UNIT-4
Dynamic programming

AVOID CALCULATING SAME STUFF TWICE

COMING UP!!

- Introduction
- The principle of Optimality
- Making Change Problem
- The 0-1 Knapsack Problem
- Shortest Path- Floyd’s Algorithm
- Matrix Chain Multiplication
- Longest Common Subsequence

INTRODUCTION

- Dynamic Programming is a stage-wise search method suitable for optimization problem whose solutions may be viewed as the result of a sequence of decisions.
- The underlying idea of DP is: "*Avoid calculating same stuff twice*", usually be keeping a table of known results of sub problems.
- Thus DP is similar to divide and Conquer but avoids duplicate work when sub problems are identical

Bottom Up Approach for DP

- Bottom Up Means:
  - Start with the smallest subproblems.
  - Combine their solutions and obtain the solution to subproblems of increasing size
  - Until arrive at the solution of the original problem.

Dynamic Programming Vs Divide & Conquer

Divide And Conquer	Dynamic Programming
Divide and Conquer works by dividing the problem into sub-problems, conquer each sub-problem recursively and combine these solutions.	Dynamic Programming is a technique for solving problems with overlapping subproblems. Each sub-problem is solved only once and the result of each sub-problem is stored in a table for future references. These sub-solutions may be used to obtain the original solution and the technique of storing the sub-problem solutions is known as memoization.
Works Best When Sub problems are independent	Works Best When Sub problems are dependent
Example: Merge Sort, Quick Sort, Binary Search..etc	Example: Fibonacci Series, 0-1 Knapsack Problem..etc
Less Complex	More Complex
Top Down Approach	Bottom Up Approach

Greedy v/s. Dynamic Programming

- *Greedy algorithms focus on making the best local choice at* each decision point. In the absence of a correctness proof such greedy algorithms are very likely to fail.
- Dynamic programming gives us a way to design custom algorithms which systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid recomputing (thus providing efficiency)

Greedy Algorithm Vs Dynamic Programming

Comparison:

Dynamic Programming	Greedy Algorithms
• At each step, the choice is determined based on solutions of subproblems.	• At each step, we quickly make a choice that currently looks best. --A local optimal (greedy) choice.
• Sub-problems are solved first.	• Greedy choice can be made first before solving further sub-problems.
• Bottom-up approach	• Top-down approach
• Can be slower, more complex	• Usually faster, simpler

The Principle of Optimality

- Although this principle may appear obvious, it does not apply to every problem we might encounter.

- **When the principle of optimality does not apply, it will probably not be possible to attack the problem in question using dynamic programming.**

- This is the case, for instance, when a problem concerns the optimal use of limited resources.

- Here the optimal solution to an instance may not be obtained by combining the optimal solutions to two or more sub instances, if resources used in these sub solutions add up to more than the total resources available.

The Principle of Optimality

- For example shortest route from Rajkot to Ahemdabad.

- For example finding longest simple route between two cities, using a given set of roads.

The Principle of Optimality

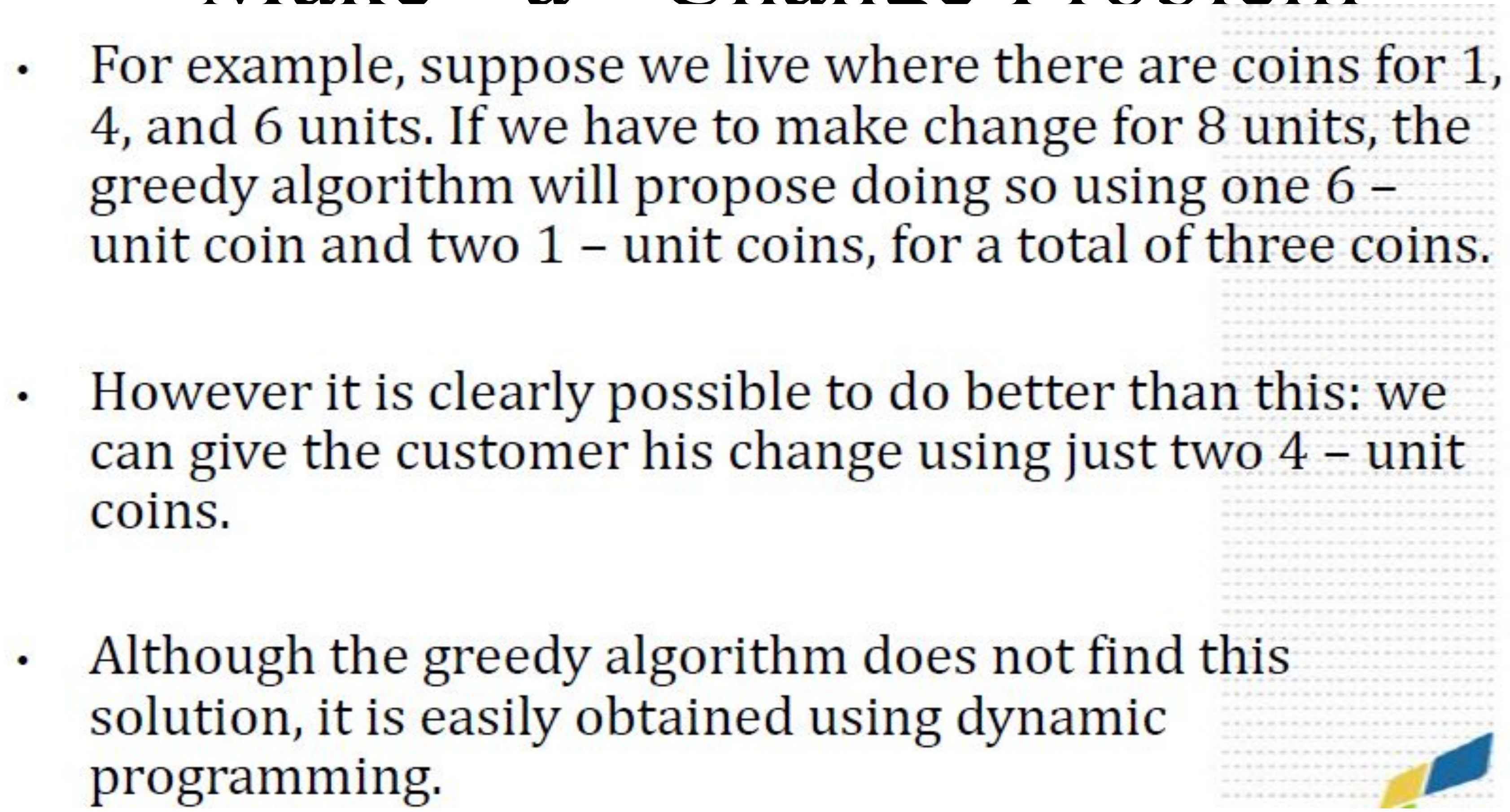
- Nevertheless, the principle of optimality applies more often than not.

- When it does, it can be restated as follows:

  - **The optimal solution to any nontrivial instance of a problem is a combination of optimal solutions to some of its sub instances.**

- The difficulty in turning this principle into an algorithm is that it is not usually obvious which sub instances are relevant to the instance under consideration.

Make - a - Change Problem



- For example, suppose we live where there are coins for 1, 4, and 6 units. If we have to make change for 8 units, the greedy algorithm will propose doing so using one 6 – unit coin and two 1 – unit coins, for a total of three coins.

- However it is clearly possible to do better than this: we can give the customer his change using just two 4 – unit coins.

- Although the greedy algorithm does not find this solution, it is easily obtained using dynamic programming.

Algorithm – Make Change

- Function coins(N)
  {Given the minimum number of coins needed to make change for N units. Array d[1…n] specifies the coinage: in the example there are coins for 1, 4 and 6 units.}

```
array d[1…n] = [1, 4, 6]
array c[1..n, 0..N]
for i ← 1 to n do c[i, 0] ← 0
for i ← 1 to n do
        for j ← 1 to N do
                c[i, j] ← if i = 1 and j < d[i] then +∞
                        else if i = 1 then 1 + c[1, j – d[1]]
                        else if j < d[i] then c[i-1, j]
                        else if min(c[i-1, j], 1+c[I, j – d[i]])

return c[n, N]
```

Formula for Computation

- If $i=1$ and $j<d_i$ then $+\infty$
- If $i=1$ then $c[i,j] = 1 + c[i,j-d_i]$
- If $j<d_i$ then $c[i,j] = c[i-1,j]$
- Otherwise $c[i,j] = \min(c[i-1,j],1+c[i,j-d_i])$

Example: There are 3 coin denominations $d_1 = 1$, $d_2 = 4$, and $d_3 = 6$, and the total amount to make change for is $K = 8$. The following table shows how to compute $C[3,8]$, using the recurrence as a basis but arranging the computation steps in a tabular form (by rows and within the row by columns):

i	Amount	0	1	2	3	4	5	6	7	8
0		0	0	0	0	0	0	0	0	0
1	$d_1 = 1$	0	1	2	3	4	5	6	7	8
2	$d_2 = 4$	0	1	2	3	1	2	3	4	$\boxed{2}$
3	$d_3 = 6$	0	1	$\boxed{2}$	3	1	2	1	2	$\boxed{2}$

$C[2, 8]$

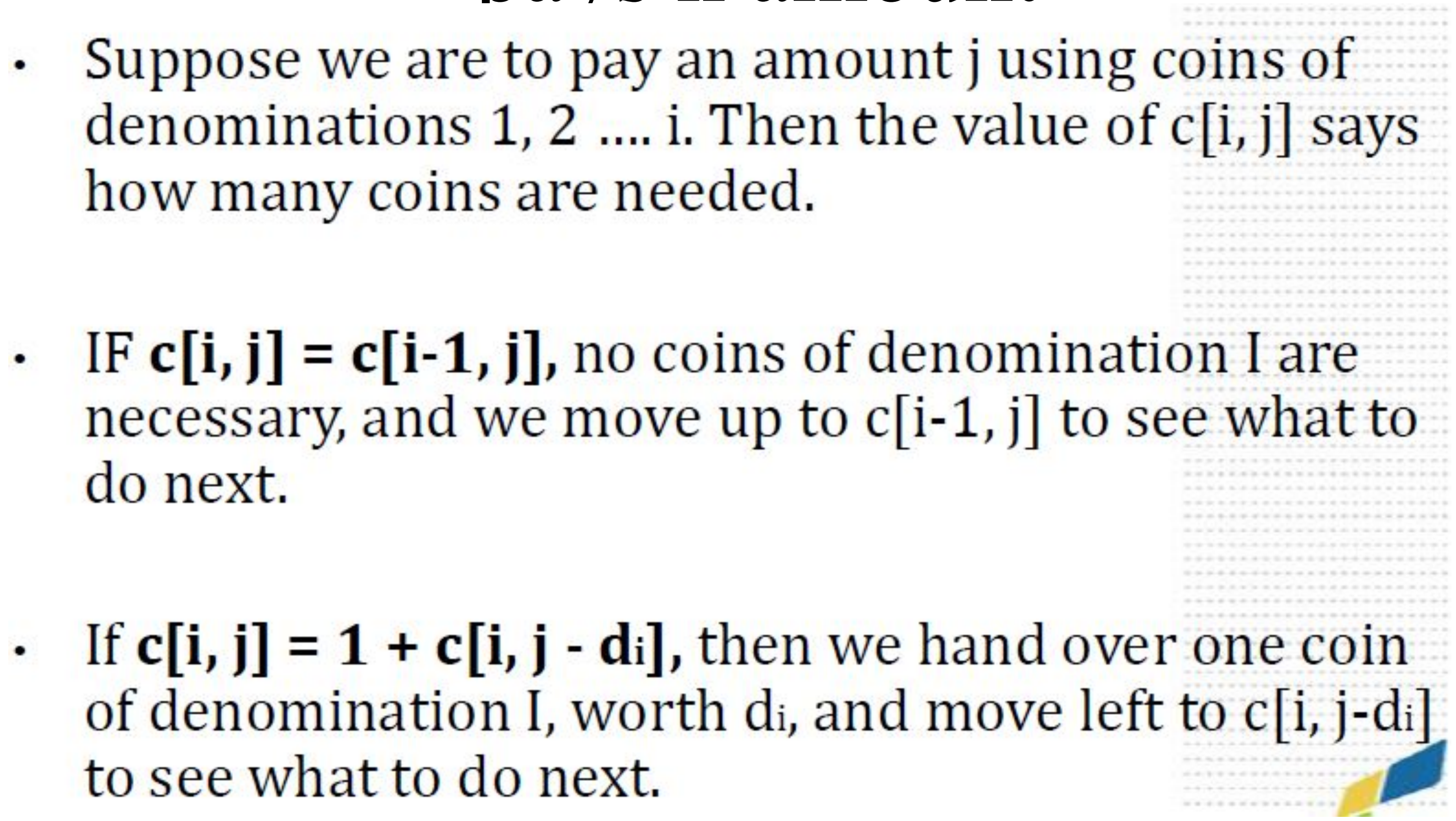
Boundary condition for amount $j = 0$

$C[3, 8 - 6]$

$C[3, 8] = \min(1 + C[3, 8 - 6], C[2, 8])$

Note the time complexity for computing $C[n, K]$ is $O(nK)$, using space $O(K)$ by maintaining last two rows.

Trace Back table to find which coins pays n amount



- Suppose we are to pay an amount j using coins of denominations 1, 2 …. i. Then the value of c[i, j] says how many coins are needed.
- IF **c[i, j] = c[i-1, j],** no coins of denomination I are necessary, and we move up to c[i-1, j] to see what to do next.
- If **c[i, j] = 1 + c[i, j - d_i],** then we hand over one coin of denomination I, worth d_i, and move left to c[i, j-d_i] to see what to do next.

- Solve Making Change problem using Dynamic Programming. (denominations:d1=1,d2=6,d3=10). Give your answer for making change of Rs. 12.

i / j	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
D1=1	0	1	2	3	4	5	6 $\boxed{1}$	7	8	9	10	11	12 $\boxed{2}$
D2=6	0	1	2	3	4	5	1	2	3	4	5	6	2
D3=10	0	1	2	3	4	5	1	2	3	4	1	2	2

Examples to solve

- Solve Making Change problem using Dynamic Programming. (denominations:d1=1,d2=4,d3=5,d4=10). Give your answer for making change of Rs. 8.

- Solve Making Change problem using Dynamic Programming. (denominations:d1=1,d2=5,d3=7,d4=10,d5=25). Give your answer for making change of

Knapsack problem

Given some items, pack the knapsack to get the maximum total value. Each item has some weight and some value. Total weight that we can carry is no more than some fixed number W.
So we must consider weights of items as well as their values.

Item #	Weight	Value
1	1	8
2	3	6
3	5	5

Knapsack problem

There are two versions of the problem:

1. "0-1 knapsack problem"
   - Items are indivisible; you either take an item or not. Some special instances can be solved with *dynamic programming*

2. "Fractional knapsack problem"
   - Items are divisible: you can take any fraction of an item, this can be solved with *greedy programming*

The knapsack problem

Given $W > 0$, and two n-tuples of positive numbers

$$\langle v_1, v_2, \ldots, v_n \rangle \quad \text{and} \quad \langle w_1, w_2 \ldots, w_n \rangle,$$

we wish to determine the subset

$T \subseteq \{1, 2, \ldots, n\}$ (of items to carry) that

$$\text{maximizes} \quad \sum_{i \in T} v_i,$$

$$\text{subject to} \quad \sum_{i \in T} w_i \leq W.$$

The knapsack problem

- Unfortunately the greedy algorithm turns out not to work when xi is required to be 0 or 1.
- For example, **suppose we have three objects available, the first of which weight 6 units and has a value of 8, while the other two weight 5 units each and have a value of 5 each.**
- If the knapsack can **carry 10 units,** then the optimal load includes the two lighter objects for a total value of 10.
- The greedy algorithm, on the other hand, would begin by choosing the object that weighs 6 units, since this is the one with the greatest value per unit weight.
- However if objects cannot be broken the algorithm will be unable to use the remaining capacity in the knapsack. The load it produces therefore consists of just one object with a value of only 8.

The knapsack problem

We construct an array $V[0..n, 0..W]$.

For $1 \leq i \leq n$, and $0 \leq w \leq W$, the entry $V[i, w]$ will store the maximum (combined) value of any subset of items $\{1, 2, \ldots, i\}$ of (combined) weight at most w.

That is

$$V[i,w] = \max\left\{\sum_{j\in T} v_j \ : \ T \subseteq \{1,2,\ldots,i\}, \sum_{j\in T} w_j \leq w\right\}.$$

If we can compute all the entries of this array, then the array entry $V[n, W]$ will contain the solution to our problem.

The knapsack problem

Initial Settings: Set

$$V[0,w]=0 \quad \text{for } 0\le w\le W, \quad \text{no item}$$
$$V[i,w]=-\infty \quad \text{for } w<0, \quad \text{illegal}$$

Recursive Step: Use

$$V[i,w]=\max(V[i-1,w],v_i+V[i-1,w-w_i])$$

for $1\le i\le n, 0\le w\le W$.

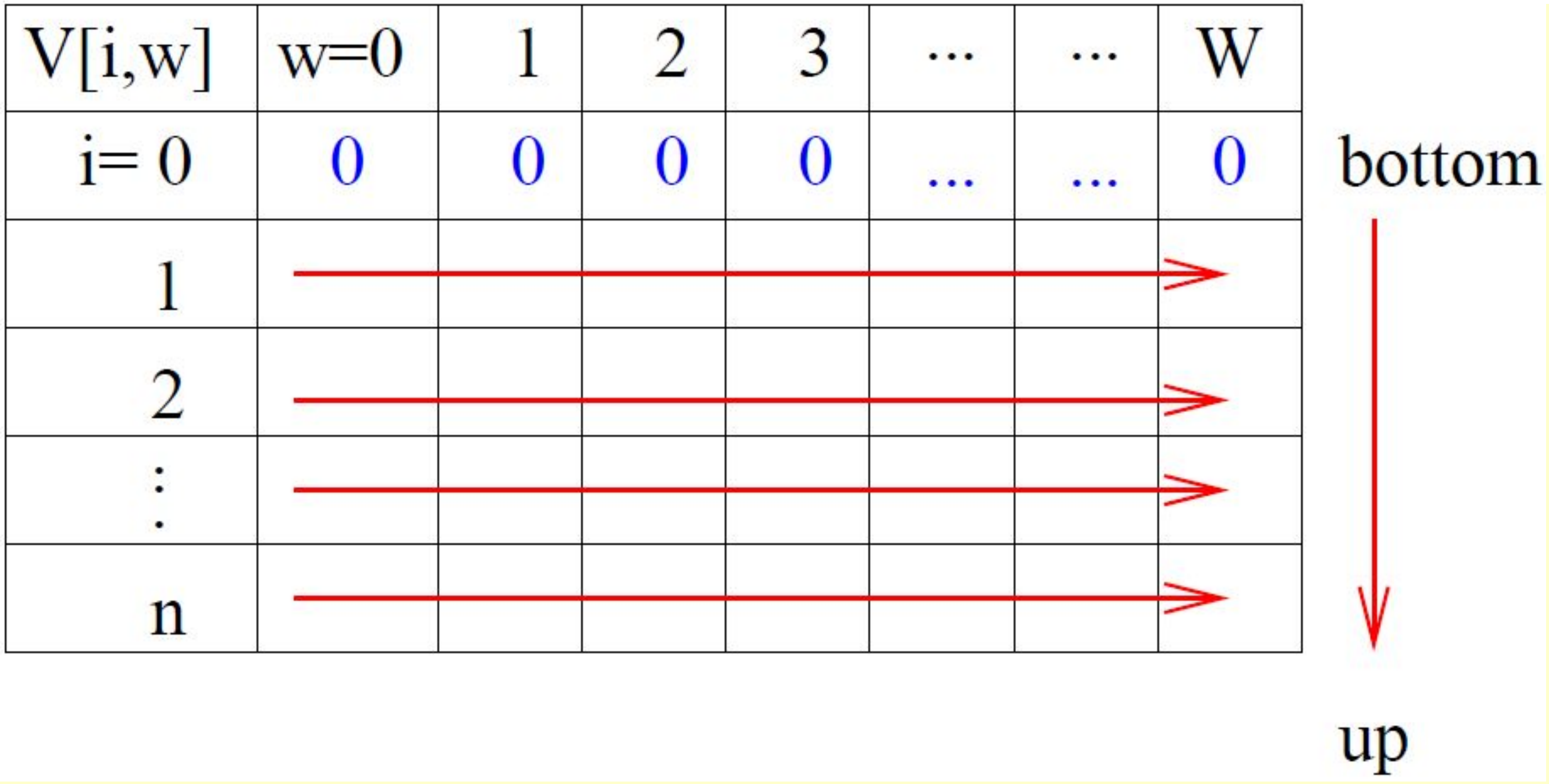
Intuitively, an optimal solution would either choose item i is or not choose item i.

The knapsack problem

Bottom-up computation: Computing the table using

$$V[i,w] = \max(V[i-1,w], v_i + V[i-1, w-w_i])$$

row by row.



V[i,w]	w=0	1	2	3	⋯	⋯	W
i= 0	0	0	0	0	...	...	0
1							
2							
⋮							
n							

The knapsack problem

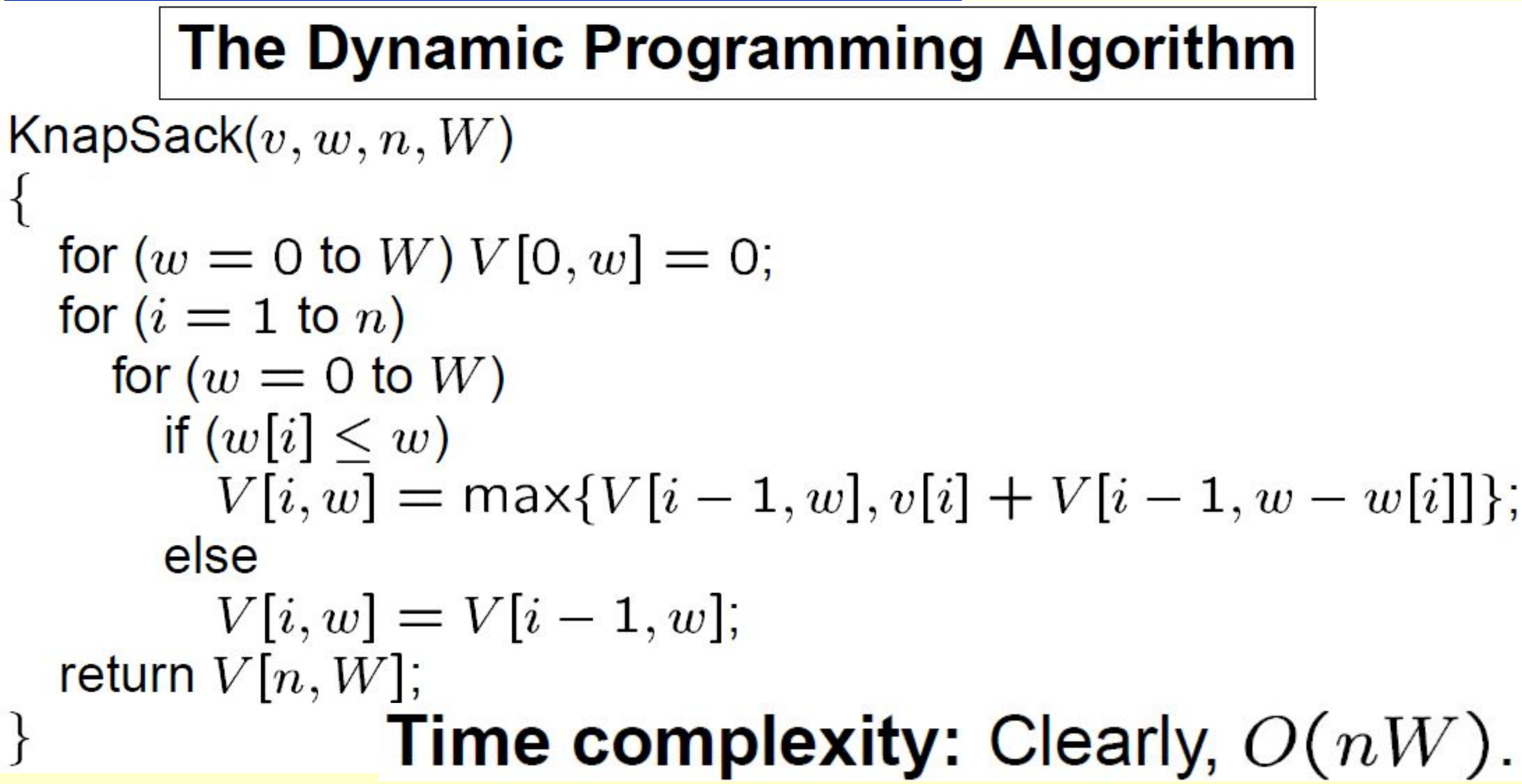
Let $W = 10$ and

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w-w_i])$$

The knapsack problem



The Dynamic Programming Algorithm

```
KnapSack($v, w, n, W$)
{
  for ($w = 0$ to $W$) $V[0, w] = 0$;
  for ($i = 1$ to $n$)
    for ($w = 0$ to $W$)
      if ($w[i] \le w$)
        $V[i, w] = \max\{V[i-1, w], v[i] + V[i-1, w-w[i]]\}$;
      else
        $V[i, w] = V[i-1, w]$;
  return $V[n, W]$;
}
```

Time complexity: Clearly, $O(nW)$.

The problem can be solved using dynamic programming (i.e., a bottom-up approach to carrying out the computation steps) based on a tabular form when the weights are integers.

Example: There are n = 5 objects with integer weights $w[1..5]$ = {1,2,5,6,7}, and values $v[1..5]$ = {1,6,18,22,28}. The following table shows the computations leading to $V[5,11]$ (i.e., assuming a knapsack capacity of 11).

Time: $O(nW)$
space: $O(W)$

Kpsack capacity		0	1	2	3	4	5	6	7	8	9	10	11
wi	*vi*												
1	1	0	1	1	1	1	1	1	1	1	1	1	1
2	6	0	1	6	7	7	7	7	7	7	7	7	7
5	18	0	1	$\boxed{6}$	7	7	18	19	24	$\boxed{25}$	25	25	25
6	22	0	1	6	7	7	18	22	24	$\boxed{28}$	29	29	40
7	28	0	1	6	7	7	18	22	28	29	34	35	40

$V[3, 8]$

$V[4, 8]$ =max ($V[3, 8]$, 22 + $V[3, 2]$)

$V[3,8 - w_4] = V[3, 2]$

Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Example (2)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

```
for w = 0 to W
    V[0,w] = 0
```

Example (3)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

```
for i = 1 to n
    V[i,0] = 0
```

Example (4)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	↓ 0				
2	0					
3	0					
4	0					

i=1

$b_i=3$

$w_i=2$

w=1

$w-w_i = -1$

```
if $w_i <= w$ // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w- w_i]$
    else
        $V[i,w] = V[i-1,w]$
else V[i,w] = V[i-1,w]  // $w_i > w$
```

Example (5)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	**3**			
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w-w_i=0$

```
if $w_i <= w$ // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w-w_i]$
    else
        $V[i,w] = V[i-1,w]$
else $V[i,w] = V[i-1,w]$  // $w_i > w$
```

Example (6)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	**3**		
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

```
if $w_i$ <= w // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w-w_i]$
    else
        $V[i,w] = V[i-1,w]$
else $V[i,w] = V[i-1,w]$  // $w_i > w$
```

Example (7)

Items:

$\boxed{1: (2,3)}$
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	**3**	
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i=2$

```
if $w_i <= w$ // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $\mathbf{V[i,w] = b_i + V[i-1,w- w_i]}$
    else
        $V[i,w] = V[i-1,w]$
else $V[i,w] = V[i-1,w]$  // $w_i > w$
```

Example (8)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	**3**
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=3$

```
if $w_i <= w$ // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w-w_i]$
    else
        $V[i,w] = V[i-1,w]$
else $V[i,w] = V[i-1,w]$  // $w_i > w$
```

Items:

1: (2,3)
2: (3,4)

3: (4,5)

4: (5,6)

Example (9)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	**0**				
3	0					
4	0					

i=2

b_i=4

w_i=3

w=1

$w-w_i$ =-2

```
if $w_i$ <= w // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w- w_i]$
    else
        $V[i,w] = V[i-1,w]$
else **V[i,w] = V[i-1,w]**  // $w_i > w$
```

Example (10)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	$\downarrow$ **3**			
3	0					
4	0					

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

```
if $w_i <= w$ // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w- w_i]$
    else
        $V[i,w] = V[i-1,w]$
else V[i,w] = V[i-1,w]  // $w_i > w$
```

Example (11)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	**4**		
3	0					
4	0					

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

```
if $w_i <= w$ // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w- w_i]$
    else
        $V[i,w] = V[i-1,w]$
else $V[i,w] = V[i-1,w]$  // $w_i > w$
```

Example (12)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	**4**	
3	0					
4	0					

$i=2$
$b_i=4$
$w_i=3$
$w=4$
$w-w_i=1$

```
if $w_i <= w$ // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w-w_i]$
    else
        $V[i,w] = V[i-1,w]$
else $V[i,w] = V[i-1,w]$  // $w_i > w$
```

Example (13)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w-w_i=2$

```
if $w_i$ <= w // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w-w_i]$
    else
        $V[i,w] = V[i-1,w]$
else $V[i,w] = V[i-1,w]$  // $w_i > w$
```

Example (14)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	**0**	**3**	**4**		
4	0					

$i=3$

$b_i=5$

$w_i=4$

$w= 1..3$

```
if $w_i$ <= w // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w- w_i]$
    else
        $V[i,w] = V[i-1,w]$
else $\mathbf{V[i,w] = V[i-1,w]}$  // $w_i > w$
```

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Example (15)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	**5**	
4	0					

$i=3$

$b_i=5$

$w_i=4$

$w= 4$

$w- w_i=0$

```
if $w_i <= w$ // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w- w_i]$
    else
        $V[i,w] = V[i-1,w]$
else $V[i,w] = V[i-1,w]$  // $w_i > w$
```

Example (16)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	$\downarrow$ **7**
4	0					

i=3

b_i=5

w_i=4

w= 5

w- w_i=1

```
if $w_i$ <= w // item i can be part of the solution
    if $b_i$ + V[i-1,w-$w_i$] > V[i-1,w]
        V[i,w] = $b_i$ + V[i-1,w- $w_i$]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w]  // $w_i$ > w
```

Example (17)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	$\downarrow$ **0**	$\downarrow$ **3**	$\downarrow$ **4**	$\downarrow$ **5**	

i=4

b_i=6

w_i=5

w= 1..4

```
if $w_i$ <= w // item i can be part of the solution
    if $b_i$ + V[i-1,w-$w_i$] > V[i-1,w]
        V[i,w] = $b_i$ + V[i-1,w- $w_i$]
    else
        V[i,w] = V[i-1,w]
else **V[i,w] = V[i-1,w]** // $w_i$ > w
```

Example (18)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	**7**

$i=4$

$b_i=6$

$w_i=5$

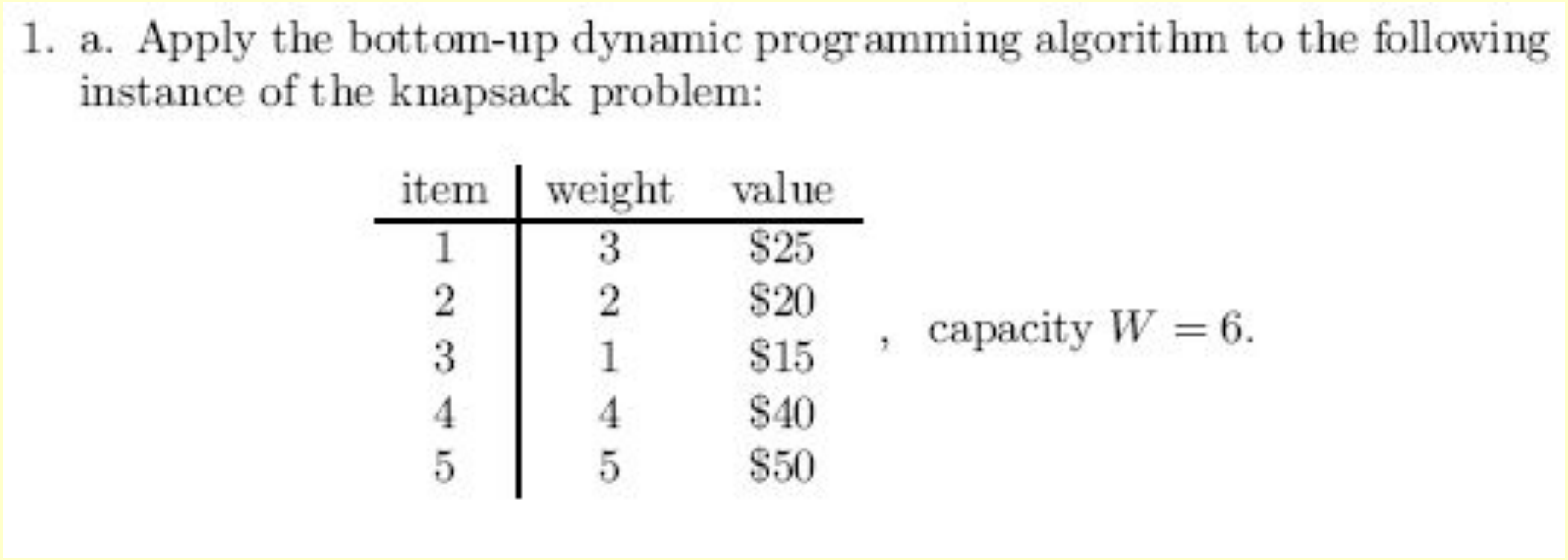
$w= 5$

$w- w_i=0$

```
if $w_i$ <= w // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w- w_i]$
    else
        **V[i,w] = V[i-1,w]**
else V[i,w] = V[i-1,w]  // $w_i > w$
```

Exercise

- P303 8.2.1 (a).



1. a. Apply the bottom-up dynamic programming algorithm to the following instance of the knapsack problem:

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

, capacity $W = 6$.

- How to find out which items are in the optimal subset?

Example 4

Let $W = 10$ and

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

Example 5

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

Knapsack of capacity

$W = 5$

capacity *j* □

		0	1	2	3	4	5
	0						
$w_1 = 2, v_1 = 12$	1						
$w_2 = 1, v_2 = 10$	2						
$w_3 = 3, v_3 = 20$	3						
$w_4 = 2, v_4 = 15$	4						

Example 6

- There are n = 5 objects with integer weights $w[1..5]$ = {1,2,5,6,7}, and values $v[1..5]$ = {1,6,18,22,28}. The following table shows the computations leading to $V[5,11]$ (i.e., assuming a knapsack capacity of 11).

Time: $O(nW)$
space: $O(W)$

Knapsack capacity		0	1	2	3	4	5	6	7	8	9	10	11
wi	*vi*	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	1	1	1	1	1	1	1	1	1	1
2	6	0	1	6	7	7	7	7	7	7	7	7	7
5	18	0	1	$\boxed{6}$	7	7	18	19	24	$\boxed{25}$	25	25	25
6	22	0	1	6	7	7	18	22	24	$\boxed{28}$	29	29	40
7	28	0	1	6	7	7	18	22	28	29	34	35	40

$V[3, 8]$

$V[4, 8] = \max (V[3, 8], 22 + V[3, 2])$

$V[3,8 - w_4] = V[3, 2]$

Matrix-chain Multiplication

- Suppose we have a sequence or chain A_1, A_2, …, A_n of *n* matrices to be multiplied
  - That is, we want to compute the product $A_1A_2…A_n$

- There are many possible ways (parenthesizations) to compute the product

Matrix-chain Multiplication …contd

- Example: consider the chain A_1, A_2, A_3, A_4 of 4 matrices
  - Let us compute the product $A_1A_2A_3A_4$
- There are 5 possible ways:

1. $(A_1(A_2(A_3A_4)))$
2. $(A_1((A_2A_3)A_4))$
3. $((A_1A_2)(A_3A_4))$
4. $((A_1(A_2A_3))A_4)$
5. $(((A_1A_2)A_3)A_4)$

Matrix-chain Multiplication …contd

- To compute the number of scalar multiplications necessary, we must know:
  - Algorithm to multiply two matrices
  - Matrix dimensions

- Can you write the algorithm to multiply two matrices?

Algorithm to Multiply 2 Matrices

Input: Matrices $A_{p \times q}$ and $B_{q \times r}$ (with dimensions $p \times q$ and $q \times r$)

Result: Matrix $C_{p \times r}$ resulting from the product $A \cdot B$

```
MATRIX-MULTIPLY($A_{p \times q}$, $B_{q \times r}$)
1.  for $i \leftarrow 1$ to $p$
2.      for $j \leftarrow 1$ to $r$
3.          $C[i, j] \leftarrow 0$
4.          for $k \leftarrow 1$ to $q$
5.              $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
6.  return $C$
```

Scalar multiplication in line 5 dominates time to compute C

Number of scalar multiplications = pqr

Matrix-chain Multiplication

…contd

- Example: Consider three matrices $A_{10\times100}$, $B_{100\times5}$, and $C_{5\times50}$
- There are 2 ways to parenthesize
  - $((AB)C) = D_{10\times5} \cdot C_{5\times50}$
    - $AB \Rightarrow 10\cdot100\cdot5=5{,}000$ scalar multiplications
    - $DC \Rightarrow 10\cdot5\cdot50 =2{,}500$ scalar multiplications

Total: 7,500

  - $(A(BC)) = A_{10\times100} \cdot E_{100\times50}$
    - $BC \Rightarrow 100\cdot5\cdot50=25{,}000$ scalar multiplications
    - $AE \Rightarrow 10\cdot100\cdot50 =50{,}000$ scalar multiplications

Matrix-chain Multiplication …contd

- Matrix-chain multiplication problem
  - Given a chain $A_1, A_2, \ldots, A_n$ of n matrices, where for i=1, 2, …, n, matrix A_i has dimension $p_{i-1} \times p_i$
  - Parenthesize the product $A_1 A_2 \ldots A_n$ such that the total number of scalar multiplications is minimized
- Brute force method of exhaustive search takes time exponential in n

Dynamic Programming Approach

- The structure of an optimal solution
  - Let us use the notation $A_{i..j}$ for the matrix that results from the product $A_i A_{i+1} \ldots A_j$
  - An optimal parenthesization of the product $A_1A_2\ldots A_n$ splits the product between A_k and A_{k+1} for some integer k where $1 \le k < n$
  - First compute matrices $A_{1..k}$ and $A_{k+1..n}$; then multiply them to get the final matrix $A_{1..n}$

Dynamic Programming Approach

…contd

- **Key observation**: parenthesizations of the subchains $A_1A_2 \ldots A_k$ and $A_{k+1}A_{k+2} \ldots A_n$ must also be optimal if the parenthesization of the chain $A_1A_2 \ldots A_n$ is optimal (why?)

- That is, the optimal solution to the problem contains within it the optimal solution to subproblems

Dynamic Programming Approach

…contd

- Recursive definition of the value of an optimal solution
  - Let $m[i, j]$ be the minimum number of scalar multiplications necessary to compute $A_{i..j}$
  - Minimum cost to compute $A_{1..n}$ is $m[1, n]$
  - Suppose the optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1} for some integer k where $i \leq k < j$

Dynamic Programming Approach

…contd

- $A_{i..j} = (A_i A_{i+1} \ldots A_k) \cdot (A_{k+1} A_{k+2} \ldots A_j) = A_{i..k} \cdot A_{k+1..j}$
- Cost of computing $A_{i..j}$ = cost of computing $A_{i..k}$ + cost of computing $A_{k+1..j}$ + cost of multiplying $A_{i..k}$ and $A_{k+1..j}$
- Cost of multiplying $A_{i..k}$ and $A_{k+1..j}$ is $p_{i-1} p_k p_j$

- $m[i, j\ ] = m[i, k] + m[k+1, j\ ] + p_{i-1} p_k p_j$ *for* $i \leq k < j$
- $m[i, j\ ] = 0$ for i=1,2,…,n

Dynamic Programming Approach

…contd

- But… optimal parenthesization occurs at one value of k among all possible $i \le k < j$
- Check all these and select the best one

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

Dynamic Programming Approach

…contd

- To keep track of how to construct an optimal solution, we use a table *s*
- $s[i, j\,]$ = value of *k* at which $A_i\, A_{i+1} \ldots A_j$ is split for optimal parenthesization
- Algorithm: next slide
  - First computes costs for chains of length l=1
  - Then for chains of length l=2,3, … and so on
  - Computes the optimal cost bottom-up

Algorithm to Compute Optimal Cost

Input: Array $p[0 \ldots n]$ containing matrix dimensions and n

Result: Minimum-cost table m and split table s

```
MATRIX-CHAIN-ORDER(p[ ], n)
    for $i \leftarrow 1$ to $n$
        $m[i, i] \leftarrow 0$
    for $l \leftarrow 2$ to $n$
        for $i \leftarrow 1$ to $n-l+1$
            $j \leftarrow i+l-1$
            $m[i, j] \leftarrow \infty$
            for $k \leftarrow i$ to $j-1$
                $q \leftarrow m[i, k] + m[k+1, j] + p[i-1]\, p[k]\, p[j]$
                if $q < m[i, j]$
                    $m[i, j] \leftarrow q$
                    $s[i, j] \leftarrow k$
    return $m$ and $s$
```

Takes $O(n^3)$ time

Requires $O(n^2)$ space

Constructing Optimal Solution

- Our algorithm computes the minimum-cost table *m* and the split table *s*
- The optimal solution can be constructed from the split table *s*
  - Each entry $s[i, j\ ]=k$ shows where to split the product $A_i\ A_{i+1}\ \ldots\ A_j$ for the minimum cost

Example

- Show how to multiply this matrix chain optimally

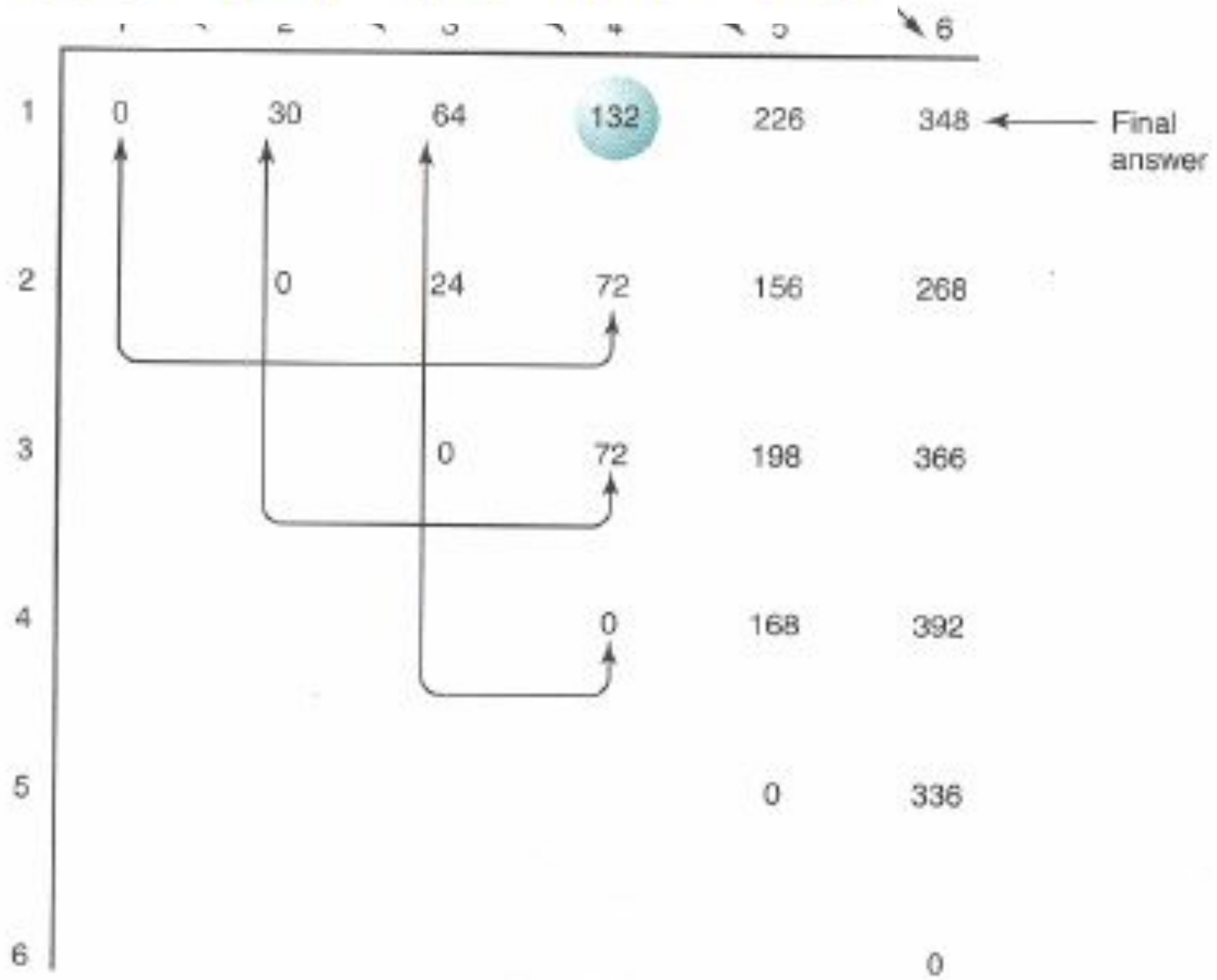
Matrix	Dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

- Solution on the board
  - Minimum cost 15,125
  - Optimal parenthesization $((A_1(A_2A_3))((A_4 A_5)A_6))$

Example

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

$5 \times 2 \quad 2 \times 3 \quad 3 \times 4 \quad 4 \times 6 \quad 6 \times 7 \quad 7 \times 8$



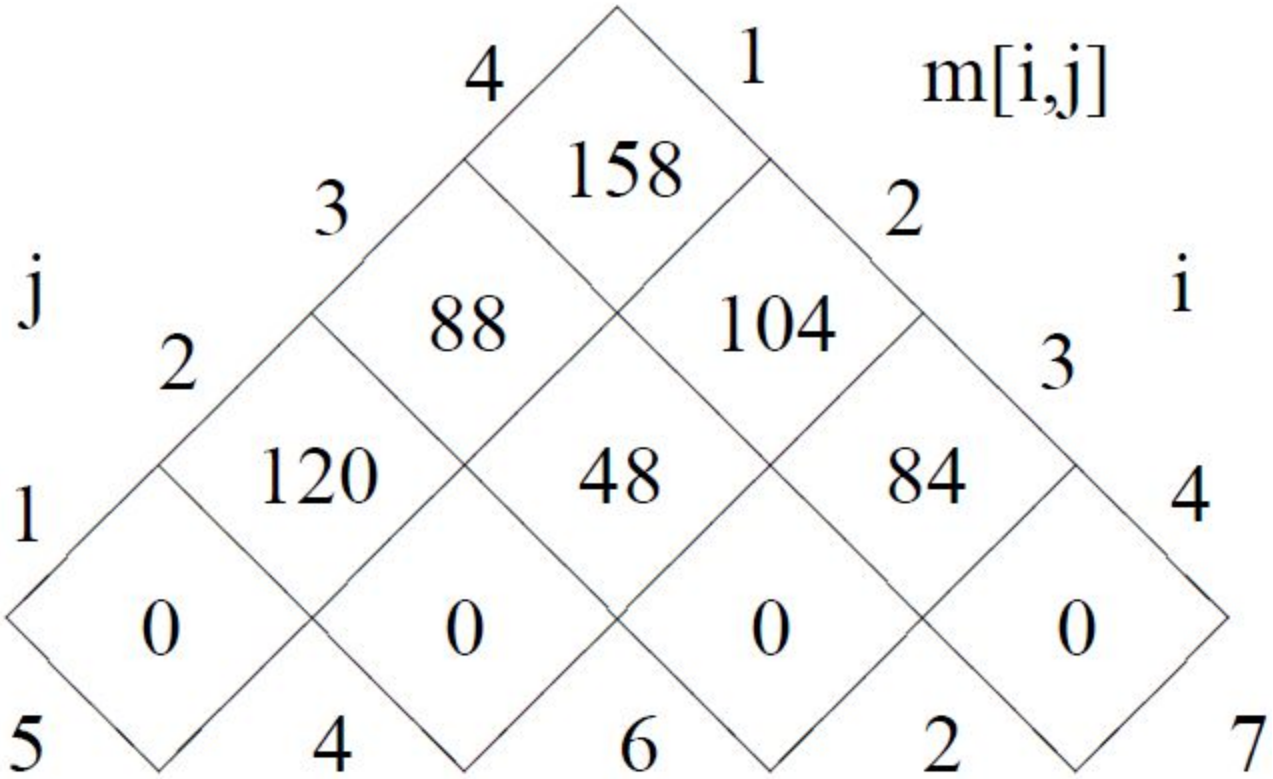
	1	2	3	4	5	6
1	0	30	64	132	226	348
2		0	24	72	156	268
3			0	72	198	366
4				0	168	392
5					0	336
6						0

Final answer

$$
\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & & 1 & 1 & 1 & 1 & 1 \\
2 & & & 2 & 3 & 4 & 5 \\
3 & & & & 3 & 4 & 5 \\
4 & & & & & 4 & 5 \\
5 & & & & & & 5
\end{array}
$$

$$(A_1(A_2A_3))((A_4A_5)A_6).$$

Example: Given a chain of four matrices A_1, A_2, A_3 and A_4, with $p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$. Find $m[1, 4]$.



K[I,j

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

Subsequence

- A **subsequence** is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements.

- **Examples:**
  LCS for "ABCDGH" and "AEDFHR" is "ADH" of length 3.
  LCS for "AGGTAB" and "GXTXAYB" is "GTAB" of length 4.

Longest-common-subsequence problem:

- We are given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ and wish to find a maximum length common subsequence of X and Y.

- We Define i_{th} prefix of X,
  - $X_i = \langle x_1, x_2, \ldots, x_i \rangle$.

A recursive solution to subproblem

- Define $c\,[i, j]$ is the length of the LCS of X_i and Y_j.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 , \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j , \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j . \end{cases}$$

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)
1. m = length(X)  // get the # of symbols in X
2. n  = length(Y) // get the # of symbols in Y
3. for i = 1 to m  c[i,0] = 0   // special case: $Y_0$
4. for j = 1 to n   c[0,j] = 0   // special case: $X_0$
5. for i = 1 to m                // for all $X_i$
6.  for j = 1 to n               // for all $Y_j$
7.      if ( $X_i$ == $Y_j$ )
8.          c[i,j] = c[i-1,j-1] + 1
9.      else c[i,j] = max( c[i-1,j], c[i,j-1] )
10. return c
```

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**

LCS Example (0)

ABCB
BDCAB

	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]

LCS Example (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**					
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
```

LCS Example (2)

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

LCS Example (3)

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**		
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

```
if ( $X_i == Y_j$ )
    $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max( c[i-1,j], c[i,j-1] )$
```

LCS Example (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	↑ 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 $c[i,j] = \max( c[i-1,j], c[i,j-1] )$
```

LCS Example (5)

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**					
3	**C**	**0**					
4	**B**	**0**					

```
if ( $X_i$ == $Y_j$ )
    $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max( c[i-1,j], c[i,j-1] )$
```

LCS Example (6)

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

LCS Example (7)

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

LCS Example (8)

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					
4	B	0					

```
if ( $X_i == Y_j$ )
    $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max( c[i-1,j], c[i,j-1] )$
```

LCS Example (10)

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

LCS Example (11)

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

LCS Example (12)

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

LCS Example (13)

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])

LCS Example (14)

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] )
```

LCS Example (15)

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)

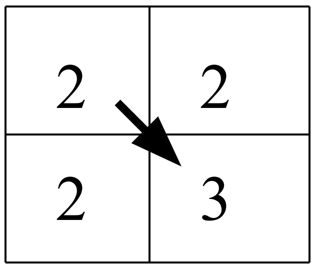
since each c[i,j] is calculated in constant time, and there are m*n elements in the array

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:



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

Tracing Back The Algorithm To find Subsequence

```
LCS-LENGTH(X, Y)
$m = X.length$
$n = Y.length$
let $b[1..m, 1..n]$ and $c[0..m, 0..n]$ be new tables
for $i = 1$ to $m$
    $c[i, 0] = 0$
for $j = 0$ to $n$
    $c[0, j] = 0$
for $i = 1$ to $m$
    for $j = 1$ to $n$
        if $x_i == y_j$
            $c[i, j] = c[i-1, j-1] + 1$
            $b[i, j] = $ "$\nwarrow$"
        elseif $c[i-1, j] \geq c[i, j-1]$
            $c[i, j] = c[i-1, j]$
            $b[i, j] = $ "$\uparrow$"
        else $c[i, j] = c[i, j-1]$
            $b[i, j] = $ "$\leftarrow$"
return $c$ and $b$
```

```
PRINT-LCS($b, X, i, j$)
1  if $i == 0$ or $j == 0$
2      return
3  if $b[i, j] ==$ "$\nwarrow$"
4      PRINT-LCS($b, X, i - 1, j - 1$)
5      print $x_i$
6  elseif $b[i, j] ==$ "$\uparrow$"
7      PRINT-LCS($b, X, i - 1, j$)
8  else PRINT-LCS($b, X, i, j - 1$)
```

How to find actual LCS - continued

- Remember that

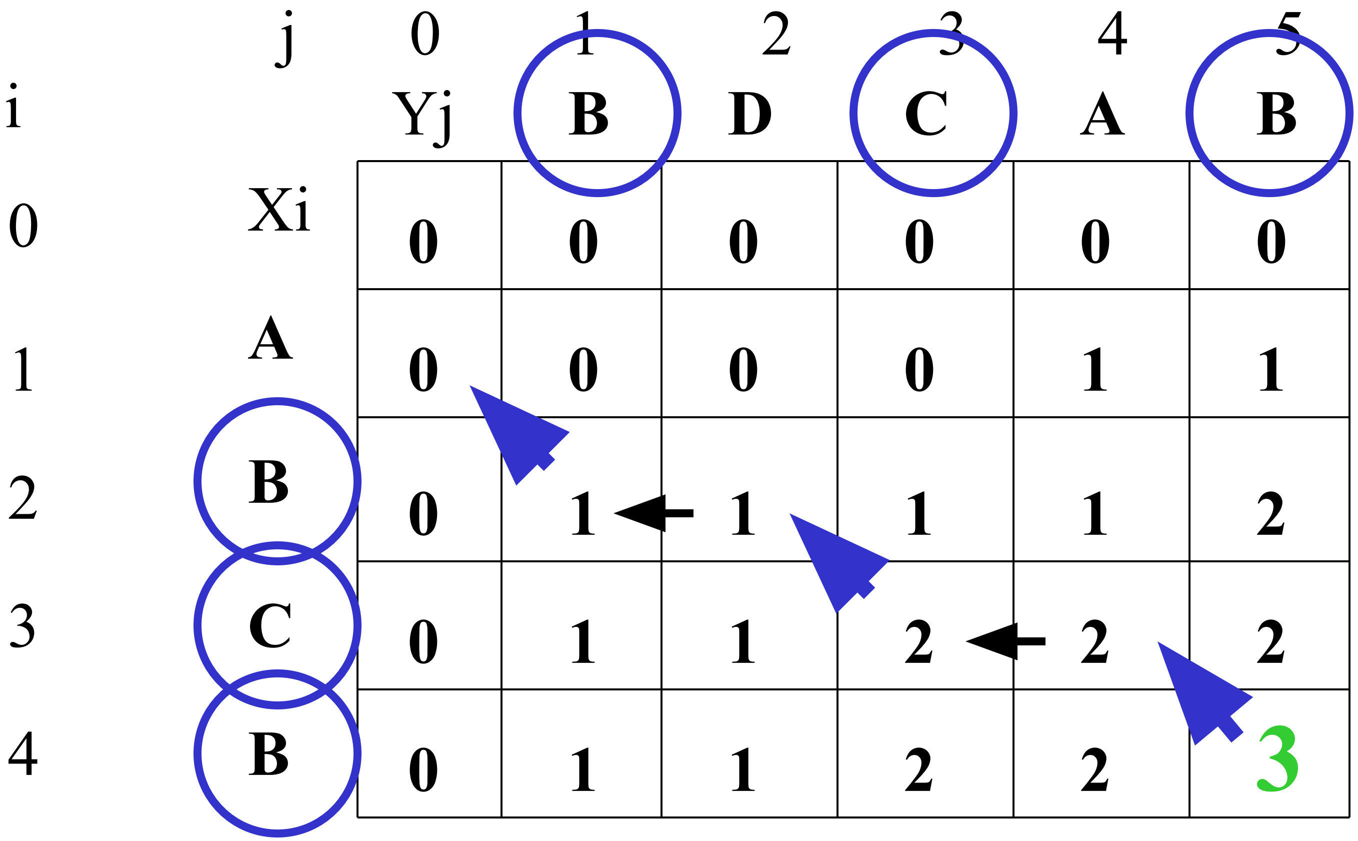
$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- So we can start from *c[m,n]* and go backwards
- Whenever *c[i,j] = c[i-1, j-1]+1*, 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

Finding LCS

	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**

Finding LCS (2)



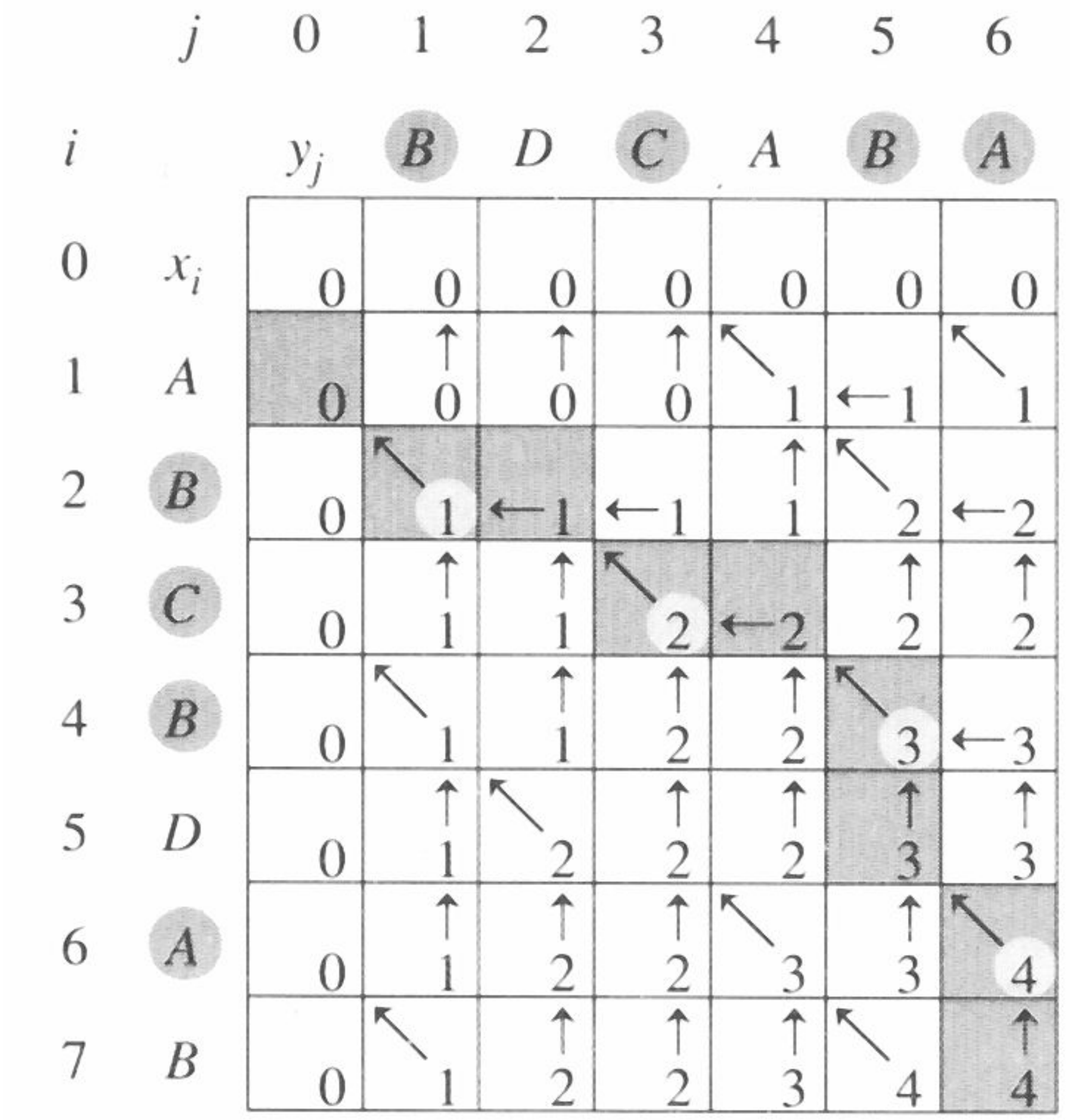
	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	↖ **1**	← **1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	↖ **2**	← **2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	↖ **3**

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

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

(this string turned out to be a palindrome)

X[i]= “ABCBDAB”
Y[j]= “BDCABA”
Find LCS.



j		0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

X=PRESIDENT
Y=PROVIDENCE

i	*j* *0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
		p	*r*	*o*	*v*	*i*	*d*	*e*	*n*	*c*	*e*
0	0	0	0	0	0	0	0	0	0	0	0
1 *p*	0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2 *r*	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
3 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4 *s*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3
5 *i*	0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3	↑ 3	↑ 3	↑ 3	↑ 3
6 *d*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4	← 4	← 4	← 4	← 4
7 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	↖ 5
8 *n*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6
9 *t*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6

i	*j* *0*	*1* *p*	*2* *r*	*3* *o*	*4* *v*	*5* *i*	*6* *d*	*7* *e*	*8* *n*	*9* *c*	*10* *e*
0	**0**	0	0	0	0	0	0	0	0	0	0
1 *p*	0	**↖ 1**	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2 *r*	0	↑ 1	**↖ 2**	**← 2**	**← 2**	← 2	← 2	← 2	← 2	← 2	← 2
3 *e*	0	↑ 1	↑ 2	↑ 2	**↑ 2**	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4 *s*	0	↑ 1	↑ 2	↑ 2	**↑ 2**	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3
5 *i*	0	↑ 1	↑ 2	↑ 2	↑ 2	**↖ 3**	← 3	↑ 3	↑ 3	↑ 3	↑ 3
6 *d*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	**↖ 4**	← 4	← 4	← 4	← 4
7 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	**↖ 5**	← 5	← 5	↖ 5
8 *n*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	**↖ 6**	**← 6**	**← 6**
9 *t*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	**↑ 6**

Output: *priden*

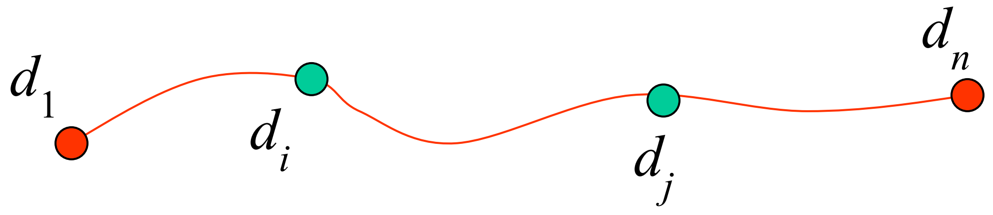
More Examples

1. X=ALLIGNMENT V/S Y=ASSIGNMENT
2. X=abbacdcba V/S Y=bcdbbcaac
3. X=XYZYTXY V/S Y=YTZXYX

Principle of Optimality:

In solving optimization problems which require making a sequence of decisions, such as the change making problem, we often apply the following principle in setting up a recursive algorithm: any subsequence of an optimal solution constitutes an optimal sequence of decisions for the corresponding subproblem. This is known as the principle of optimality which can be illustrated by the shortest paths in weighted graphs as follows:

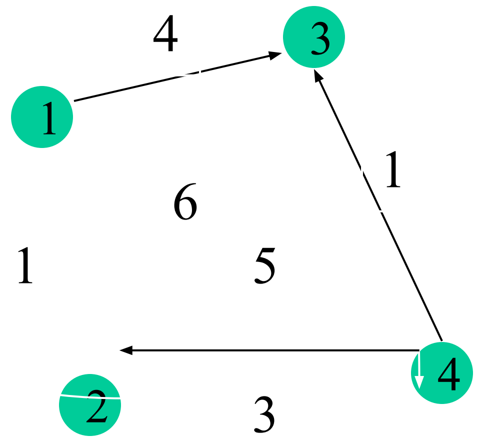
shortest path from d_1 to d_n



Problem: In a weighted (di)graph, find shortest paths between every pair of vertices

Same idea: construct solution through series of matrices $D^{(0)}$, …, $D^{(n)}$ using increasing subsets of the vertices allowed as intermediate

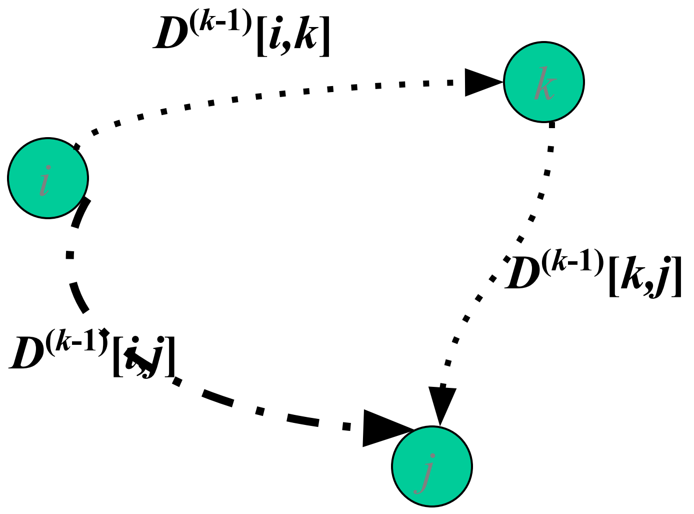
Example:



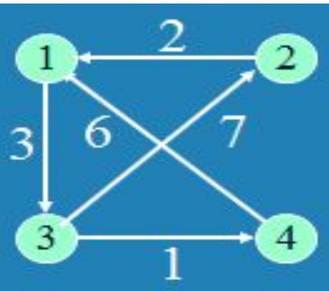
$$\begin{matrix} 0 & \infty & 4 & \infty \\ 1 & 0 & 6 & 3 \\ \infty & \infty & 0 & \infty \\ 6 & 5 & 1 & 0 \end{matrix}$$

On the k-th iteration, the algorithm determines shortest paths between every pair of vertices i, j that use only vertices among 1,…,k as intermediate

$$D^{(k)}[i,j] = \min\{D^{(k-1)}[i,j],\ D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$$



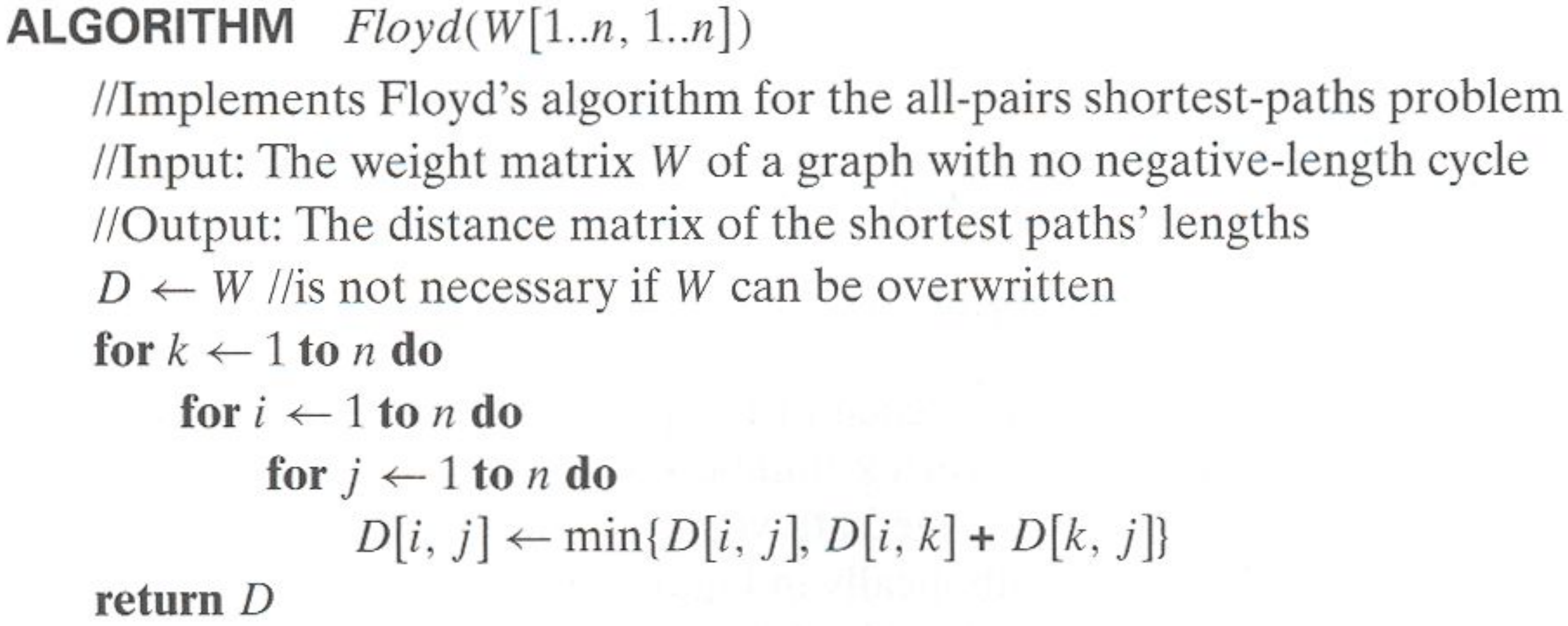
Example



$$D^{(0)} = \begin{array}{cccc} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{array} \qquad D^{(1)} = \begin{array}{cccc} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{array}$$

$$D^{(2)} = \begin{array}{cccc} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{array} \qquad D^{(3)} = \begin{array}{cccc} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{array} \qquad D^{(4)} = \begin{array}{cccc} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{array}$$

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 ← W //is not necessary if W can be overwritten
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
    return D
```

Dynamic Programming

- Indications: optimal substructure, repeated subproblems
- ***What is the difference between memoization and dynamic programming?***
- A: same basic idea, but:
  - ***Memoization***: recursive algorithm, looking up subproblem solutions after computing once
  - ***Dynamic programming***: build table of subproblem solutions bottom-up

Conclusion

- Dynamic programming is a useful technique of solving certain kind of problems
- When the solution can be recursively described in terms of partial solutions, we can store these partial solutions and re-use them as necessary
- Running time (Dynamic Programming algorithm vs. naïve algorithm):
  - LCS: **$O(m*n)$** vs. **$O(n * 2^m)$**
  - 0-1 Knapsack problem: **$O(W*n)$** vs. **$O(2^n)$**