A_4$**

**30x1    1x40    40x10    10x25**

Array d

<b>30</b>	<b>1</b>	<b>40</b>	<b>10</b>	<b>25</b>
<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>

Call to matrixOrder(d, 4)

d	30	1	40	10	25
	0	1	2	3	4

cost

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

for  $i = 0$  to  $n-1$   
 $\text{cost}[i][i+1] = 0;$

d	30	1	40	10	25
	0	1	2	3	4

cost

	0	1	2	3	4
0		0	1200		
1			0		
2				0	
3					0
4					

	0	1	2	3	4
0			1		
1					
2					
3					
4					

last

for l = 2 to n

for i = 0 to n-l

j = i + l;

cost[i][j] =  $\infty$ ;

for k = i+1 to j-1

c = cost[i][k] + cost[k][j]  
+ d[i]\*d[k]\*d[j];

if (c < cost[i][j])

cost[i][j] = c;

last[i][j] = k;

l=2, i=0, j=2, k=1

k=1:  
Cost[0][1]+  
Cost[1][2] + 1200

d	30	1	40	10	25
	0	1	2	3	4

cost

	0	1	2	3	4
0		0	1200		
1			0	400	
2				0	
3					0
4					

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

last

for l = 2 to n

for i = 0 to n-l

j = i + l;

cost[i][j] =  $\infty$ ;

for k = i+1 to j-1

c = cost[i][k] + cost[k][j]  
+ d[i]\*d[k]\*d[j];

if (c < cost[i][j])

cost[i][j] = c;

last[i][j] = k;

l=2, i=1, j=3, k=2

k=2:  
Cost[1][2]+  
Cost[2][3] + 400

d

30	1	40	10	25
0	1	2	3	4

cost

	0	1	2	3	4
0		0	1200		
1			0	400	
2				0	1000
3					0
4					
	0	1	2	3	4

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

last

for l = 2 to n

for i = 0 to n-l

j = i + l;

cost[i][j] =  $\infty$ ;

for k = i+1 to j-1

c = cost[i][k] + cost[k][j]  
+ d[i]\*d[k]\*d[j];

if (c < cost[i][j])

cost[i][j] = c;

last[i][j] = k;

l=2, i=2, j=4, k=3

k=3:  
Cost[2][3]+  
Cost[3][4] + 10000

d

30	1	40	10	25
0	1	2	3	4

cost

	0	1	2	3	4
0		0	1200	700	
1			0	400	
2				0	1000
3					0
4					
	0	1	2	3	4

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

last

for l = 2 to n

for i = 0 to n-l

j = i + l;

cost[i][j] =  $\infty$ ;

for k = i+1 to j-1

c = cost[i][k] + cost[k][j]  
+ d[i]\*d[k]\*d[j];

if (c < cost[i][j])

cost[i][j] = c;

last[i][j] = k;

l=3, i=0, j=3, k=1,2

k=1:  
Cost[0][1]+  
Cost[1][3] + 300  
k=2:  
Cost[0][2]+  
Cost[2][3] + 12000

d

30	1	40	10	25
0	1	2	3	4

cost

	0	1	2	3	4
0		0	1200	700	
1			0	400	650
2				0	1000
3					0
4					
	0	1	2	3	4

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

last

for l = 2 to n

for i = 0 to n-l

j = i + l;

cost[i][j] = ∞;

for k = i+1 to j-1

c = cost[i][k] + cost[k][j]  
+ d[i]\*d[k]\*d[j];

if (c < cost[i][j])

cost[i][j] = c;

last[i][j] = k;

l=3, i=1, j=4, k=2,3

k=2:  
Cost[1][2]+  
Cost[2][4] + 1000  
k=3:  
Cost[1][3]+  
Cost[3][4] + 250

d	30	1	40	10	25
	0	1	2	3	4

cost

	0	1	2	3	4
0		0	1200	700	1400
1			0	400	650
2				0	1000
3					0
4					

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

last

for l = 2 to n

for i = 0 to n-l

j = i + l;

cost[i][j] =  $\infty$ ;

for k = i+1 to j-1

c = cost[i][k] + cost[k][j]  
+ d[i]\*d[k]\*d[j];

if (c < cost[i][j])

cost[i][j] = c;

last[i][j] = k;

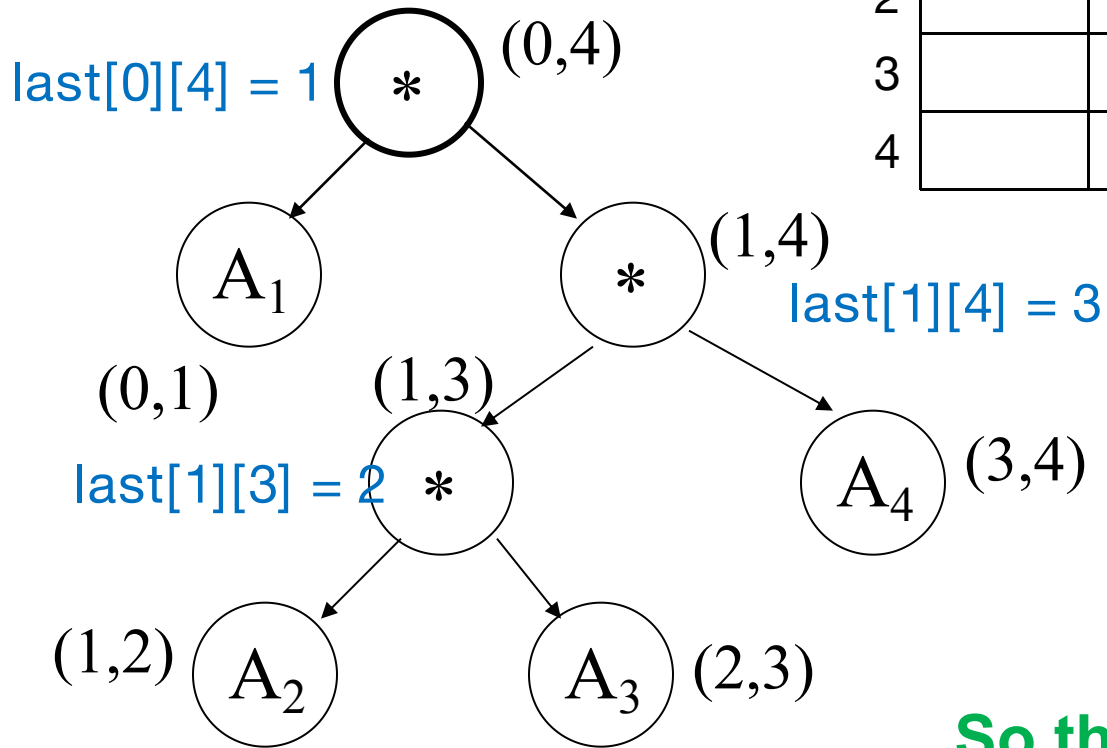
l=4, i=0, j=4, k=1,2,3

k=1:  
Cost[0][1]+  
Cost[1][4] + 750  
k=2:  
Cost[0][2]+  
Cost[2][4] + 30000  
k=3:  
Cost[0][3]+  
Cost[3][4] + 7500



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

last



**So the best sequence is  
(A1 x ((A2 x A3 ) x A4 ))**

# 0/1 Knapsack problem

Problem definition: We have a knapsack of capacity weight  $C$  (a positive integer) and  $n$  objects with weights  $w_1, w_2, \dots, w_n$  and profits  $p_1, p_2, \dots, p_n$  (all  $w_i$  and all  $p_i$  are positive integers), find the largest total profit of any subset of the objects that fits in the knapsack.

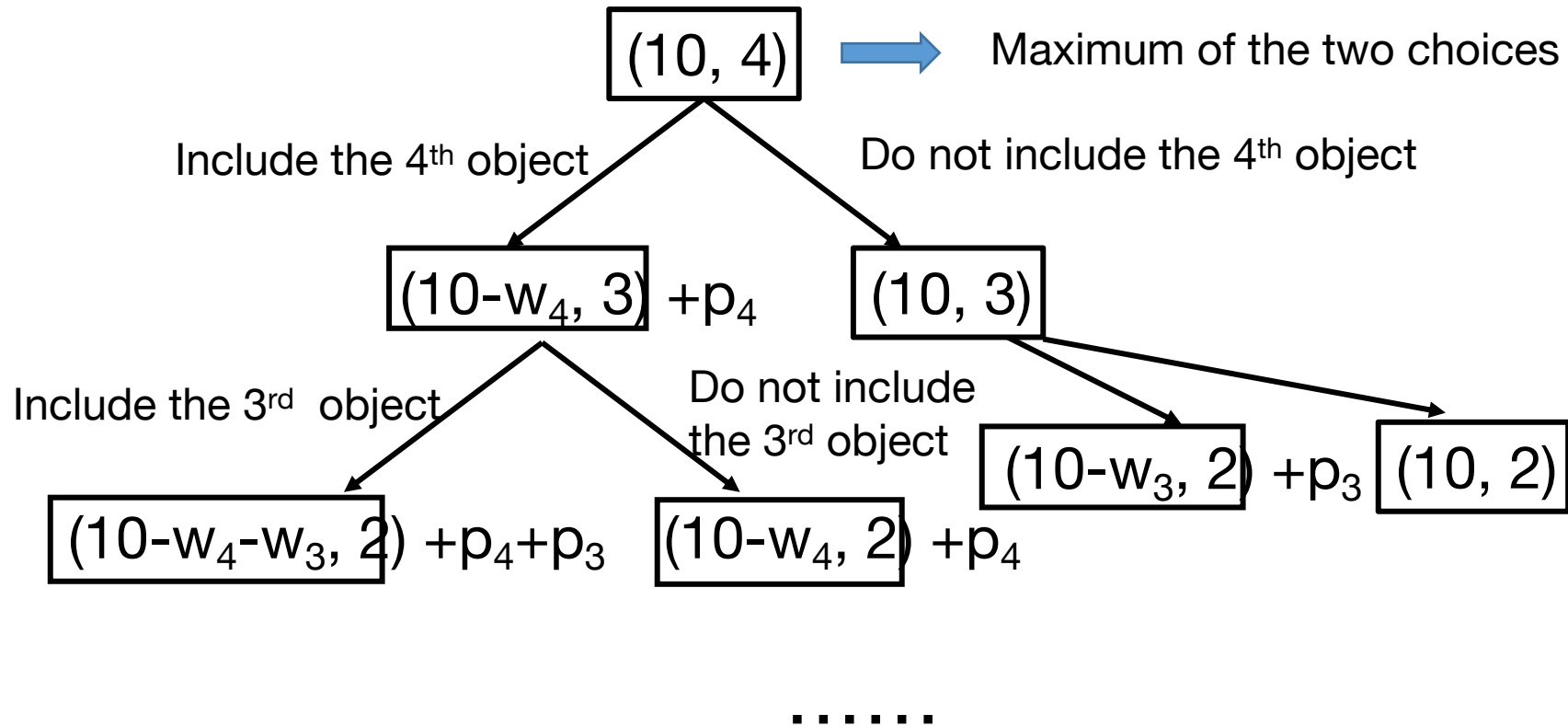
- We have to take whole objects.
- There are  $2^n$  subsets of  $n$  objects: examining all subsets takes  $O(2^n)$  time

E.g. application:  $C$  is amount of money to invest, weights,  $w_1, \dots$  are investment amounts and profit is the expected return on investment.



- Step 1: formulate the 0/1 knapsack problem in terms of smaller versions of the same problem
  - Consider the last object of the  $n$  objects with weights  $w_1, w_2, \dots, w_n$ .
  - If we include it in the knapsack, the available weight capacity in the knapsack will be reduced by  $w_n$ . Then our profit will be  $p_n$ , plus the maximum we can get from solving the subproblem of  $n-1$  objects and capacity of  $C - w_n$ .
  - If we do not include it in the knapsack, our profit will be the maximum we can get from solving the subproblem of  $n-1$  objects and capacity of  $C$ .
  - The dynamic programming selection rule: **when given a number of possibilities, compute all and take the best.**

For example, a knapsack of capacity 10 and 4 objects



Step 2: Turn this formulation into a recursive function to solve the 0/1 knapsack problem

Let  $P(C, j)$  be the maximum profit that can be made by selecting a subset of the  $j$  objects with knapsack capacity of  $C$ .

$$P(C, 0) = P(0, j) = 0$$

$$P(C, j) = \max(P(C, j-1), p_j + P(C-w_j, j-1))$$

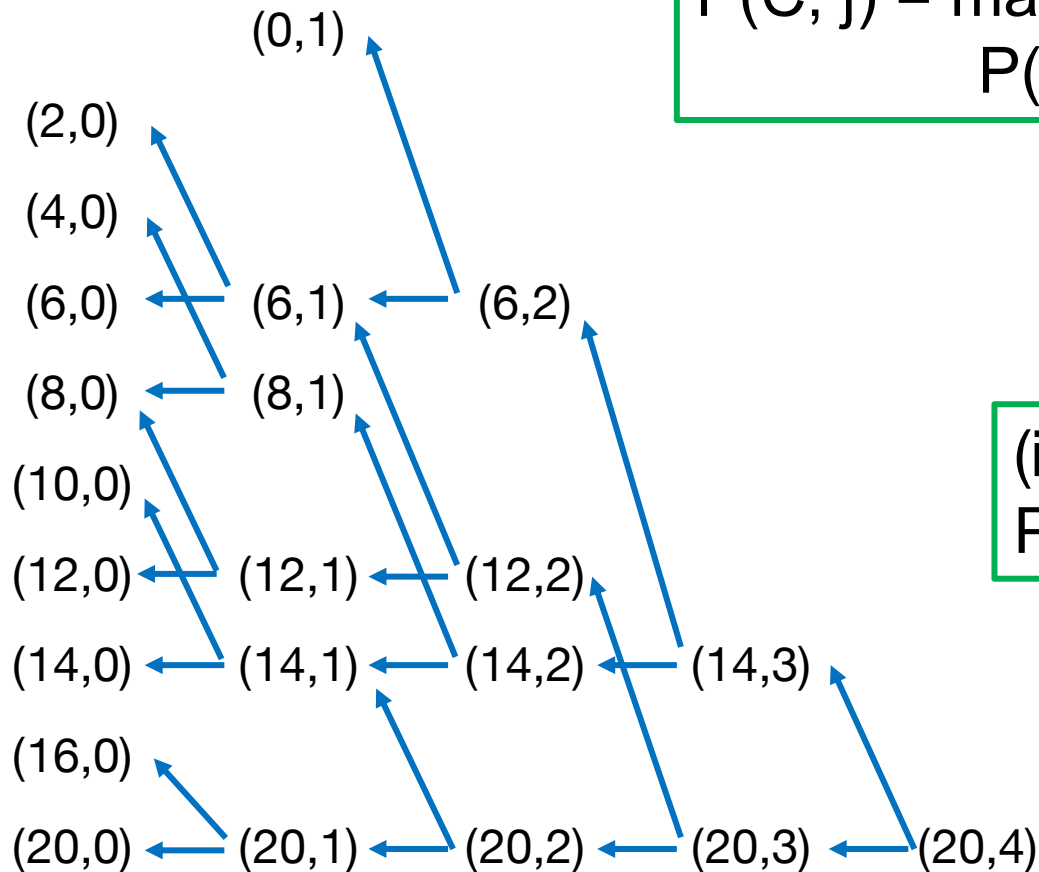
- Step 3: Draw the subproblem graph and find the dependencies among subproblems

For example,  $C = 20$

	1	2	3	4
$w_i$	4	6	8	6
$p_i$	7	6	9	5

	1	2	3	4
$w_i$	4	6	8	6
$p_i$	7	6	9	5

$$P(C, j) = \max(P(C, j-1), p_j + P(C-w_j, j-1))$$



$(i, j)$  represents  $P(i, j)$

## Step 4 :Dictionary: $\text{profit}[C+1][n+1]$

	0	1	2	...	n
0					
1					
2					
3					
4					
5					
6					
7					
...					
C					

int knapsack(int [] w, int [] p, int C, int n) Step 5

{ for c = 0 to n profit[0][c] = 0;

for r = 1 to C profit[r][0] = 0;

for r = 1 to C

for c = 1 to n

profit[r][c] = profit[r][c-1];

if (w[c] <= r)

if (profit[r][c] < profit[r-w[c]][c-1] + p[c])

profit[r][c] = profit[r-w[c]][c-1] + p[c];

}

Complexity :  
 $O(nC)$

$$P(C, j) = \max(P(C, j-1), p_j + P(C-w_j, j-1))$$



Example 1:  
C = 20

	1	2	3	4
$w_i$	4	6	8	6
$p_i$	7	6	9	5

	0	1	2	3	4
0	0	0	0	0	0
2	0				
4	0				
6	0				
8	0				
10	0				
12	0				
14	0				
16	0				
20	0				

for c = 0 to n  
 profit[0][c] =  
 0;  
 for r = 1 to C  
 profit[r][0] =  
 0;  
 Not all rows  
 are shown

Example 1:  
C = 20

	1	2	3	4
$w_i$	4	6	8	6
$p_i$	7	6	9	5

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	7	7	7	7
5	0	7	7	7	7
6	0	7	7	7	7
8	0	7	7	9	9
9	0	7	7	9	9
10	0	7	13	13	13
11	0	7	13	13	13

for r = 1 to C

for c = 1 to n

profit[r][c] = profit[r][c-1];

if (w[c] <= r)

if (profit[r][c] < profit[r-w[c]][c-1] + p[c])

profit[r][c] = profit[r-w[c]][c-1] + p[c]

Example 1:  
C = 20

	1	2	3	4
$w_i$	4	6	8	6
$p_i$	7	6	9	5

	0	1	2	3	4
10	0	7	13	13	13
11	0	7	13	13	13
12	0	7	13	16	16
13	0	7	13	16	16
14	0	7	13	16	16
15	0	7	13	16	16
16	0	7	13	16	18
17	0	7	13	16	18
18	0	7	13	22	22
19	0	7	13	22	22
20	0	7	13	22	22

for r = 1 to C

for c = 1 to n

profit[r][c] = profit[r][c-1];

if (w[c] <= r)

if (profit[r][c] < profit[r-w[c]][c-1] + p[c])

profit[r][c] = profit[r-w[c]][c-1] + p[c]

Example 2:  
C = 3

	1	2	3
$w_i$	1	2	3
$p_i$	1	4	6

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

```

for c = 0 to n
    profit[0][c] =
        0;
for r = 1 to C
    profit[r][0] =
        0;

```

Example 2:  
C = 3

	1	2	3
w <sub>i</sub>	1	2	3
p <sub>i</sub>	1	4	6

	0	1	2	3
0	0	0	0	0
1	0	1	1	1
2	0	1	4	4
3	0	1	5	6

for r = 1 to C

for c = 1 to n

profit[r][c] = profit[r][c-1];

if (w[c] <= r)

if (profit[r][c] <  
profit[r-w[c]][c-1] + p[c])

profit[r][c] =  
profit[r-w[c]][c-1]  
+ p[c]

- The dynamic programming algorithm has a complexity of  $O(nC)$ .
- An algorithm is polynomial time if it is a polynomial function of the size of the input.  
E.g. there are  $n$  weight numbers and  $n$  profit numbers
- An algorithm is pseudo-polynomial time if it is a polynomial function of the value of the input.  
E.g. there is only one number specifying  $C$
- So the dynamic programming algorithm for knapsack problem is pseudo-polynomial.