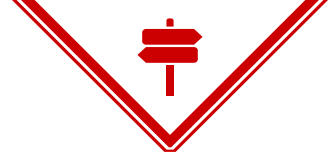




# *Dynamic Programming*



# *Dynamic Programming – The IDEA*

- ◆ A technique for solving problems with overlapping sub – problems.
- ◆ These sub – problems arise from a recurrence relating a solution to a given problem with solutions to its smaller sub – problems of the same type.
- ◆ Ex: Fibonacci Number Sequence



# *Dynamic Programming – The IDEA*

- ◆ Dynamic Programming suggests:
  - ◆ Solve each of these sub – problems only once
  - ◆ Store the results in a table
  - ◆ Use these results to obtain solutions to the original problem.



# *Dynamic Programming – An EXAMPLE*

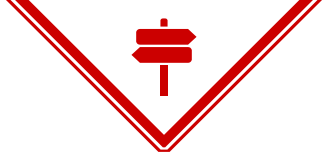
## FIBONACCI NUMBERS

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, .....

Recurrence Relation:

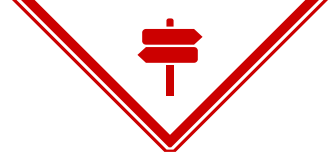
$$F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2$$

$$F(0) = 0, F(1) = 1$$



# *Dynamic Programming – An EXAMPLE*

- ◆ If the recurrence is directly used to compute the  $n^{\text{th}}$  Fibonacci number, we would have to recompute the same values again and again.
- ◆ Ex:  $F(6) = F(5) + F(4)$   
 $F(5) = F(4) + F(3)$   
 $F(4) = F(3) + F(2) \dots\dots$



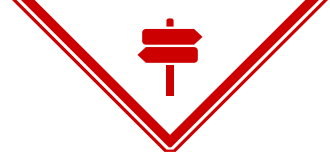
# *Dynamic Programming – An EXAMPLE*

- ◆ Instead, we can simply fill elements of a one – dimensional array with the  $n + 1$  consecutive values of  $F(n)$  by starting with the initial elements and using the equation to produce all other elements.



1

*Computing a Binomial  
Co - Efficient*

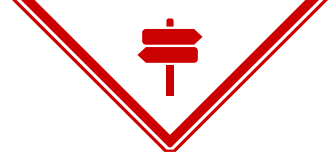


## *What is Binomial Co – Efficient?*

- ❖ **Binomial Co - efficient**, denoted  $C(n, k)$  is the number of combinations (subsets) of  $k$  elements from an  $n$ -element set ( $0 \leq k \leq n$ ).
- ❖ The name comes from the participation of these numbers in the binomial formula:

$$(a + b)^n = C(n, 0)a^n + \dots + C(n, k)a^{n-k}b^k + \dots + C(n, n)b^n.$$





## ***Binomial Co – Efficient***

### ◆ **Properties:**

- $C(n, k) = C(n-1, k-1) + C(n-1, k)$  for  $n > k > 0$
- $C(n, 0) = C(n, n) = 1$

The nature of the recurrence is such that it can be solved by Dynamic Programming.



## *Binomial Co – Efficient*

To do this, we record the values of the binomial coefficients in a table of  $n + 1$  rows and  $k + 1$  columns, numbered from 0 to  $n$  and from 0 to  $k$ , respectively

	0	1	2	...	$k-1$	$k$
0	1					
1	1	1				
2	1	2	1			
$\vdots$						
$k$	1					1
$\vdots$						
$n-1$	1			$C(n-1, k-1)$	$C(n-1, k)$	
$n$	1					$C(n, k)$



# ***Binomial Co – Efficient – The ALGORITHM***

**ALGORITHM** *Binomial*( $n, k$ )

//Computes  $C(n, k)$  by the dynamic programming algorithm

//Input: A pair of nonnegative integers  $n \geq k \geq 0$

//Output: The value of  $C(n, k)$

**for**  $i \leftarrow 0$  **to**  $n$  **do**

**for**  $j \leftarrow 0$  **to**  $\min(i, k)$  **do**

**if**  $j = 0$  **or**  $j = i$

$C[i, j] \leftarrow 1$

**else**  $C[i, j] \leftarrow C[i - 1, j - 1] + C[i - 1, j]$

**return**  $C[n, k]$



# *Binomial Co – Efficient – The EFFICIENCY*

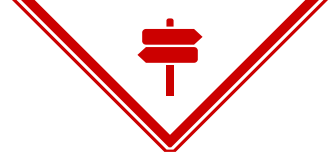
$$\begin{aligned} A(n, k) &= \sum_{i=1}^k \sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^n \sum_{j=1}^k 1 = \sum_{i=1}^k (i-1) + \sum_{i=k+1}^n k \\ &= \frac{(k-1)k}{2} + k(n-k) \in \Theta(nk). \end{aligned}$$

— — —



2

*Warshall's Algorithm*

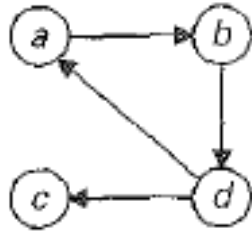


## *What is Transitive Closure?*

The transitive closure of a directed graph with  $n$  vertices can be defined as the  $n$ -by- $n$  boolean matrix  $T = \{t_{i,j}\}$ , in which the element in the  $i^{\text{th}}$  row ( $1 \leq i \leq n$ ) and the  $j^{\text{th}}$  column ( $1 \leq j \leq n$ ) is 1 if there exists a nontrivial directed path (i.e., a directed path of a positive length) from the  $i^{\text{th}}$  vertex to the  $j^{\text{th}}$  vertex; otherwise,  $t_{ij}$  is 0.



# *What is Transitive Closure?*



(a)

$$A = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{matrix}$$

(b)

$$T = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{matrix}$$

(c)

- a) Directed Graph
- b) Adjacency Matrix
- c) Transitive Closure



## *Warshall's Algorithm – The IDEA*

- ❖ Warshall's Algorithm constructs the transitive closure of a digraph with  $n$  vertices through a series of  $n$  – by –  $n$  Boolean matrices.
- ❖  $R^{(0)}, R^{(1)}, \dots, R^{(k-1)}, R^{(k)}, \dots, R^{(n)}$
- ❖ The element  $r_{ij}^{(k)}$  in the  $i^{\text{th}}$  row and  $j^{\text{th}}$  column of matrix  $R^{(k)}$  ( $k = 0, 1, \dots, n$ ) is equal to 1 if and only if there exists a directed path (of a positive length) from the  $i^{\text{th}}$  vertex to the  $j^{\text{th}}$  vertex with each intermediate vertex, if any, numbered not higher than  $k$ .





# *Warshall's Algorithm – The IDEA*

- ◆ Rules for generating elements of matrix  $R^{(k)}$  from  $R^{(k-1)}$ :
  - ◆ If an element  $r_{ij}$  is 1 in  $R^{(k)}$ , it remains so in  $R^{(k-1)}$ .
  - ◆ If an element  $r_{ij}$  is 0 in  $R^{(k-1)}$ , it can be changed to 1 in  $R^{(k)}$  if and only if the element in its row  $i$  and column  $k$  and the element in row  $k$  and column  $j$  are both 1's in  $R^{(k-1)}$ .



## *Warshall's Algorithm – The IDEA*

$$R^{(k-1)} = \begin{array}{c} \begin{array}{cc} & j & k \\ \begin{array}{c} k \\ i \end{array} & \begin{bmatrix} & & \\ 1 & & \\ & & \end{bmatrix} \end{array} \Rightarrow R^{(k)} = \begin{array}{c} \begin{array}{cc} & j & k \\ \begin{array}{c} k \\ i \end{array} & \begin{bmatrix} & & \\ 1 & & \\ 1 & & 1 \end{bmatrix} \end{array}$$

The diagram illustrates the update step of Warshall's Algorithm. On the left, the matrix  $R^{(k-1)}$  is shown with indices  $j$  and  $k$  for columns and  $k$  and  $i$  for rows. A horizontal bar highlights the row  $k$ , and a vertical bar highlights the column  $k$ . The intersection cell  $R^{(k-1)}[k][k]$  contains the value 1. Below this cell, an upward arrow points from the value 0, and a rightward arrow points to the value 1, indicating the update rule: if  $R^{(k-1)}[i][k] = 1$  and  $R^{(k-1)}[k][j] = 1$ , then  $R^{(k)}[i][j] = 1$ . On the right, the updated matrix  $R^{(k)}$  is shown, where the cell  $R^{(k)}[i][j]$  now contains the value 1, in addition to the other 1s from the previous step.



# *Warshall's Algorithm*

**ALGORITHM** *Warshall*( $A[1..n, 1..n]$ )

//Implements Warshall's algorithm for computing the transitive closure

//Input: The adjacency matrix  $A$  of a digraph with  $n$  vertices

//Output: The transitive closure of the digraph

$R^{(0)} \leftarrow A$

**for**  $k \leftarrow 1$  **to**  $n$  **do**

**for**  $i \leftarrow 1$  **to**  $n$  **do**

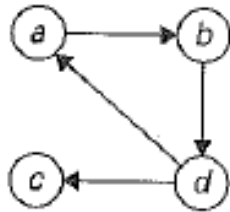
**for**  $j \leftarrow 1$  **to**  $n$  **do**

$R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j] \text{ or } (R^{(k-1)}[i, k] \text{ and } R^{(k-1)}[k, j])$

**return**  $R^{(n)}$



# *Warshall's Algorithm – An EXAMPLE*



$$R^{(0)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{matrix}$$

Ones reflect the existence of paths with no intermediate vertices ( $R^{(0)}$  is just the adjacency matrix); boxed row and column are used for getting  $R^{(1)}$ .

$$R^{(1)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{matrix}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex  $a$  (note a new path from  $d$  to  $b$ ); boxed row and column are used for getting  $R^{(2)}$ .



# *Warshall's Algorithm – An EXAMPLE*

$$R^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e.,  $a$  and  $b$  (note two new paths); boxed row and column are used for getting  $R^{(3)}$ .

$$R^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e.,  $a$ ,  $b$ , and  $c$  (no new paths); boxed row and column are used for getting  $R^{(4)}$ .

$$R^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 1 & 1 & 1 & 1 \\ b & 1 & 1 & 1 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e.,  $a$ ,  $b$ ,  $c$ , and  $d$  (note five new paths).

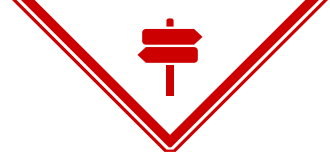


# *Warshall's Algorithm*

**Efficiency:  $\Theta(n^3)$**

3

*Floyd's Algorithm for  
All Pairs Shortest Path*



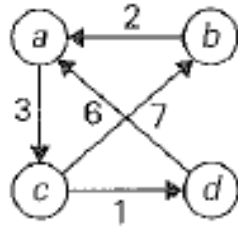
## *All Pairs Shortest Path*

- The *all-pairs shortest paths problem* asks to find the distances (the lengths of the shortest paths) from each vertex to all other vertices.
- It is convenient to record the lengths of shortest paths in an  $n$ -by- $n$  matrix  $D$  called the *distance matrix*: the element  $d_{ij}$  in the  $i^{\text{th}}$  row and the  $j^{\text{th}}$  column of this matrix indicates the length of the shortest path from the  $i^{\text{th}}$  vertex to the  $j^{\text{th}}$  vertex ( $1 \leq i, j \leq n$ ).





## *All pairs shortest path*



(a)

$$W = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{matrix}$$

(b)

$$D = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{matrix}$$

(c)

**FIGURE 8.5** (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.



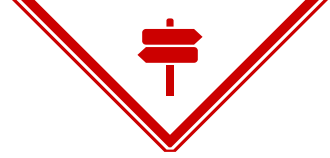
## *Floyd's Algorithm – The IDEA*

- ◆ Floyd's Algorithm computes the distance matrix of a weighted graph with  $n$  vertices through a series of  $n - by - n$  matrices:
- ◆  $D(0), D(1), \dots, D(k-1), D(k), \dots, D(n)$
- ◆ The element  $d_{ij}^{(k)}$  in the  $i^{th}$  row and  $j^{th}$  column of matrix  $D^{(k)}$  is equal to the length of the shortest path among all paths from the  $i^{th}$  vertex to the  $j^{th}$  vertex with each intermediate vertex, if any, numbered not higher than  $k$ .



## *Floyd's Algorithm – The IDEA*

- ◆  $d_{ij}^{(k)}$  is equal to the length of the shortest path among all paths from  $i^{\text{th}}$  vertex  $v_i$  to  $j^{\text{th}}$  vertex  $v_j$  with their intermediate vertices not numbered higher than  $k$ .
- ◆ All such paths can be partitioned into two disjoint subsets: ones which do not use  $v_k$  as the intermediate vertex and which do.



## *Floyd's Algorithm – The IDEA*

- ◆ Taking into account both these subsets lead to the following recurrence:

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1, \quad d_{ij}^{(0)} = w_{ij}.$$



# *Floyd's Algorithm*

**ALGORITHM** *Floyd*( $W[1..n, 1..n]$ )

//Implements Floyd's algorithm for the all-pairs shortest-paths problem

//Input: The weight matrix  $W$  of a graph with no negative-length cycle

//Output: The distance matrix of the shortest paths' lengths

$D \leftarrow W$  //is not necessary if  $W$  can be overwritten

**for**  $k \leftarrow 1$  **to**  $n$  **do**

**for**  $i \leftarrow 1$  **to**  $n$  **do**

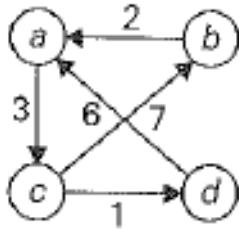
**for**  $j \leftarrow 1$  **to**  $n$  **do**

$D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$

**return**  $D$



# *Floyd's Algorithm – An EXAMPLE*



$$D^{(0)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{matrix}$$

Lengths of the shortest paths  
with no intermediate vertices  
( $D^{(0)}$  is simply the weight matrix).

$$D^{(1)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{matrix}$$

Lengths of the shortest paths  
with intermediate vertices numbered  
not higher than 1, i.e. just  $a$   
(note two new shortest paths from  
 $b$  to  $c$  and from  $d$  to  $c$ ).



## *Floyd's Algorithm – An EXAMPLE*

$$D^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & 5 & \infty \\ c & \mathbf{9} & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \end{array}$$

Lengths of the shortest paths  
with intermediate vertices numbered  
not higher than 2, i.e.  $a$  and  $b$   
(note a new shortest path from  $c$  to  $a$ ).

$$D^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \mathbf{10} & 3 & \mathbf{4} \\ b & 2 & 0 & 5 & \mathbf{6} \\ c & 9 & 7 & 0 & 1 \\ d & \mathbf{6} & \mathbf{16} & 9 & 0 \end{array}$$

Lengths of the shortest paths  
with intermediate vertices numbered  
not higher than 3, i.e.  $a$ ,  $b$ , and  $c$   
(note four new shortest paths from  $a$  to  $b$ ,  
from  $a$  to  $d$ , from  $b$  to  $d$ , and from  $d$  to  $b$ ).

$$D^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & \mathbf{7} & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

Lengths of the shortest paths  
with intermediate vertices numbered  
not higher than 4, i.e.  $a$ ,  $b$ ,  $c$ , and  $d$   
(note a new shortest path from  $c$  to  $a$ ).



# *Floyd's Algorithm*

**Efficiency:  $\Theta(n^3)$**





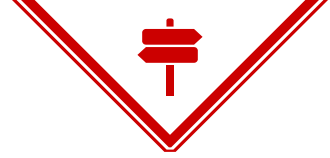
4

*Knapsack Problem and  
Memory Functions*



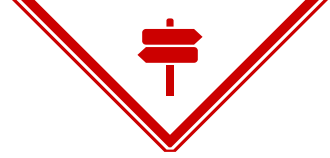
# *Knapsack Problem*

- ◆ Given ' $n$ ' items of known weights  $w_1, w_2, w_3, \dots, w_n$ , and values  $v_1, v_2, v_3, \dots, v_n$  and a knapsack of capacity  $W$ , find the most valuable subset of items that fit into the knapsack.
- ◆ We have already solved this problem using Exhaustive Search.
- ◆ Now, we shall see how to solve the same problem using Dynamic Programming.



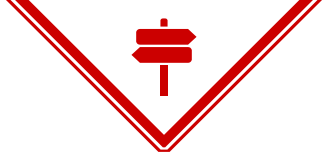
# *Knapsack Problem*

- ◆ Let us consider an instance defined by the first  $I$  items  $1 \leq i \leq n$ , with weights  $w_1, w_2, w_3, \dots, w_i$ , and values  $v_1, v_2, v_3, \dots, v_i$  and a knapsack of capacity  $j$ ,  $1 \leq j \leq W$ .
- ◆ Let  $V[i, j]$  be the value of an optimal solution to this instance.
- ◆ We can divide all the subsets of the first  $i$  items that fit into knapsack of capacity  $j$  into two categories: those that do not include the  $i^{\text{th}}$  item and those that do.



# *Knapsack Problem*

- ◆ Among the subsets that do not include the  $i^{\text{th}}$  item, the value of an optimal subset is, by definition,  $V[i - 1, j]$ .
- ◆ Among the subsets that do include the  $i^{\text{th}}$  item (hence,  $j - w_i \geq 0$ ), an optimal subset is made up of this item and an optimal subset of the first  $i - 1$  items that fit into the knapsack of capacity  $j - w_i$ . The value of such an optimal subset is  $v_i + V[i - 1, j - w_i]$ .



## *Knapsack Problem – Recurrence and Initial Condition*

$$V[i, j] = \begin{cases} \max\{V[i-1, j], v_i + V[i-1, j-w_i]\} & \text{if } j - w_i \geq 0 \\ V[i-1, j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0, j] = 0 \text{ for } j \geq 0 \text{ and } V[i, 0] = 0 \text{ for } i \geq 0.$$

Our goal is to find  $V[n, W]$



## *Table For Solving Knapsack Problem*

		0	$j-w_i$	$j$	$W$
$w_i, v_i$	0	0	0	0	0
	$i-1$	0	$V[i-1, j-w_i]$	$V[i-1, j]$	
	$i$	0		$V[i, j]$	
	$n$	0			goal



## *Table For Solving Knapsack Problem*

- ◆ For  $i, j > 0$ , to compute the entry in the  $i^{\text{th}}$  row and the  $j^{\text{th}}$  column,  $V[i, j]$ , we compute the maximum of the entry in the previous row and the same column and the sum of  $v_i$  and the entry in the previous row and  $w_i$  columns to the left.
- ◆ The table can be filled either row by row or column by column.



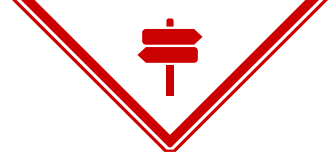
## *Knapsack Problem - Example*

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity  $W = 5$

		capacity $j$					
$i$		0	1	2	3	4	5
0		0	0	0	0	0	0
1	$w_1 = 2, v_1 = 12$	0	0	12	12	12	12
2	$w_2 = 1, v_2 = 10$	0	10	12	22	22	22
3	$w_3 = 3, v_3 = 20$	0	10	12	22	30	32
4	$w_4 = 2, v_4 = 15$	0	10	15	25	30	37





## *Knapsack Problem – Efficiency*

- ◆ The time efficiency and space efficiency are both  $\Theta(nW)$ .
- ◆ The time needed to find the composition of an optimal solution is  $O(n + W)$ .



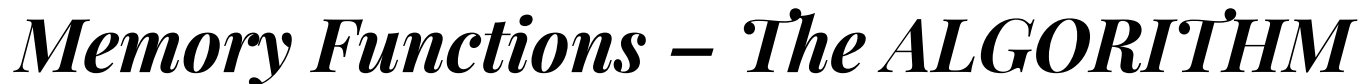
# *Memory Functions*

- ❖ Dynamic Programming deals with problems which satisfy a recurrence relation with overlapping sub problems.
- ❖ The top - down approach to solve such a recurrence leads to an algorithm that solves common sub problems more than once.
- ❖ The dynamic programming approach (bottom - up) solves these sub problems only once.
- ❖ But the disadvantage is that sometimes it solves sub problems which are not necessary for solving the problem for the given instance.
- ❖ The goal is to build a method which solves only those sub problems which are necessary.



# *Memory Functions*

- ◆ This method solves a problem in the top – down manner but maintains a table similar to the one maintained by the bottom – up approach.
- ◆ Initially, all the table's entries are initialized with a special "null" symbol to indicate that they have not yet been calculated. Thereafter, whenever a new value needs to be calculated, the method checks the corresponding entry in the table first:
  - if this entry is not "null," it is simply retrieved from the table;
  - otherwise, it is computed by the recursive call whose result is then recorded in the table.



```

//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer  $i$  indicating the number of the first
//       items being considered and a nonnegative integer  $j$  indicating
//       the knapsack's capacity
//Output: The value of an optimal feasible subset of the first  $i$  items
//Note: Uses as global variables input arrays  $Weights[1..n]$ ,  $Values[1..n]$ ,
//and table  $V[0..n, 0..W]$  whose entries are initialized with  $-1$ 's except for
//row 0 and column 0 initialized with 0's
if  $V[i, j] < 0$ 
    if  $j < Weights[i]$ 
         $value \leftarrow MFKnapsack(i - 1, j)$ 
    else
         $value \leftarrow \max(MFKnapsack(i - 1, j),$ 
                            $Values[i] + MFKnapsack(i - 1, j - Weights[i]))$ 
     $V[i, j] \leftarrow value$ 
return  $V[i, j]$ 

```

```

return V[i, j]

```



## *Memory Functions - EXAMPLE*

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity  $W = 5$

		capacity $j$					
	$i$	0	1	2	3	4	5
$w_1 = 2, v_1 = 12$	0	0	0	0	0	0	0
$w_2 = 1, v_2 = 10$	1	0	0	12	12	12	12
$w_3 = 3, v_3 = 20$	2	0	-	12	22	-	22
$w_4 = 2, v_4 = 15$	3	0	-	-	22	-	32
	4	0	-	-	-	-	37



# *Knapsack Problem with Memory Functions – Efficiency*

- ◆ Same as the efficiency of Bottom – Up algorithm except for a constant factor gain.