Analysis of Algorithms

Chapter 8.1, 8.2, 8.3, 8.4

ROAD MAP

- **Dynamic Programming**
  - Three Basic Examples
  - The Knapsack Problem
  - Matrix Chain Product
  - All Pairs Shortest Paths

Dynamic Programming

- **Definition :**
  - Dynamic programming is an interesting algorithm design technique for *optimizing multistage decision problems*
  - Programming in the name of this technique stands for *planning*
    - Does not refer to computer programming
  - It is a technique for solving problems with overlapping subproblems
    - Typically these subproblems arise from a recurrence relations
    - Suggests solving each of the smaller subproblems only once and recording the results in a table

Dynamic Programming

- Main idea:
  - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  - solve smaller instances once
  - record solutions in a table
  - extract solution to the initial instance from that table

Dynamic programming usually used for optimization problems

How do we get the recurrence relation?

Dynamic Programming

Principle of Optimality

A general principle that underlines dynamic programming for optimization problems.

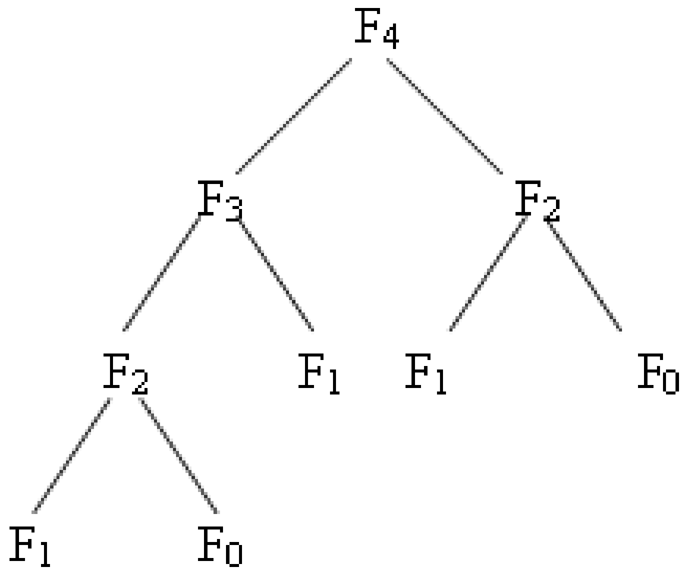
An optimal solution to any instance of an optimization problem is composed of optimal solutions to its subinstances.

Fibonacci Numbers

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

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

top-down solution: divide & conquer   - > O(2^n)



Fibonacci Numbers

Computing the nth 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	. . .	$F(n\text{-}2)$	$F(n\text{-}1)$	$F(n)$

Efficiency:

- time

- space

ROAD MAP

- **Dynamic Programming**
  - **Three Basic Examples**
  - The Knapsack Problem
  - Matrix Chain Product
  - All Pairs Shortest Paths

Coin-row problem

- *There is a row of n coins whose values are some positive integers c_1, c_2,...,c_n, not necessarily distinct.*

- *The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.*

E.g.: 5, 1, 2, 10, 6, 2. What is the best selection?

Coin-row problem

- *Let F(n) be the maximum amount that can be picked up from the row of n coins. To derive a recurrence for F(n), we partition all the allowed coin selections into two groups:*

those without last coin – the max amount is ?
those with the last coin -- the max amount is ?

Thus we have the following *<u>recurrence:</u>*

$$F(n) = \max\{c_n + F(n-2),\ F(n-1)\} \quad \text{for } n > 1$$
$$F(0) = 0, \qquad F(1) = c_1.$$

Coin-row problem

$$F(n) = \max\{c_n + F(n-2),\ F(n-1)\} \quad \text{for } n > 1$$

$$F(0) = 0, \qquad F(1) = c_1.$$

$F[0] = 0$, $F[1] = c_1 = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5					

$F[2] = \max\{1 + 0, 5\} = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5				

$F[3] = \max\{2 + 5, 5\} = 7$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7			

$F[4] = \max\{10 + 5, 7\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15		

$F[5] = \max\{6 + 7, 15\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	

$F[6] = \max\{2 + 15, 15\} = 17$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	**17**

Coin-row problem

$$F(n) = \max\{c_n + F(n-2),\ F(n-1)\} \quad \text{for } n > 1$$
$$F(0) = 0, \qquad F(1) = c_1.$$

ALGORITHM $CoinRow(C[1..n])$

//Applies formula (8.3) bottom up to find the maximum amount of money
//that can be picked up from a coin row without picking two adjacent coins
//Input: Array $C[1..n]$ of positive integers indicating the coin values
//Output: The maximum amount of money that can be picked up
$F[0] \leftarrow 0;\quad F[1] \leftarrow C[1]$
for $i \leftarrow 2$ **to** n **do**
    $F[i] \leftarrow \max(C[i] + F[i-2],\ F[i-1])$
return $F[n]$

Change-making problem

- *Give change for amount n using the minimum number of denominations $d_1 < d_2 < \ldots\ldots < d_m$.*

- *Let F(n) be the minimum number of coins whose values add up to n and F(0) = 0. The amount n can be obtained by adding one coin of denomination dj to the amount n-d_j for j = 1, 2, ..., m such that $n \geq d_j$. Therefore, we can consider all such denominations and select the one minimizing F(n-d_j) + 1.*

The recurrence:

$$F(n) = \min_{j:\, n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$

$$F(0) = 0.$$

Change-making problem

$$F(n) = \min_{j:n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$

$$F(0) = 0.$$

$n = 6$

denominations 1, 3, and 4.

$F[0] = 0$

n	0	1	2	3	4	5	6
F	0						

$F[1] = \min\{F[1-1]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1					

$F[2] = \min\{F[2-1]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2				

$F[3] = \min\{F[3-1], F[3-3]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1			

$F[4] = \min\{F[4-1], F[4-3], F[4-4]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1	1		

$F[5] = \min\{F[5-1], F[5-3], F[5-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	

$F[6] = \min\{F[6-1], F[6-3], F[6-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	**2**

Change-making problem

$$F(n) = \min_{j:n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$

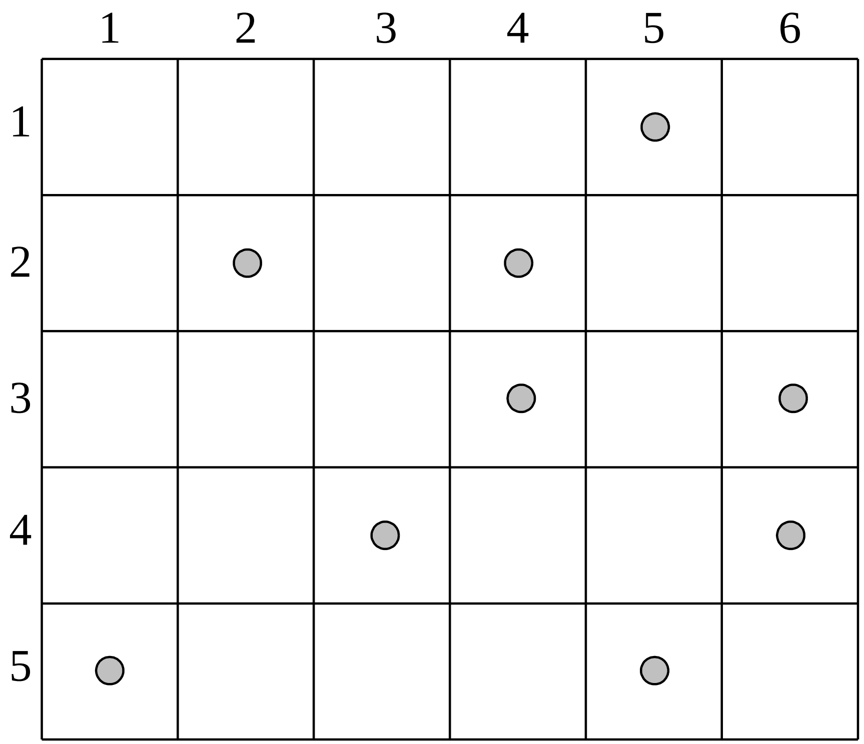
$$F(0) = 0.$$

ALGORITHM $ChangeMaking(D[1..m], n)$

//Applies dynamic programming to find the minimum number of coins
//of denominations $d_1 < d_2 < \cdots < d_m$ where $d_1 = 1$ that add up to a
//given amount n
//Input: Positive integer n and array $D[1..m]$ of increasing positive
// integers indicating the coin denominations where $D[1] = 1$
//Output: The minimum number of coins that add up to n
$F[0] \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
  $temp \leftarrow \infty$; $j \leftarrow 1$
  while $j \leq m$ **and** $i \geq D[j]$ **do**
    $temp \leftarrow \min(F[i - D[j]], temp)$
    $j \leftarrow j + 1$
  $F[i] \leftarrow temp + 1$
return $F[n]$

Coin-collecting by robot

Several coins are placed in cells of an $n \times m$ board. A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell. On each step, the robot can move either one cell to the right or one cell down from its current location.



Coin-collecting problem

Let F(*i*,*j*) be the largest number of coins the robot can collect and bring to cell (*i*,*j*) in the *i*th row and *j*th column.

The largest number of coins that can be brought to cell (*i*,*j*):

from the left neighbor ?
from the neighbor above?

<u>The recurrence</u>:

$$F(i, j) = \max\{F(i-1, j), F(i, j-1)\} + c_{ij} \quad \text{for } 1 \le i \le n, \ 1 \le j \le m$$

$$F(0, j) = 0 \ \text{ for } 1 \le j \le m \quad \text{and} \quad F(i, 0) = 0 \ \text{ for } 1 \le i \le n,$$

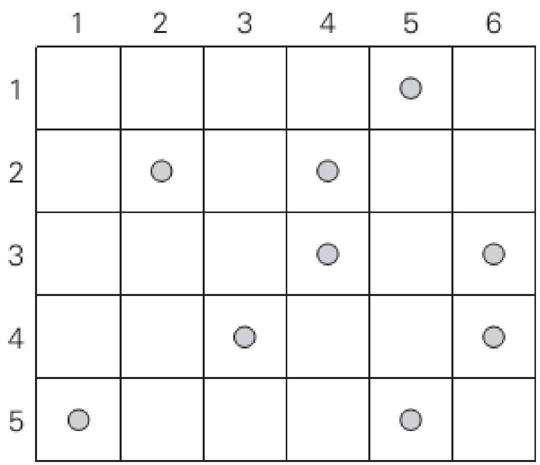
where $c_{ij} = 1$ if there is a coin in cell (i, j), and $c_{ij} = 0$ otherwise

Coin-collecting problem

$$F(i, j) = \max\{F(i-1, j), F(i, j-1)\} + c_{ij} \quad \text{for } 1 \le i \le n, \ \ 1 \le j \le m$$

$$F(0, j) = 0 \ \text{ for } 1 \le j \le m \quad \text{and} \quad F(i, 0) = 0 \ \text{ for } 1 \le i \le n,$$

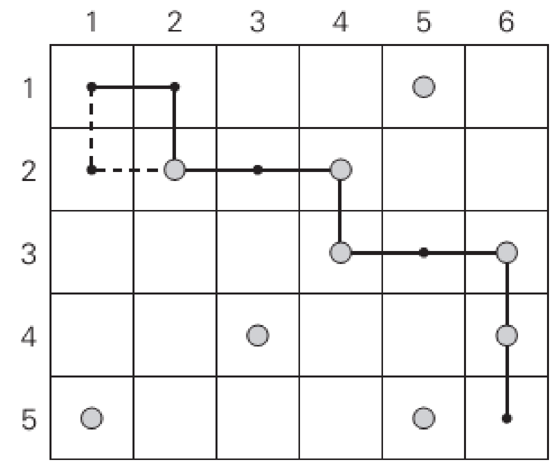
where $c_{ij} = 1$ if there is a coin in cell (i, j), and $c_{ij} = 0$ otherwise



(a)

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

(b)



(c)

Coin-collecting problem

$$F(i, j) = \max\{F(i-1, j), F(i, j-1)\} + c_{ij} \quad \text{for } 1 \le i \le n, \ 1 \le j \le m$$

$$F(0, j) = 0 \ \text{ for } 1 \le j \le m \quad \text{and} \quad F(i, 0) = 0 \ \text{ for } 1 \le i \le n,$$

where $c_{ij} = 1$ if there is a coin in cell (i, j), and $c_{ij} = 0$ otherwise

ALGORITHM *RobotCoinCollection*$(C[1..n, 1..m])$

//Applies dynamic programming to compute the largest number of
//coins a robot can collect on an $n \times m$ board by starting at (1, 1)
//and moving right and down from upper left to down right corner
//Input: Matrix $C[1..n, 1..m]$ whose elements are equal to 1 and 0
//for cells with and without a coin, respectively
//Output: Largest number of coins the robot can bring to cell (n, m)
$F[1, 1] \leftarrow C[1, 1]$; **for** $j \leftarrow 2$ **to** m **do** $F[1, j] \leftarrow F[1, j-1] + C[1, j]$
for $i \leftarrow 2$ **to** n **do**
  $F[i, 1] \leftarrow F[i-1, 1] + C[i, 1]$
  for $j \leftarrow 2$ **to** m **do**
    $F[i, j] \leftarrow \max(F[i-1, j], F[i, j-1]) + C[i, j]$
return $F[n, m]$

ROAD MAP

- **Dynamic Programming**
  - Three Basic Examples
  - **The Knapsack Problem**
  - Matrix Chain Product
  - All Pairs Shortest Paths

Knapsack Problem

- Definition:
  - Given n items of known weights $w_1, w_2, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity W
  - Find the most valuable subset of the items that fit into the knapsack
  - How can we design a dynamic programming algorithm ?

Knapsack Problem

- **Approach :**
  - We need to derive a recurrence relation
    - expresses a solution to an instance of the problem in terms of solutions to its smaller subinstances
  - Consider an instance defined by the first i items $1 \le i \le n$
    - with weights $w_1, \ldots, w_i$
    - values $v_1, \ldots, v_i$
    - capacity j $1 \le j \le W$
  - $V[i, j]$ be the value of an optimal solution to this instance
    - Total value of the most suitable subset of first i items that fit into the knapsack of capacity j

Knapsack Problem

<u>**Approach :**</u>

We can divide all subsets of the first i items that fit the knapsack of capacity j into two categories

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

Knapsack Problem

- So, the following recurrence

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

- Initial conditions

$$V[0,j] = 0 \quad \text{for} \quad j \geq 0$$

$$V[i,0] = 0 \quad \text{for} \quad i \geq 0$$

How to solve this recurrence??

Knapsack Problem

		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

Table for solving the knapsack problem by dynamic programming

- **Example :**

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

capacity $W = 5$

The given instance

capacity j

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

Dynamic programming table

Maximal value is *V[4, 5] = 37*

Knapsack Problem

- Analysis :
  - Time efficiency and space effciency of this algorithm is $\Theta(nW)$
  - The time needed to find the composition of an optimal solution is in $O(n+W)$

ROAD MAP

- **Dynamic Programming**
  - Three Basic Examples
  - The Knapsack Problem
  - **Matrix Chain Product**
  - All Pairs Shortest Paths

Matrix Chain Product

- Multiply *n* given matrices

  $M_1 \text{ x } M_2 \text{ x } \dots \text{ x } M_n$

- <u>Goal :</u>
  - Find the parenthesization to minimize the number of multiplications
    - M_1 , M_2 , M_3

      10x30 30x5 5x60

$$M_1 \, x \, M_2 \; x \, M_3 \quad = \begin{cases} (M_1 \, x \, M_2) \, x \, M_3 & 10x30x5 + 10x5x60 \quad = 4500 \\ M_1 \, x \, (M_2 \; x \, M_3) & 30x5x60 + 10x30x60 = 27000 \end{cases}$$

- (n-1) ! = $O(2^n)$ different parenthesization

Matrix Chain Product

- What is the sequence of decisions?
- What are possible choices at each decision point?
- What about principle of optimality…
- How to write the recurrence relation?

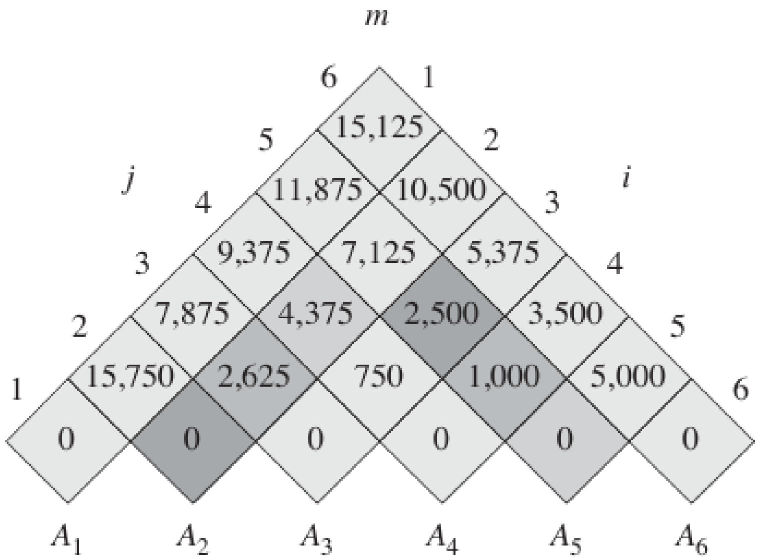
Matrix Chain Product

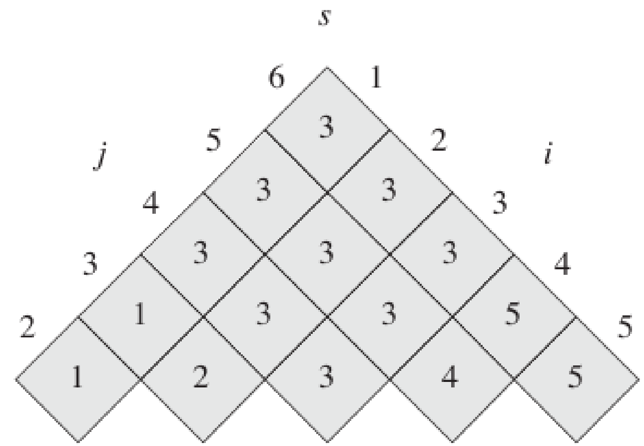
$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j \end{cases}$$

- # of m(i,j)'s = $O(n^2)$
- For each m(i,j) we check (j-i) expressions
- Overall complexity is $O(n^3)$

Matrix Chain Product

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25
	$p_0 \; x \; p_1$	$p_1 \; x \; p_2$	$p_2 \; x \; p_3$	$p_3 \; x \; p_4$	$p_4 \; x \; p_5$	$p_5 \; x \; p_6$





$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000 , \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 , \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}$$

$$= 7125 .$$

Matrix Chain Product

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
    print "A"_i
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

PRINT-OPTIMAL-PARENS$(s, 1, 6)$

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

ROAD MAP

- **Dynamic Programming**
  - Three Basic Examples
  - The Knapsack Problem
  - Matrix Chain Product
  - **All Pairs Shortest Paths**
    - **Floyd’s Algorithm**

All-Pair Shortest Path Problem

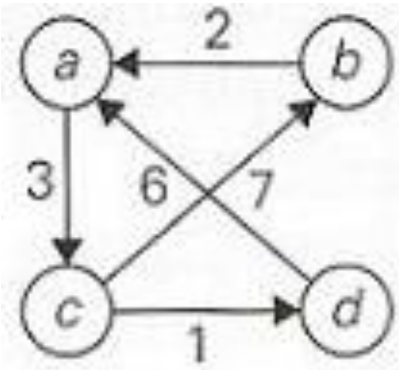
- Definition:
  - Given a weighted graph G,
    - G has no cycle with negative length
  - Compute the distances (***the length of the shortest paths)*** between every pair of vertices in a graph *G*

Specifically:

- Find D = Distance matrix

  where d_{ij} = length of the shortest path from i to j

All-Pair Shortest Path Problem



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

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

Digraph **weight matrix** **distance matrix**

All-Pair Shortest Path Problem

- What is the sequence of decisions?
- What are possible choices at each decision point?
- What about principle of optimality…
- How to write the recurrence relation?

All-Pair Shortest Path Problem

Idea:

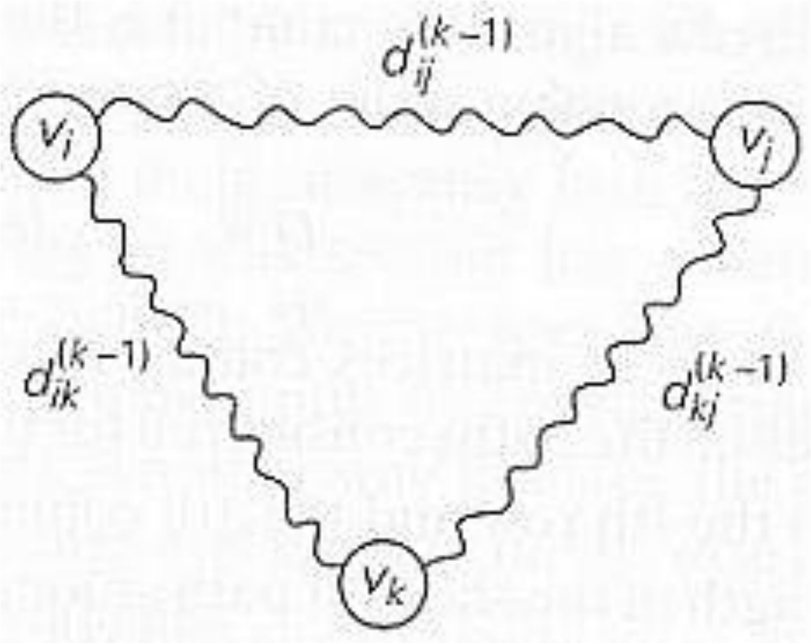
- Compute D through a series of *n-by-n* matrices

$$W=D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)}=D$$

- where $d_{ij}^{(k)}$ = the length of the shortest path from i^{th} vertex to j^{th} vertex with each intermediate vertex, if any, numbered not higher than k
  - k is the largest index on the path
- Optimal path from i to j contains no cycle
  - Vertex k appears only once on the path
- Because of the principle of optimality

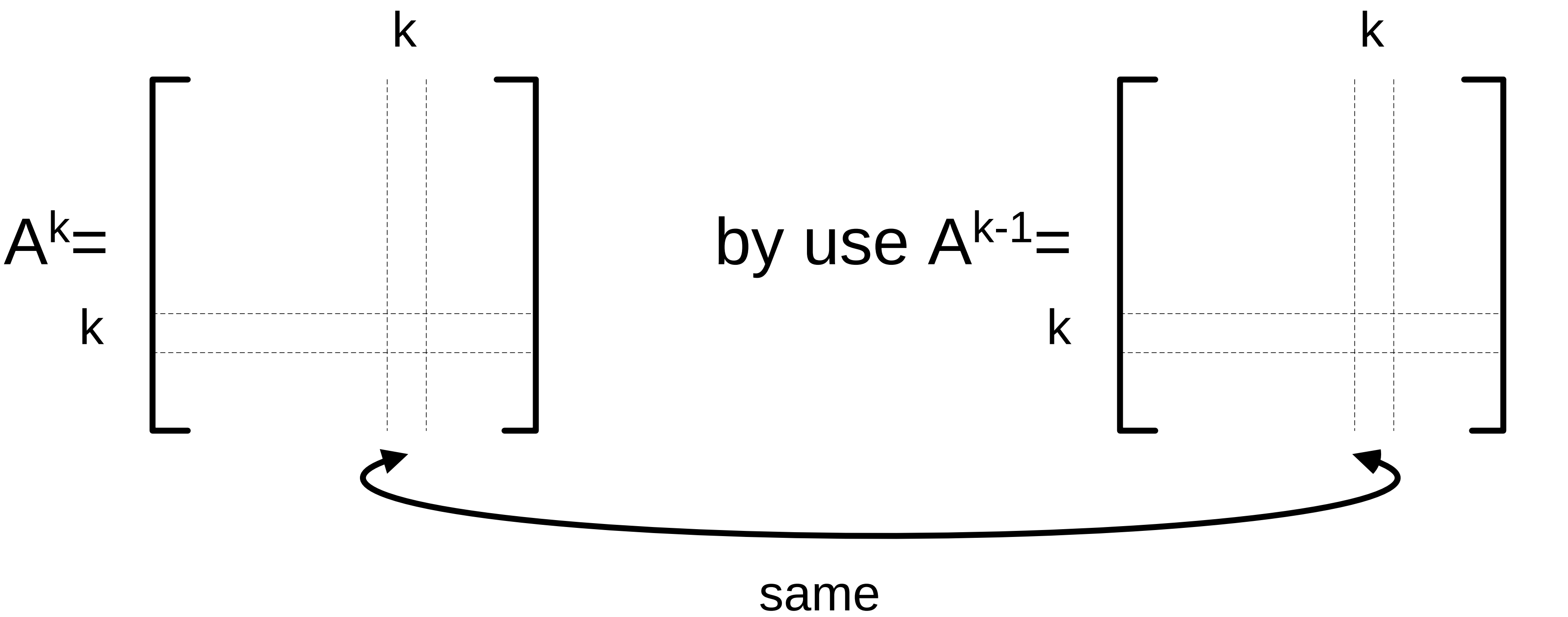
$$v_i, \text{ vertices numbered} \le k-1, v_k, \text{ vertices numbered} \le k-1, v_j$$

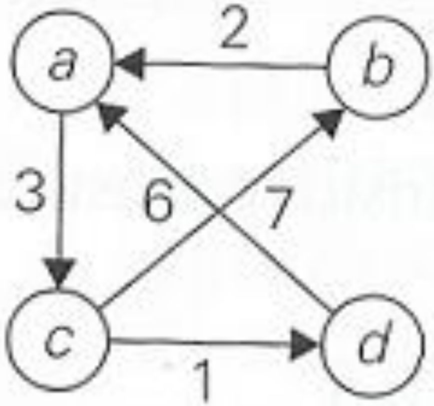
Floyd's Algorithm



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

All-Pair Shortest Path Problem





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

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

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

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

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

Floyd's Algorithm

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

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

Time efficiency is cubic