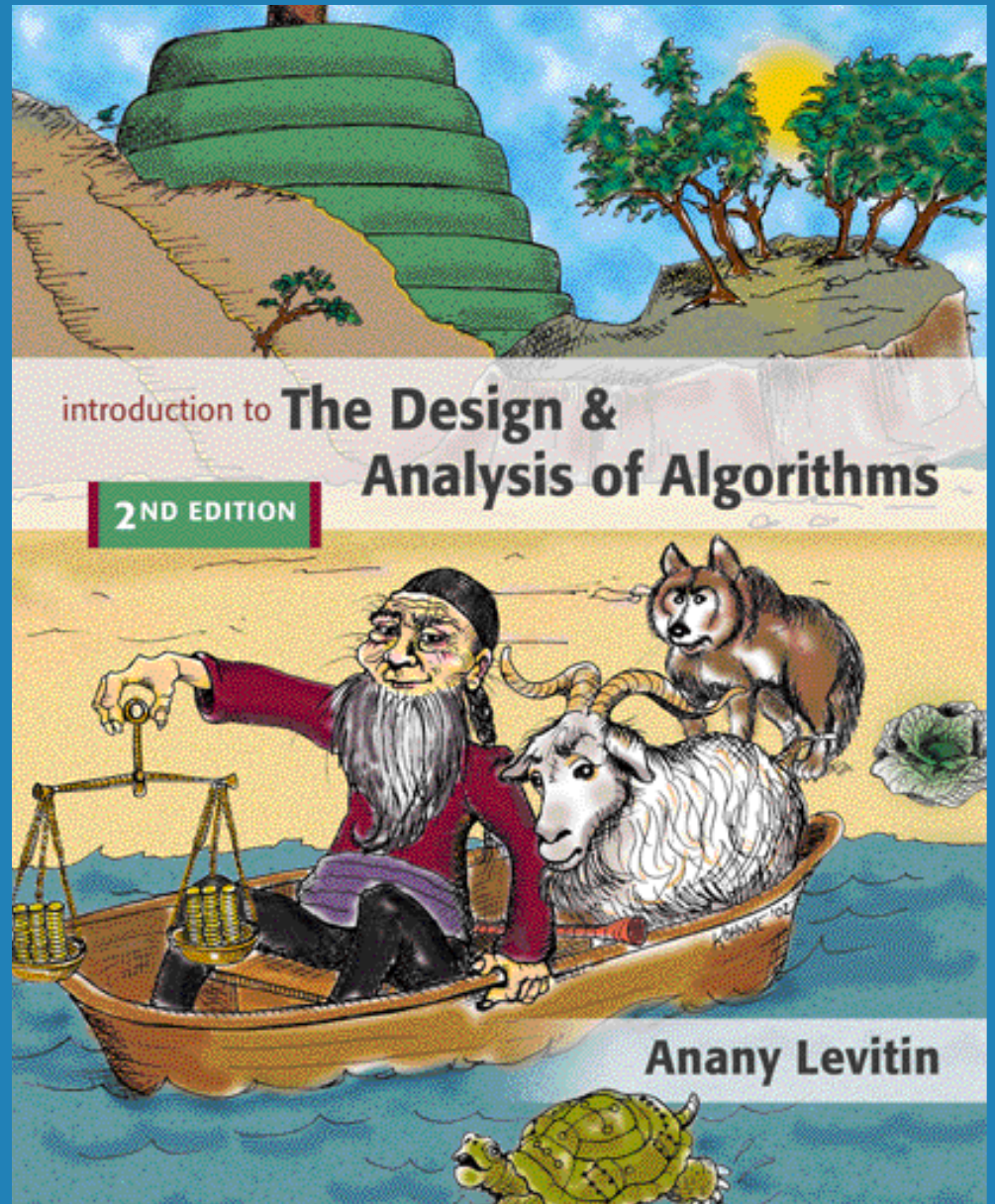
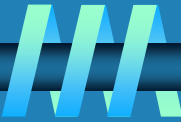


Chapter 8

Dynamic Programming



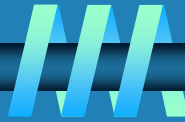
Dynamic Programming



Dynamic Programming is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping subinstances

- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and
- “Programming” here means “planning”
- Main idea:
 - set up a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the same type
 - solve smaller instances once
 - record solutions in a table
 - extract solution to the initial instance from that table

Example: Fibonacci numbers



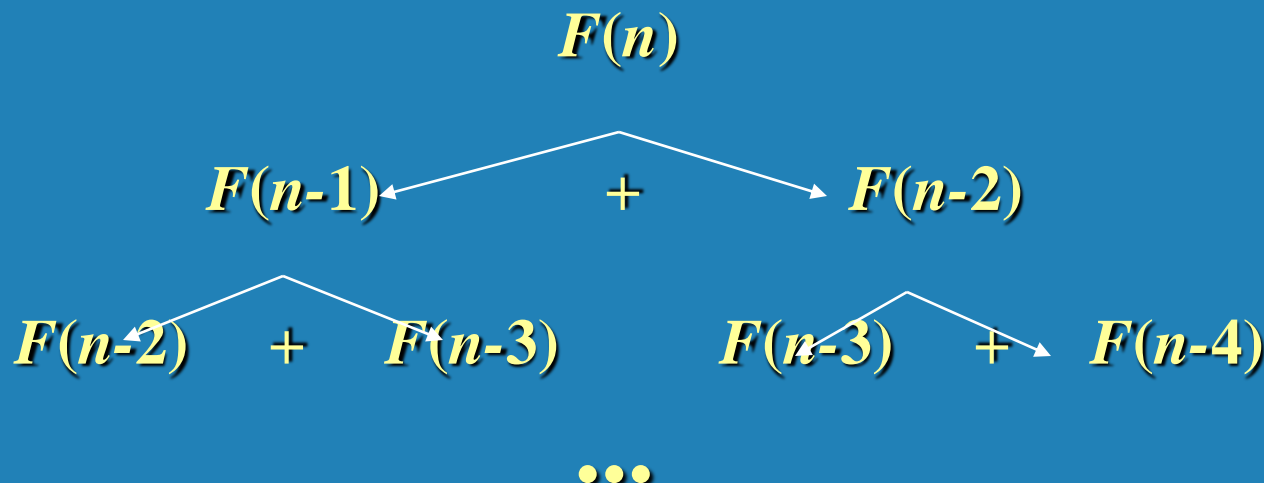
- Recall definition of Fibonacci numbers:

$$F(n) = F(n-1) + F(n-2)$$

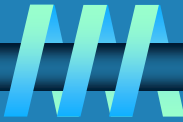
$$F(0) = 0$$

$$F(1) = 1$$

- Computing the n^{th} Fibonacci number recursively (top-down):



Example: Fibonacci numbers (cont.)



Computing the n^{th} Fibonacci number using bottom-up iteration and recording results:

$$F(0) = 0$$

$$F(1) = 1$$

$$F(2) = 1 + 0 = 1$$

...

$$F(n-2) =$$

$$F(n-1) =$$

$$F(n) = F(n-1) + F(n-2)$$

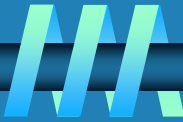
0	1	1	. . .	$F(n-2)$	$F(n-1)$	$F(n)$
---	---	---	-------	----------	----------	--------

Efficiency:

- time n

- space n

Examples of DP algorithms



- **Computing a binomial coefficient**
- **Warshall's algorithm for transitive closure**
- **Floyd's algorithm for all-pairs shortest paths**
- **Some instances of difficult discrete optimization problems:**
 - **traveling salesman**
 - **knapsack**



Computing a binomial coefficient by DP

Computing a binomial coefficient is a standard example of applying dynamic programming to a nonoptimization problem. You may recall from your studies of elementary combinatorics that the *binomial coefficient*, denoted $C(n, k)$ or $\binom{n}{k}$, is the number of combinations (subsets) of k elements from an n -element set ($0 \leq k \leq n$). The name “binomial coefficients” comes from the participation of these numbers in the binomial formula:

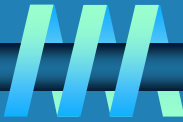
$$(a + b)^n = C(n, 0)a^n + \dots + C(n, k)a^{n-k}b^k + \dots + C(n, n)b^n.$$

Of the numerous properties of binomial coefficients, we concentrate on two:

$$C(n, k) = C(n - 1, k - 1) + C(n - 1, k) \quad \text{for } n > k > 0 \quad (8.3)$$

and

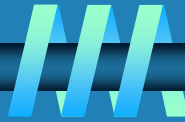
$$C(n, 0) = C(n, n) = 1. \quad (8.4)$$



Value of $C(n,k)$ can be computed by filling a table:

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

Computing $C(n, k)$: pseudocode and analysis



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** $j = i$

$C[i, j] \leftarrow 1$

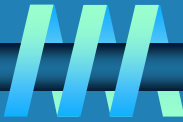
else $C[i, j] \leftarrow C[i - 1, j - 1] + C[i - 1, j]$

return $C[n, k]$

Time efficiency: $\Theta(nk)$

What is the time efficiency of this algorithm? Clearly, the algorithm's basic operation is addition, so let $A(n, k)$ be the total number of additions made by this algorithm in computing $C(n, k)$. Note that computing each entry by formula (8.3) requires just one addition. Also note that because the first $k + 1$ rows of the table form a triangle while the remaining $n - k$ rows form a rectangle, we have to split the sum expressing $A(n, k)$ into two parts:

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



	0	1	2	3
0	1			
1	1	1		
2	1	2	1	
3	1	3	3	1
4	1	4	6	4
5	1	5	10	10
6	1	6	15	20

Warshall's Algorithm

Recall that the adjacency matrix $A = \{a_{ij}\}$ of a directed graph is the boolean matrix that has 1 in its i th row and j th column if and only if there is a directed edge from the i th vertex to the j th vertex. We may also be interested in a matrix containing the information about the existence of directed paths of arbitrary lengths between vertices of a given graph.

DEFINITION The *transitive closure* of a directed graph with n vertices can be defined as the n -by- n boolean matrix $T = \{t_{ij}\}$, in which the element in the i th row ($1 \leq i \leq n$) and the j 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 th vertex to the j th vertex; otherwise, t_{ij} is 0.

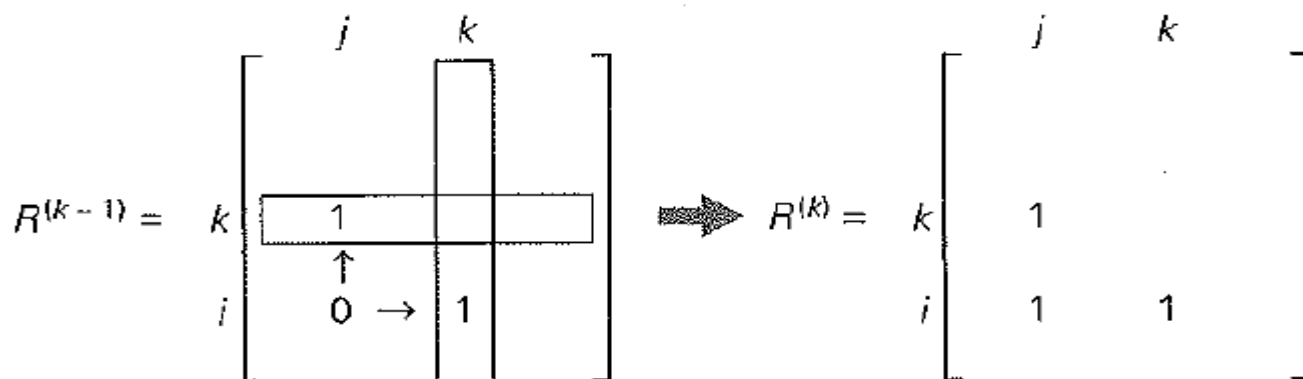
Warshall's algorithm after S. Warshall [War62]. Warshall's algorithm constructs the transitive closure of a given digraph with n vertices through a series of n -by- n boolean matrices:

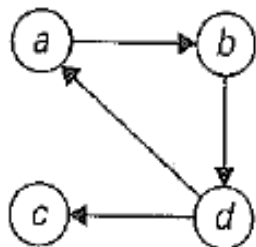
$$R^{(0)}, \dots, R^{(k-1)}, R^{(k)}, \dots, R^{(n)}. \quad (8.5)$$

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \text{ or } \left(r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)} \right). \quad (8.7)$$

Formula (8.7) is at the heart of Warshall's algorithm. This formula implies the following rule for generating elements of matrix $R^{(k)}$ from elements of matrix $R^{(k-1)}$, which is particularly convenient for applying Warshall's algorithm by hand:

- If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$.
- If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$. (This rule is illustrated in Figure 8.3.)





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

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 $R^{(1)}$.

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

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 $R^{(2)}$.

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

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).

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**

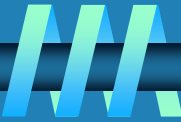
for $i \leftarrow 1$ **to** n **do**

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 $R^{(n)}$

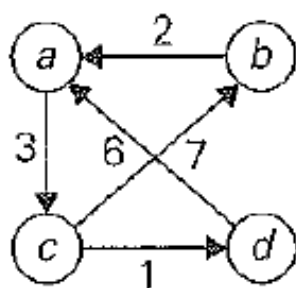




$$R^{(4)} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Floyd's Algorithm for the All-Pairs Shortest-Paths Problem

Given a weighted connected graph (undirected or directed), 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 th row and the j th column of this matrix indicates the length of the shortest path from the i th vertex to the j th vertex ($1 \leq i, j \leq n$). For an example, see Figure 8.5.



(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)

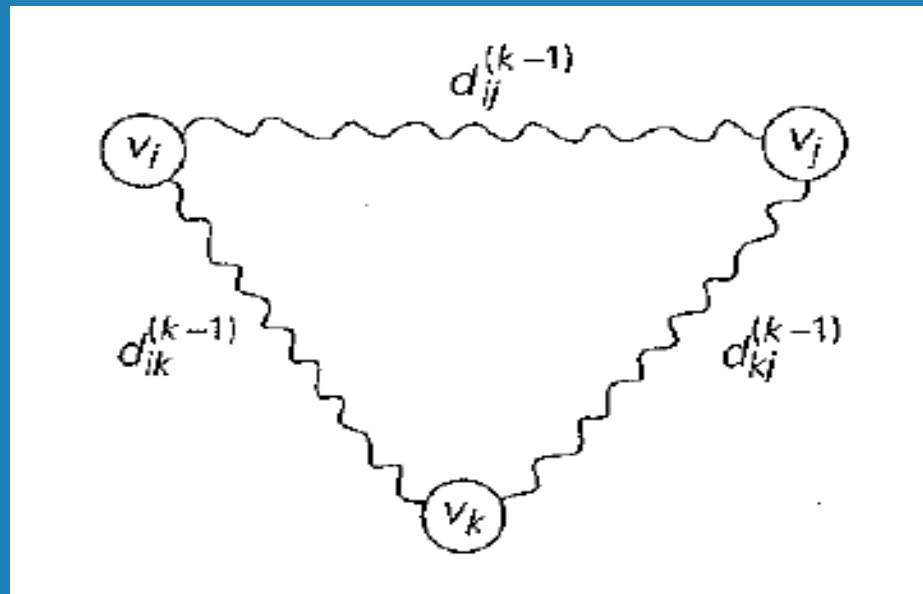
$$D = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{matrix}$$

(c)

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

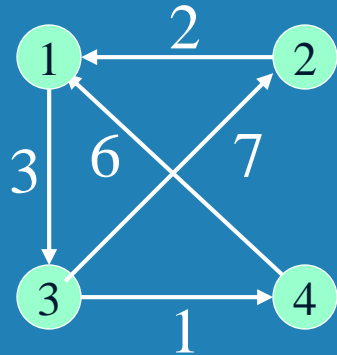
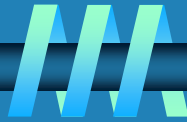
Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of n -by- n matrices:

$$D^{(0)}, \dots, D^{(k-1)}, D^{(k)}, \dots, D^{(n)}, \quad (8.8)$$



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

Floyd's Algorithm (example)



$$D^{(0)} =$$

0	∞	3	∞
2	0	∞	∞
∞	7	0	1
6	∞	∞	0

$$D^{(1)} =$$

0	∞	3	∞
2	0	5	∞
∞	7	0	1
6	∞	9	0

$$D^{(2)} =$$

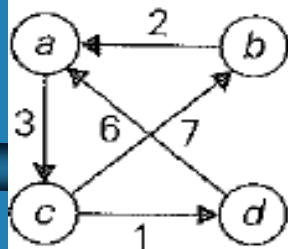
0	∞	3	∞
2	0	5	∞
9	7	0	1
6	∞	9	0

$$D^{(3)} =$$

0	10	3	4
2	0	5	6
9	7	0	1
6	16	9	0

$$D^{(4)} =$$

0	10	3	4
2	0	5	6
7	7	0	1
6	16	9	0



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

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

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

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).

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

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).

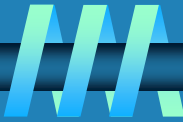
$$D^{(3)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{bmatrix} \end{matrix}$$

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).

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

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 (pseudocode and analysis)



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**

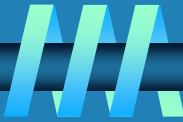
for $i \leftarrow 1$ **to** n **do**

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

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

return D

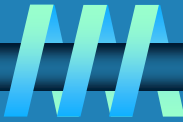
Problem



Solve the all-pairs shortest-path problem for the digraph with the weight matrix.

$$D^{(5)} = \begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & \infty & 1 & 8 \\ \infty & 0 & \infty & 1 & 8 \\ \infty & 0 & \infty & 1 & 8 \\ 3 & 0 & \infty & 1 & 8 \end{bmatrix}$$

Knapsack Problem by DP



Given n items of

integer weights: $w_1 \ w_2 \ \dots \ w_n$

values: $v_1 \ v_2 \ \dots \ v_n$

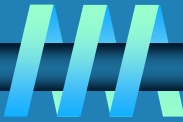
a knapsack of integer capacity W

find most valuable subset of the items that fit into the knapsack

To design a dynamic programming algorithm, we need to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller subinstances. Let us consider an instance defined by the first i items, $1 \leq i \leq n$, with weights w_1, \dots, w_i , values v_1, \dots, v_i , and knapsack capacity j , $1 \leq j \leq W$. Let $V[i, j]$ be the value of an optimal solution to this instance, i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j . We can divide all the subsets of the first i items that fit into the knapsack of capacity j into two categories: those that do not include the i th item and those that do. Note the following:

1. Among the subsets that do not include the i th item, the value of an optimal subset is, by definition, $V[i-1, j]$.
2. Among the subsets that do include the i 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 $v_i + V[i-1, j - w_i]$.

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



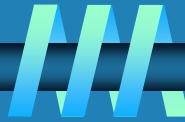
Consider instance defined by first i items and capacity j ($j \leq W$).

Let $V[i,j]$ be optimal value of such an instance. Then

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

Initial conditions: $V[0,j] = 0$ and $V[i,0] = 0$

Knapsack Problem by DP (example)



Example: Knapsack of capacity $W = 5$

item	weight	value
------	--------	-------

1	2	\$12
---	---	------

2	1	\$10
---	---	------

3	3	\$20
---	---	------

4	2	\$15
---	---	------

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

capacity j

		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

Backtracing
finds the actual
optimal subset,
i.e. solution.

Memory functions

ALGORITHM *MFKnapsack*(i, j)

//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 $j < \text{Weights}[i]$

$value \leftarrow \text{MFKnapsack}(i - 1, j)$

else

$value \leftarrow \max(\text{MFKnapsack}(i - 1, j),$

$\text{Values}[i] + \text{MFKnapsack}(i - 1, j - \text{Weights}[i]))$

$V[i, j] \leftarrow value$

return $V[i, j]$

Example

Example: Knapsack of capacity

$W = 5$

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

i/j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	-	12	22	-	22
3	0	-	-	22	-	32
4	0	-	-	-	-	37

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

capacity $W = 6$.