

Course – Analysis and Design of Algorithms

Unit 4: Dynamic Programming



Unit 3: Dynamic Programming

- Three Basic Examples: Coin-Row Problem, Change-Making Problem and Coin-Collecting Problem
- The Knapsack Problem [Without Memory Functions],
- Warshall's and Floyd's Algorithms

What is Dynamic Programming technique ?

- Dynamic programming is a technique for solving problems with **overlapping subproblems**.
- Typically, these subproblems arise from a **recurrence** relating a given problem's solution to solutions of its smaller subproblems.
- Rather than solving overlapping subproblems again and again, dynamic programming suggests **solving each of the smaller subproblems only once and recording the results in a table** from which a solution to the original problem can then be obtained.
- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems.

What is Dynamic Programming technique ?

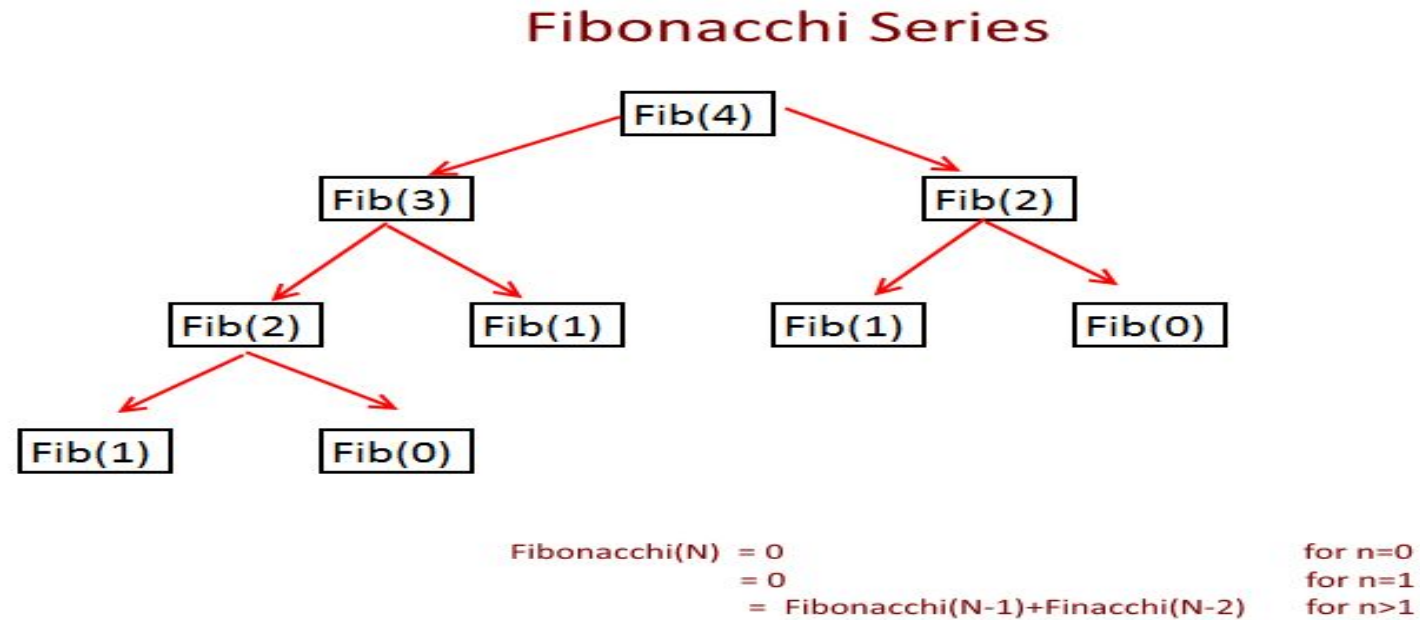
- The word “Programming” in the name of this technique refers to “**Planning**” and does not refer to computer programming.
- Dynamic programming is a technique to solve the recursive problems in more efficient manner. Many times in recursion we solve the sub-problems repeatedly. In dynamic programming we store the solution of these sub-problems so that we do not have to solve them again, this is called **Memoization**.
- **Dynamic programming** and **memoization** works together. So most of the problems are solved with two components of dynamic programming (DP):
 - **Recursion** — Solve the sub-problems recursively
 - **Memoization** — Store the solution of these sub-problems so that we do not have to solve them again

Example: (Top-Down)

Fibonacci(N)

```
0          for n=0
1          for n=1
Fibonacci(N-1)+Finacci(N-2)    for n>1
int fibRecur(int x) {
    if (x == 0)
        return 0;
    if (x == 1)
        return 1;
    else {
        int f = fibRecur(x - 1) + fibRecur(x - 2);
        return f;
    }
}
```

Example



Now as you can see in the picture above while you are calculating Fibonacci(4) you need Fibonacci(3) and Fibonacci(2), Now for Fibonacci(3), you need Fibonacci (2) and Fibonacci (1) but you notice you have calculated Fibonacci(2) while calculating Fibonacci(4) and again calculating it. So we are solving many sub-problems again and again.

Time Complexity:

$$T(n) = T(n-1) + T(n-2) + 1 = 2^n = \mathbf{O(2^n)}$$

Example: (Bottom-Up)

- Store the sub-problems result so that you don't have to calculate again. So first check if solution is already available, if yes then use it else calculate and store it for future.

```
int fibDP(int x) {  
    int fib[] = new int[x + 1];  
    fib[0] = 0;  
    fib[1] = 1;  
    for (int i = 2; i < x + 1; i++) {  
        fib[i] = fib[i - 1] + fib[i - 2];  
    }  
    return fib[x];  
}
```

- **Time Complexity: $O(n)$, Space Complexity : $O(n)$**

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

Two major properties of Dynamic programming

To decide whether problem can be solved by applying Dynamic programming we check for two properties. If problem has these two properties then we can solve that problem using Dynamic programming.

- ❑ Overlapping Sub-problems
- ❑ Optimal Substructure.

Overlapping Sub-problems:

- ❑ Overlapping sub-problems, as the name suggests the **sub-problems needs to be solved again and again**. In recursion we solve those problems every time and in dynamic programming we solve these sub problems only once and store it for future use.
- ❑ **Optimal Substructure:** If a problem can be solved by using the solutions of the sub problems then we say that problem has a ***Optimal Substructure Property***.

Dynamic programming

□ Basic Steps:

- **Analyze the structure of a solution:** how the solution of a problem depends on the solutions of subproblems. Usually this step means verifying the **optimal substructure property**.
- **Find a recurrence relation** which relates a value (e.g. the **optimization criterion**) corresponding to the problem's solution with the values corresponding to subproblems solutions.
- **Develop in a bottom up manner the recurrence relation** and construct a **table** containing information useful to construct the solution.
- **Construct the solution** based on information collected in the previous step.

DP Example 1: Coin-Row Problem

- There is a row of n coins whose values are some positive integers c_1, c_2, \dots, 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.: 5,1,6,10,5,2 What is the best selection?

DP solution to: Coin-Row Problem

- $C_1, C_2, \dots, C_{n-2}, C_{n-1}, C_n,$
- Let $F(n)$ = maximum amount that can be picked up from the first n coins in the row of 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 ?
- $C_1, C_2, \dots, C_{n-2}, C_{n-1}, C_n,$
- - Thus we have the following recurrence :
$$F(n) = \max\{C_n + F(n-2), F(n-1)\} \text{ for } n > 1,$$
$$F(0) = 0, F(1) = C_1$$

DP solution to: Coin-Row Problem

- $F(n) = \max\{cn + F(n-2), F(n-1)\}$ for $n > 1$,
 $F(0) = 0, F(1)=c_1$

index	0	1	2	3	4	5	6
coins	--	5	1	6	10	5	2
Include		C1	--	C3	C4	C5	C6
F()	F(0) 0 N	F(1) 5 Y	F(2) 5 N	F(3) 11 Y	F(4) 15 Y	F(5) 16 Y	F(6) 17 Y

Max amount:17 Coins of optimal solution:{c1,c4,c6}

DP solution to: Coin-Row Problem

To find coins, remember choice at each step:

1. Add row $P(n)$: 0/1 = leave/pickup coin n (at that point)
2. Fill in rows F and P together, left to right
3. For final choice, work backwards from right:

0=leave and move 1 left, 1=pickup and move 2 left

i	0	1	2	3	4	5	6
c(i)	--	5	1	6	10	5	2
F(i)	0	5	5	11	15	16	17
Choice		1	0	1	1	1	1
Pick		1			1		1

□ Max amount: 17 Coins of optimal solution: {c1,c4,c6}

DP solution to: Coin-Row Problem

- Using dynamic programming solve the coin-row problem for the coin row :
5,1,2,10,6,2
- $F(n) = \max\{c_n + F(n-2), F(n-1)\}$ for $n > 1$,
 $F(0) = 0, F(1) = c_1$

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

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

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

FIGURE 8.1 Solving the coin-row problem by dynamic programming for the coin row 5, 1, 2, 10, 6, 2.

DP solution to: Coin-Row Problem

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

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

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

FIGURE 8.1 Solving the coin-row problem by dynamic programming for the coin row 5, 1, 2, 10, 6, 2.

Max amount: 17

Coins of optimal solution: {c1, c4, c6}

DP solution to: Coin-Row Problem

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 $F[n]$

TIME COMPLEXITY IS $O(n)$

DP solution to: Coin-Row Problem

Solve :

5, 1, 9, 10, 8, 2

DP Example 2 : Change-Making Problem

- Consider the general instance of the following well-known problem.
- **Give change for amount n using the minimum number of coins** of denominations $d_1 < d_2 < \dots < d_m$.
- Here, we consider a dynamic programming algorithm for the general case, assuming availability of **unlimited quantities of coins** for each of the m denominations $d_1 < d_2 < \dots < d_m$ where $d_1 = 1$.



DP Example 2 : Change-Making Problem

- Let $F(n)$ be the minimum number of coins whose values add up to n ; it is convenient to define $F(0) = 0$.
- The amount n can only be obtained by adding one coin of denomination d_j to the amount $n - d_j$ for $j = 1, 2, \dots, m$ such that $n \geq d_j$.
- Therefore, we can consider all such denominations and select the one minimizing $F(n - d_j) + 1$.
- Since 1 is a constant, we can, of course, find the smallest $F(n - d_j)$ first and then add 1 to it.
- Hence, we have the following recurrence for $F(n)$:

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

$$F(0) = 0$$

DP Example 2 : Change-Making Problem

- We can compute $F(n)$ by filling a one-row table left to right, but computing a table entry here requires finding the minimum of up to m numbers.

$$F[0] = 0$$

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

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

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

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

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

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

n	0	1	2	3	4	5	6
F	0						

n	0	1	2	3	4	5	6
F	0	1					

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

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

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

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

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

FIGURE 8.2 Application of Algorithm *MinCoinChange* to amount $n = 6$ and coin denominations 1, 3, and 4.

max amount: 17 coins of optimal solution: {c1,c4,c0}

Amount n = 6 Denominations: 1,3,4

$$F(n) = \min\{F(n - d_j)\} + 1$$

$$F[0] = 0$$

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

DP Example 2 : Change-Making Problem

- To find the coins of an optimal solution, we need to backtrace the computations to see which of the denominations produced the minima in the formula:
$$F(n) = \min\{F(n - d_j)\} + 1 \quad \text{for } n > 0, j: n \geq d_j$$
- For the instance considered, the last application of the formula (for $n = 6$), the minimum was produced by $d_2 = 3$.
- The second minimum (for $n = 6 - 3$) was also produced for a coin of that denomination.
- Thus, the minimum-coin set for $n = 6$ is two coins of 3's.

DP Example 2 : Change-Making Problem

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

//Applies dynamic programming to find the minimum number of coins
//of denominations $d_1 < d_2 < \dots < d_m$ 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 $F[n]$

DP Example 3: Coin-Collection Problem

- Several coins are placed in cells of an $n \times m$ board, no more than one coin per cell.
- 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.
- When the robot visits a cell with a coin, it always picks up that coin.
- Design an algorithm to find the maximum number of coins the robot can collect and a path it needs to follow to do this.

	1	2	3	4	5	6
1					●	
2		●		●		
3				●		●
4			●			●
5	●				●	

DP Example 3: Coin-Collection Problem

- Let $F(i,j)$ be the largest number of coins the robot can collect and bring to the cell (i,j) in the i th row and j th column of the board.
- It can reach this cell either from the adjacent cell $(i-1, j)$ above it or from the adjacent cell $(i, j-1)$ to the left of it.
- The largest numbers of coins that can be brought to these cells are $F(i-1,j)$ and $F(i, j-1)$, respectively.
- Of course, there are no adjacent cells above the cells in the first row, and there are no adjacent cells to the left of them cells in the first column.
- For those cells, we assume that $F(i-1,j)$ and $F(i, j-1)$ are equal to 0 for their nonexistent neighbors.

	1	2	3	4	5	6
1					●	
2		●		●		
3				●		●
4			●			●
5	●				●	

DP Example 3: Coin-Collection Problem

- Therefore, the largest number of coins the robot can bring to cell (i,j) is the maximum of these two numbers plus one possible coin at cell (i,j) itself.
- In other words, we have the following formula for $F(i, j)$:

$$F(i, j) = \max\{F(i-1, j), F(i, j-1)\} + c_{ij} \quad \text{for } 1 \leq i \leq n, 1 \leq j \leq m$$
$$F(0, j) = 0 \text{ for } 1 \leq j \leq m \text{ and } F(i, 0) = 0 \text{ for } 1 \leq i \leq n,$$

- where $c_{ij} = 1$ if there is a coin in cell (i, j) , and $c_{ij} = 0$ otherwise.
- Using these formulas, we can fill in the $n \times m$ table of $F(i, j)$ values either row by row or column by column, as is typical for dynamic programming algorithms involving two-dimensional tables.

DP Example 3: Coin-Collection Problem

$$F(i, j) = \max\{F(i-1, j), F(i, j-1)\} + c_{ij} \quad \text{for } 1 \leq i \leq n, 1 \leq j \leq m$$
$$F(0, j) = 0 \text{ for } 1 \leq j \leq m \text{ and } F(i, 0) = 0 \text{ for } 1 \leq i \leq n,$$

	1	2	3	4	5	6
1					●	
2		●		●		
3				●		●
4			●			●
5	●				●	

(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)

DP Example 3: Coin-Collection Problem

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

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

- Tracing the computations backward makes it possible to get an optimal path:
- if $F(i-1, j) > F(i, j-1)$, an optimal path to cell (i, j) must come down from the adjacent cell above it;
- if $F(i-1, j) < F(i, j-1)$, an optimal path to cell (i, j) must come from the adjacent cell on the left; and
- if $F(i-1, j) = F(i, j-1)$, it can reach cell (i, j) from either direction.
- This yields two optimal paths for the instance in Figure 8.3a, which are shown in Figure 8.3c. If ties are ignored, one optimal path can be obtained in $(n + m)$ time.

DP Example 3: Coin-Collection Problem

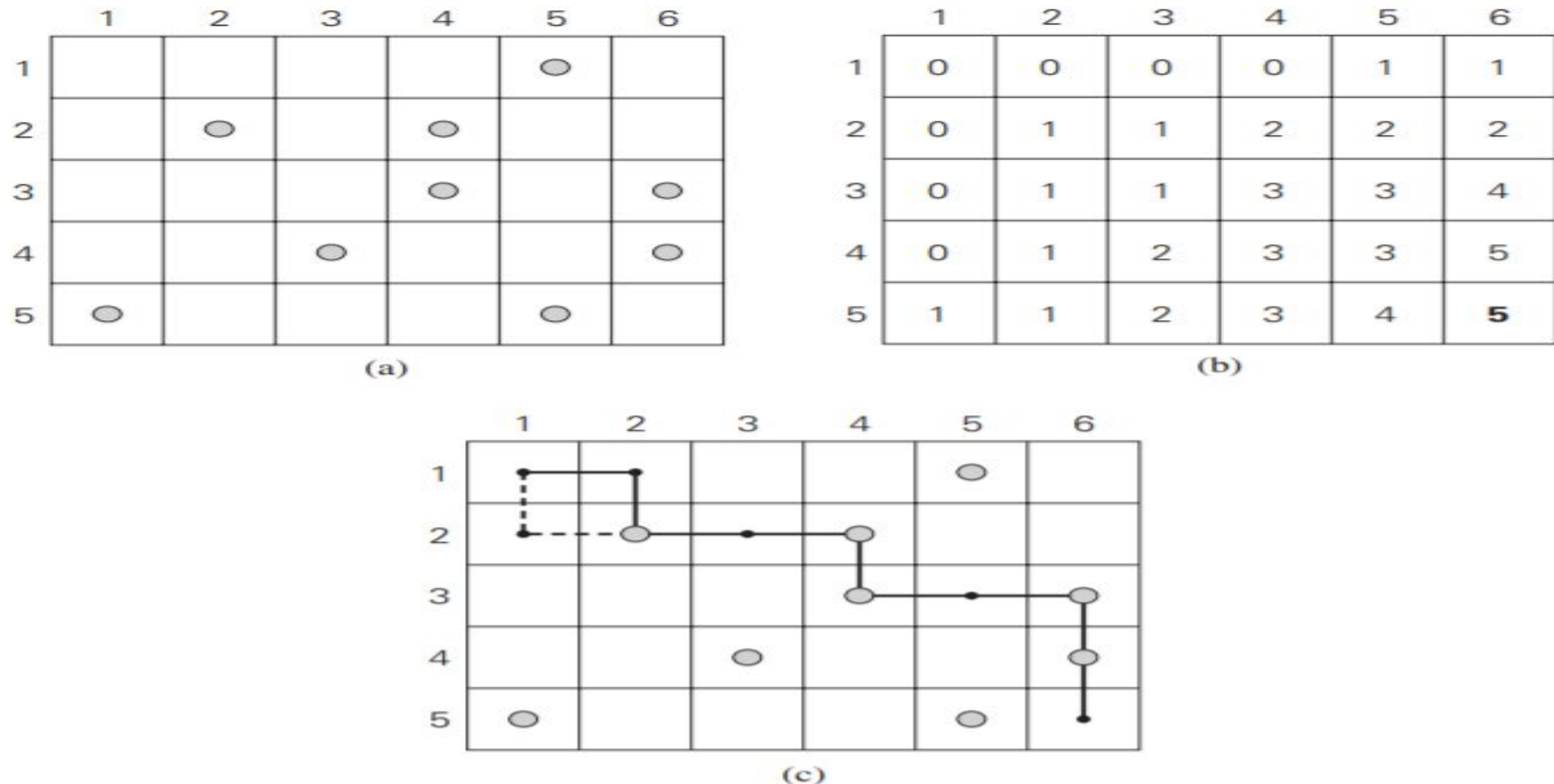


FIGURE 8.3 (a) Coins to collect. (b) Dynamic programming algorithm results. (c) Two paths to collect 5 coins, the maximum number of coins possible.

DP Example 3: Coin-Collection Problem

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 $F[n, m]$

Solving Knapsack Problem using Dynamic Programming

Recall of the Divide-and-Conquer

1. Partition the problem into subproblems.
2. Solve the subproblems.
3. Combine the solutions to solve the original one.

Remark: If the subproblems are not independent, i.e. subproblems share subsubproblems, then a divide-and-conquer algorithm repeatedly solves the common subsubproblems.

Thus, it does more work than necessary!

Question: Any better solution?

Yes–Dynamic programming (DP)!

The Idea of Dynamic Programming

Dynamic programming is a method for solving optimization problems.

The idea: Compute the solutions to the subsub-problems once and store the solutions in a table, so that they can be reused (repeatedly) later.

Remark: We trade space for time.

Principle of Optimality

Problems should satisfy “Principle of Optimality” to become eligible for solving using “Dynamic Programming” Technique.

Optimal Substructure

- A problem has optimal substructure if an optimal solution can be constructed efficiently from optimal solutions of its subproblems

Knapsack Problem by DP

Given n items of

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

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

a knapsack of integer capacity W

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

Recursive solution?

What is smaller problem?

How to use solution to smaller in solution to larger Table?

Order to solve?

Initial conditions?

Knapsack Problem by DP (example)

Example: Knapsack of capacity $W = 5$

<u>item</u>	<u>weight</u>	<u>value</u>
-------------	---------------	--------------

1	2	12
---	---	----

2	1	10
---	---	----

3	3	20
---	---	----

4	2	15
---	---	----

$w_1 = 2, v_1 = 12$	1
---------------------	---

$w_2 = 1, v_2 = 10$	2
---------------------	---

$w_3 = 3, v_3 = 20$	3
---------------------	---

$w_4 = 2, v_4 = 15$	4
---------------------	---

capacity j

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

?

Knapsack Problem by DP (example)

Example: Knapsack of capacity $W = 5$

<u>item</u>	<u>weight</u>	<u>value</u>
-------------	---------------	--------------

1	2	12
---	---	----

2	1	10
---	---	----

3	3	20
---	---	----

4	2	15
---	---	----

		capacity j					
		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

Knapsack Problem by DP (example)

1. Among the subsets that **do not include the i th item**, the value of an optimal subset is, by definition, $F(i - 1, j)$.

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 fits into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + F(i - 1, j - w_i)$.

Knapsack of capacity $W = 5$

		capacity j					
		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

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

Recursive solution?

What is smaller problem?

How to use solution to smaller in solution to larger Table?

Order to solve?

Initial conditions?

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.

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

Let $V[i,j]$ be optimal value of such 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 $V[i,0] = 0$

0/1 Knapsack Problem Solving using Dynamic Programming

- 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 sub-instances.
- 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 $F(i, j)$ 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 the knapsack of capacity j into two categories: those that do not include the i th item and those that do.

0/1 Knapsack Problem Solving using Dynamic Programming

Note the following:

- 1.** Among the subsets that **do not include the i th item**, the value of an optimal subset is, by definition, $F(i - 1, j)$.
- 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 fits into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + F(i - 1, j - w_i)$.

Thus, the value of an **optimal solution** among all feasible subsets of the first i items is the **maximum of these two values**.

Of course, if the i th item does not fit into the knapsack, the value of an optimal subset selected from the first i items is the same as the value of an optimal subset selected from the first $i - 1$ items.

0/1 Knapsack Problem Solving using Dynamic Programming

Recurrence:

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

It is convenient to define the initial conditions as follows:

$$F(0, j) = 0 \text{ for } j \geq 0 \quad \text{and} \quad F(i, 0) = 0 \text{ for } i \geq 0.$$

- Our goal is to find $F(n, W)$, the maximal value of a subset of the n given items that fit into the knapsack of capacity W , and an optimal subset itself.
- For $i, j > 0$, to compute the entry in the i th row and the j th column, $F(i, j)$, 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.

		0	$j-w_i$	j	W
w_i, v_i	0	0	0	0	0
	$i-1$	0	$F(i-1, j-w_i)$	$F(i-1, j)$	
	i	0		$F(i, j)$	
	n	0			goal

Table for solving the knapsack problem by dynamic programming.

Example of solving an instance of the knapsack problem by the dynamic programming algorithm.

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

- Thus, the maximal value is $F(4, 5) = 37$. We can find the composition of an optimal subset by backtracing the computations of this entry in the table.
- Since $F(4, 5) > F(3, 5)$, item 4 has to be included in an optimal solution along with an optimal subset for filling $5 - 2 = 3$ remaining units of the knapsack capacity.
- The value of the latter is $F(3, 3)$. Since $F(3, 3) = F(2, 3)$, item 3 need not be in an optimal subset. Since $F(2, 3) > F(1, 3)$, item 2 is a part of an optimal selection, which leaves element $F(1, 3 - 1)$ to specify its remaining composition. Similarly, since $F(1, 2) > F(0, 2)$, item 1 is the final part of the optimal solution {item 1, item 2, item 4}.

Question

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 $W = 6$.

- b. How many different optimal subsets does the instance of part (a) have?
- c. In general, how can we use the table generated by the dynamic programming algorithm to tell whether there is more than one optimal subset for the knapsack problem's instance?

Answer

a.

		<i>capacity j</i>							
		<i>i</i>	0	1	2	3	4	5	6
		0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$		1	0	0	0	25	25	25	25
$w_2 = 2, v_2 = 20$		2	0	0	20	25	25	45	45
$w_3 = 1, v_3 = 15$		3	0	15	20	35	40	45	60
$w_4 = 4, v_4 = 40$		4	0	15	20	35	40	55	60
$w_5 = 5, v_5 = 50$		5	0	15	20	35	40	55	65

The maximal value of a feasible subset is $V[5, 6] = 65$. The optimal subset is {item 3, item 5}.

b.-c. The instance has a unique optimal subset in view of the following general property: An instance of the knapsack problem has a unique optimal solution if and only if the algorithm for obtaining an optimal subset, which retraces backward the computation of $V[n, W]$, encounters no equality between $V[i - 1, j]$ and $v_i + V[i - 1, j - w_i]$ during its operation.

0/1 Knapsack Problem

The knapsack problem

Let us consider a set of n objects. Each object is characterized by its weight (or dimension - d) and its value (or profit - p). We want to fill in a knapsack of capacity C such that the total value of the selected objects is maximal.

Variants:

- (i) **Continuous variant:** entire objects or part of objects can be selected. The components of the solution are from $[0,1]$.
- (ii) **Discrete variant (0-1):** an object either is entirely transferred into the knapsack or is not transferred. The solution components are from $\{0,1\}$

Question

- a. Write a pseudocode of the bottom-up dynamic programming algorithm for the knapsack problem.
- b. Write a pseudocode of the algorithm that finds the composition of an optimal subset from the table generated by the bottom-up dynamic programming algorithm for the knapsack problem.

Algorithm

```
a. Algorithm DPKnapsack( $w[1..n], v[1..n], W$ )
//Solves the knapsack problem by dynamic programming (bottom up)
//Input: Arrays  $w[1..n]$  and  $v[1..n]$  of weights and values of  $n$  items,
//       knapsack capacity  $W$ 
//Output: Table  $V[0..n, 0..W]$  that contains the value of an optimal
//        subset in  $V[n, W]$  and from which the items of an optimal
//        subset can be found
for  $i \leftarrow 0$  to  $n$  do  $V[i, 0] \leftarrow 0$ 
for  $j \leftarrow 1$  to  $W$  do  $V[0, j] \leftarrow 0$ 
for  $i \leftarrow 1$  to  $n$  do
    for  $j \leftarrow 1$  to  $W$  do
        if  $j - w[i] \geq 0$ 
             $V[i, j] \leftarrow \max\{V[i - 1, j], v[i] + V[i - 1, j - w[i]]\}$ 
        else  $V[i, j] \leftarrow V[i - 1, j]$ 
return  $V[n, W], V$ 
```

Algorithm

```
b. Algorithm OptimalKnapsack( $w[1..n]$ ,  $v[1..n]$ ,  $W$ )
//Finds the items composing an optimal solution to the knapsack problem
//Input: Arrays  $w[1..n]$  and  $v[1..n]$  of weights and values of  $n$  items,
//       knapsack capacity  $W$ , and table  $V[0..n, 0..W]$  generated by
//       the dynamic programming algorithm
//Output: List  $L[1..k]$  of the items composing an optimal solution
 $k \leftarrow 0$  //size of the list of items in an optimal solution
 $j \leftarrow W$  //unused capacity

for  $i \leftarrow n$  downto 1 do
    if  $V[i, j] > V[i - 1, j]$ 
         $k \leftarrow k + 1$ ;  $L[k] \leftarrow i$  //include item  $i$ 
         $j \leftarrow j - w[i]$ 
return  $L$ 
```

The time efficiency and space efficiency of this algorithm are both in $\theta(nW)$.
The time needed to find the composition of an optimal solution is in $O(n)$.

Warshall's Algorithm

Warshall's algorithm for finding the
Transitive closure

Warshall's Algorithm

- Warshall's algorithm is used for computing the transitive closure of a directed graph.

- DEFINITION:

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

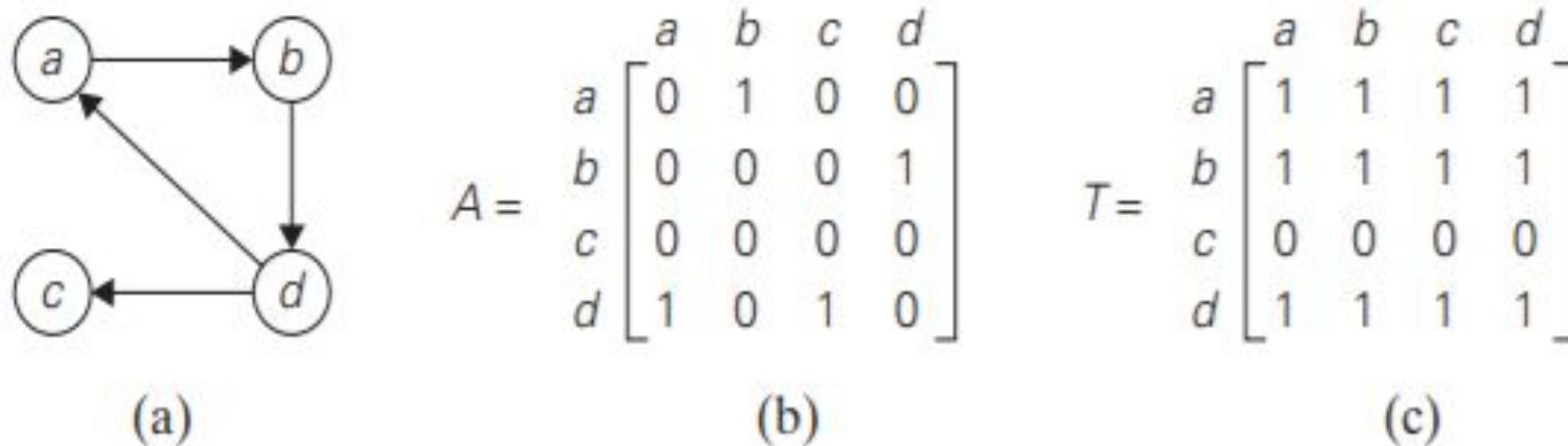


FIGURE 8.11 (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

Warshall's Algorithm

- Warshall's algorithm constructs the transitive closure through a series of $n \times n$ boolean matrices:
- $R^{(0)}, \dots, R^{(k-1)}, R^{(k)}, \dots, R^{(n)}$
- The central point of the algorithm is that we can compute all the elements of each matrix $R^{(k)}$ from its immediate predecessor $R^{(k-1)}$ in the above series.

$$R^{(k-1)} = \begin{matrix} & j & k \\ \begin{matrix} k \\ i \end{matrix} & \begin{bmatrix} & & \\ 1 & & \\ \uparrow 0 & \rightarrow & 1 \end{bmatrix} \end{matrix} \implies R^{(k)} = \begin{matrix} & j & k \\ \begin{matrix} k \\ i \end{matrix} & \begin{bmatrix} & & \\ 1 & & \\ 1 & & 1 \end{bmatrix} \end{matrix}$$

Warshall's Algorithm

- Warshall's algorithm constructs the transitive closure through a series of $n \times n$ boolean matrices:

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

- Each of these matrices provides certain information about directed paths in the digraph. Specifically, the element $r^{(k)}_{ij}$ in the i th row and j th column of matrix $R^{(k)}$ ($i, j = 1, 2, \dots, n, k = 0, 1, \dots, n$) is equal to 1 if and only if there exists a directed path of a positive length from the i th vertex to the j th vertex with each intermediate vertex, if any, numbered not higher than k .
- Thus, the series starts with $R^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $R^{(0)}$ is nothing other than the adjacency matrix of the digraph.
- $R^{(1)}$ contains the information about paths that can use the first vertex as intermediate.

Warshall's Algorithm

- Warshall's algorithm constructs the transitive closure through a series of $n \times n$ boolean matrices:
- $R^{(0)}, \dots, R^{(k-1)}, R^{(k)}, \dots, R^{(n)}$
- In general, each subsequent matrix in the above series has one more vertex to use as intermediate for its paths than its predecessor and hence may, but does not have to, contain more 1's.
- The last matrix in the series, $R^{(n)}$, reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing other than the digraph's transitive closure.

Warshall's Algorithm

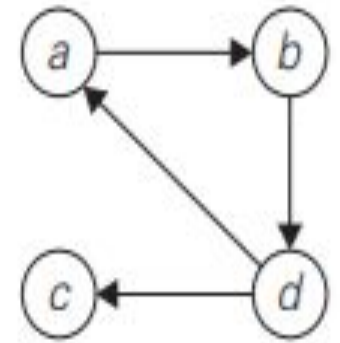
Recurrence relating elements $R^{(k)}$ to elements of $R^{(k-1)}$ is:

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

It implies the following rules for generating $R^{(k)}$ from $R^{(k-1)}$:

- Rule 1 If an element in row i and column j is 1 in $R^{(k-1)}$,
it remains 1 in $R^{(k)}$
- Rule 2 If an element in row i and column j 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 $R^{(k-1)}$

Warshall's Algorithm



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

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

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

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

1's 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 $R^{(2)}$.

$$R^{(2)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{matrix}$$

1's 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 $R^{(3)}$.

$$R^{(3)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{matrix}$$

1's 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 $R^{(4)}$.

$$R^{(4)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{matrix}$$

1's 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

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**

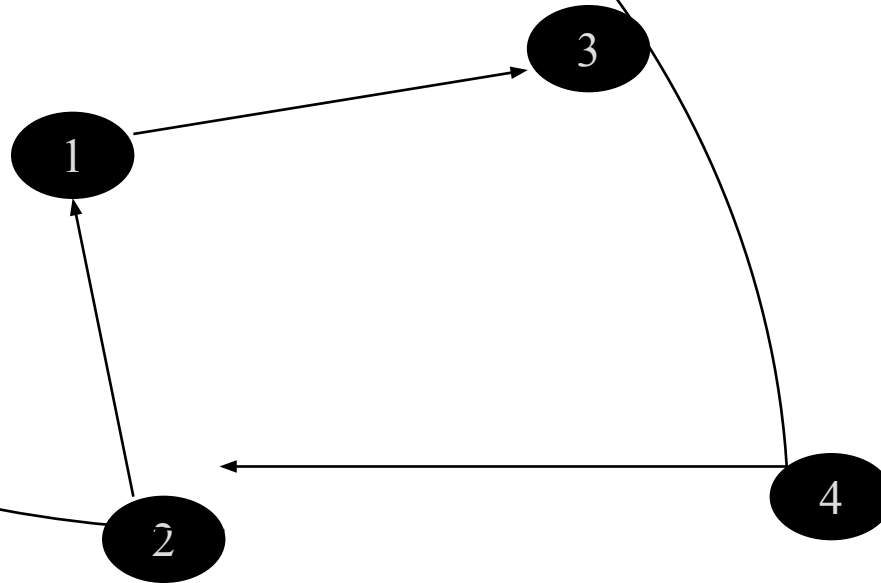
for $j \leftarrow 1$ **to** n **do**

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

return $R^{(n)}$

Time efficiency: $\Theta(n^3)$

Solve



Floyds Algorithm

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

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—i.e., the lengths of the shortest paths—from each vertex to all other vertices.

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

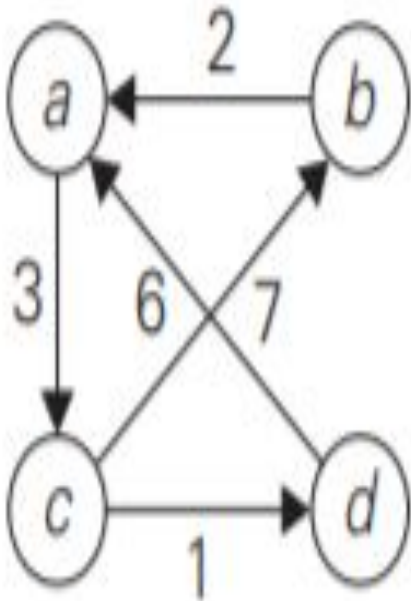
- It is convenient to record the lengths of shortest paths in an $n \times n$ matrix D called the ***distance matrix***: the element d_{ij} in the i th row and the j th column of this matrix indicates the length of the shortest path from the i th vertex to the j th vertex.

Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of $n \times n$ matrices:

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

Each of these matrices contains the lengths of shortest paths with certain constraints on the paths considered for the matrix in question. Specifically, the element $d_{ij}^{(k)}$ in the i th row and the j th column of matrix $D^{(k)}$ ($i, j = 1, 2, \dots, n$, $k = 0, 1, \dots, n$) 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 . In particular, the series starts with $D^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $D^{(0)}$ is simply the weight matrix of the graph. The last matrix in the series, $D^{(n)}$, contains the lengths of the shortest paths among all paths that can use all n vertices as intermediate and hence is nothing other than the distance matrix being sought.

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



(a)

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

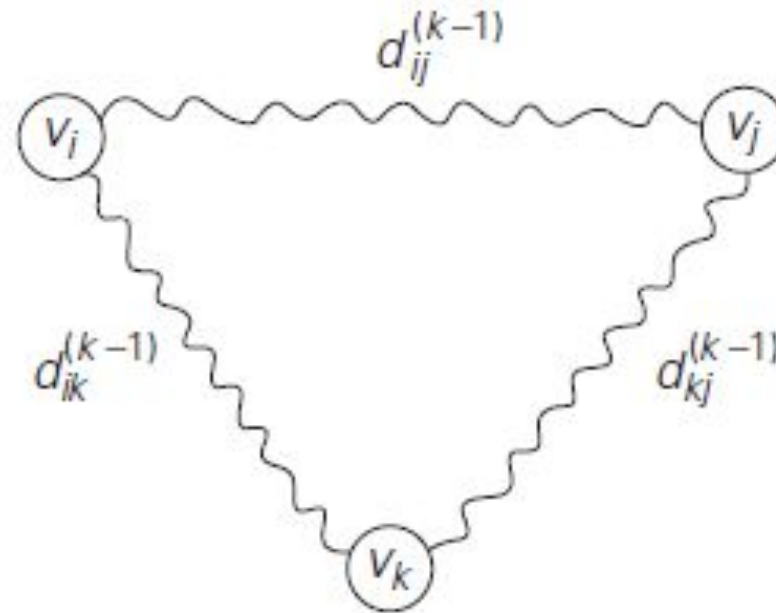
(b)

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

(c)

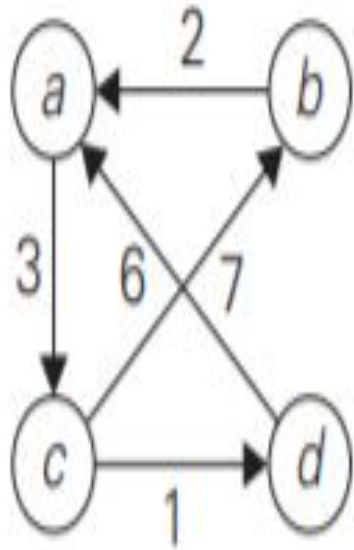
FIGURE 8.14 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

Underlying Idea of Floyd's algorithm



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

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



(a)

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

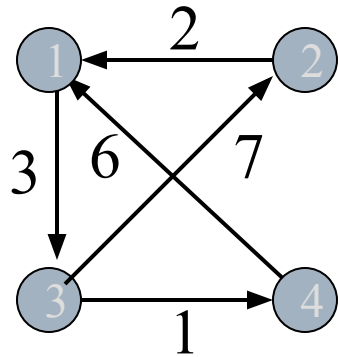
(b)

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

(c)

FIGURE 8.14 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

Floyd's Algorithm $d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$ for $k \geq 1$, $d_{ij}^{(0)} = w_{ij}$.



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

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

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

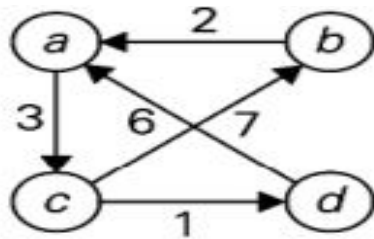
0	10	3	4
2	0	5	6
9	7	0	1
6	16	9	0

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

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

D(3): 3 to 1 not allowing 4=9. D(4): 3 to 1 with allowing 4=7

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



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

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).

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

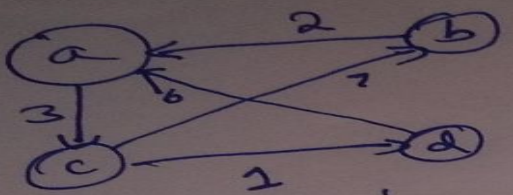
Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e. a , b , and c (note four new shortest paths from a to b , from a to d , from b to d , and from d to b).

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

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).

Example

For this example we should find D^0, D^1, D^2, D^3, D^4
 D^a, D^b, D^c, D^d



$D^0 = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} 1 & a \\ 2 & b \\ 3 & c \\ 4 & d \end{matrix} & \begin{bmatrix} 0 & 2 & 3 & 6 \\ 2 & 0 & \infty & 1 \\ \infty & 1 & 0 & \infty \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{matrix}$

$D^1 = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} 1 & a \\ 2 & b \\ 3 & c \\ 4 & d \end{matrix} & \begin{bmatrix} d_{11} & d_{12} & d_{13} & d_{14} \\ d_{21} & d_{22} & d_{23} & d_{24} \\ d_{31} & d_{32} & d_{33} & d_{34} \\ d_{41} & d_{42} & d_{43} & d_{44} \end{bmatrix} \end{matrix}$

or D^a

$d_{23}^1 = \min \{ d_{23}^0, d_{21}^0 + d_{13}^0 \} = \min \{ \infty, 2 + 3 \} = 5$

$d_{11}^1 = \min \{ d_{11}^0, d_{11}^0 + d_{11}^0 \} = \{ 0, 0 + 0 \} = 0$

$d_{12}^1 = \min \{ d_{12}^0, d_{11}^0 + d_{12}^0 \} = \{ \infty, 0 + \infty \} = \infty$

$d_{13}^1 = \min \{ d_{13}^0, d_{11}^0 + d_{13}^0 \} = \{ 3, 0 + 3 \} = 3$

Step 1: Let us calculate C^a i.e., through vertex a . Because, we are finding the path through a , a^{th} row and a^{th} column need to be considered. So, let us not worry about all other pairs (shown in white color in the figure below) and consider only the pairs with non-zero and non-infinity numbers shown in first column.

C

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

$(b, a) = 2$ and $(a, c) = 3$. So, $(b, c) = \min\{(b, c) \text{ and } (b, a) + (a, c)\}$
 $= \min\{\infty, 2 + 3\} = 5$
 So, $(b, c) = 5$

$(d, a) = 6$ and $(a, c) = 3$. So, $(d, c) = \min\{(d, c) \text{ and } (d, a) + (a, c)\}$
 $= \min\{\infty, 6 + 3\} = 9$
 So, $(d, c) = 9$

0	∞	3	∞
2	0	∞	∞
∞	7	0	1
6	∞	∞	0

All other values remain unaltered. So, $C^a =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	∞	7	0	1
d	6	∞	9	0

Step 2: Let us calculate C^b i.e., through vertex b . Because, we are finding the path through b , b^{th} row and b^{th} column need to be considered. So, let us not worry about all other pairs (shown in white color in the figure below) and consider only the pairs with non-zero and non-infinity numbers shown in second column.

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	∞	7	0	1
d	6	∞	9	0

$(c, b) = 7$ and $(b, a) = 2$ and so $(c, a) = \min\{(c, a) \text{ and } (c, b) + (b, a)\}$
 $= \min\{\infty, 7 + 2\} = 9$
 So, $(c, a) = 9$

$(c, b) = 7$ and $(b, c) = 5$ and so $(c, c) = \min\{(c, c) \text{ and } (c, b) + (b, c)\}$
 $= \min\{0, 7 + 5\} = 0$

$$\text{So, } (c, c) = 0$$

All other values remain unaltered. So, $C^b = \left\{ \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & 5 & \infty \\ c & 9 & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \end{array} \right.$

Step 3: Let us calculate C^c i.e., through vertex c . Because, we are finding the path through c , c^{th} row and c^{th} column need to be considered. So, let us not worry about all other pairs (shown in white color in the figure below) and consider only the pairs with non-zero and non-infinity numbers shown in third column.

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	9	7	0	1
d	6	∞	9	0

$$(a, c) = 3 \text{ and } (c, a) = 9. \text{ So, } (a, a) = \min\{(a, a) \text{ and } (a, c) + (c, a)\} \\ = \min\{0, 3 + 9\} = 0 \\ \text{So, } (a, a) = 0$$

$$(a, c) = 3 \text{ and } (c, b) = 7. \text{ So, } (a, b) = \min\{(a, b) \text{ and } (a, c) + (c, b)\} \\ = \min\{\infty, 3 + 7\} = 10 \\ \text{So, } (a, b) = 10$$

$$(a, c) = 3 \text{ and } (c, d) = 1. \text{ So, } (a, d) = \min\{(a, d) \text{ and } (a, c) + (c, d)\} \\ = \min\{\infty, 3 + 1\} = 4 \\ \text{So, } (a, d) = 4$$

$$(b, c) = 5 \text{ and } (c, a) = 9. \text{ So, } (b, a) = \min\{(b, a) \text{ and } (b, c) + (c, a)\} \\ = \min\{2, 5 + 9\} = 2 \\ \text{So, } (b, a) = 2$$

$$(b, c) = 5 \text{ and } (c, b) = 7. \text{ So, } (b, b) = \min\{(b, b) \text{ and } (b, c) + (c, b)\} \\ = \min\{0, 5 + 7\} = 0 \\ \text{So, } (b, b) = 0$$

$$(b, c) = 5 \text{ and } (c, d) = 1. \text{ So, } (b, d) = \min\{(b, d) \text{ and } (b, c) + (c, d)\} \\ = \min\{\infty, 5 + 1\} = 6 \\ \text{So, } (b, d) = 6$$

$(d, c) = 9$ and $(c, a) = 9$. So, $(d, a) = \min\{(d, a) \text{ and } (d, c) + (c, a)\}$
 $= \min\{6, 9 + 9\} = 6$

So, $(d, a) = 6$

$(d, c) = 9$ and $(c, b) = 7$. So, $(d, b) = \min\{(d, b) \text{ and } (d, c) + (c, b)\}$
 $= \min\{\infty, 9 + 7\} = 16$

So, $(d, b) = 16$

$(d, c) = 9$ and $(c, d) = 1$. So, $(d, d) = \min\{(d, d) \text{ and } (d, c) + (c, d)\}$
 $= \min\{0, 9 + 1\} = 0$

So, $(d, d) = 0$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

Now, C^c is given by:

Step 4: Let us calculate C^d i.e., through vertex d . Because, we are finding the path through d , d^{th} row and d^{th} column need to be considered. So, let us not worry about all other pairs (shown in white color in the figure below) and consider only the pairs with non-zero and non-infinity numbers shown in 4^{th} column.

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

$(a, d) = 4$ and $(d, a) = 6$.

$(a, d) = 4$ and $(d, b) = 16$.

$(a, d) = 4$ and $(d, c) = 9$.

$(b, d) = 6$ and $(d, a) = 6$.

$(b, d) = 6$ and $(d, b) = 16$.

$(b, d) = 6$ and $(d, c) = 9$.

$(c, d) = 1$ and $(d, a) = 6$.

$(c, d) = 1$ and $(d, b) = 16$.

$(c, d) = 1$ and $(d, c) = 9$.

So, $(a, a) = \min\{(a, a), 4 + 6\} = 0$

So, $(a, b) = \min\{(a, b), 4 + 16\} = 10$

So, $(a, c) = \min\{(a, c), 4 + 9\} = 3$

So, $(b, a) = \min\{(b, a), 6 + 6\} = 2$

So, $(b, b) = \min\{(b, b), 6 + 16\} = 0$

So, $(b, c) = \min\{(b, c), 6 + 9\} = 5$

So, $(c, a) = \min\{(c, a), 1 + 6\} = 7$

So, $(c, b) = \min\{(c, b), 1 + 16\} = 7$

So, $(c, c) = \min\{(c, c), 1 + 9\} = 0$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	7	7	0	1
d	6	16	9	0

Now, C^d is given by:

The final matrix C^d gives the shortest distances from all nodes to all other nodes.

Algorithm

- Time efficiency of Floyd's algorithm is cubic

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**

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

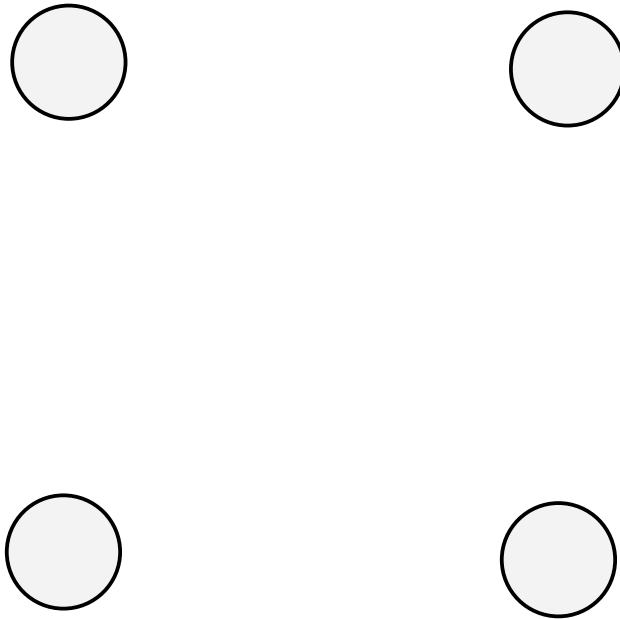
return D

Time efficiency: $\Theta(n^3)$

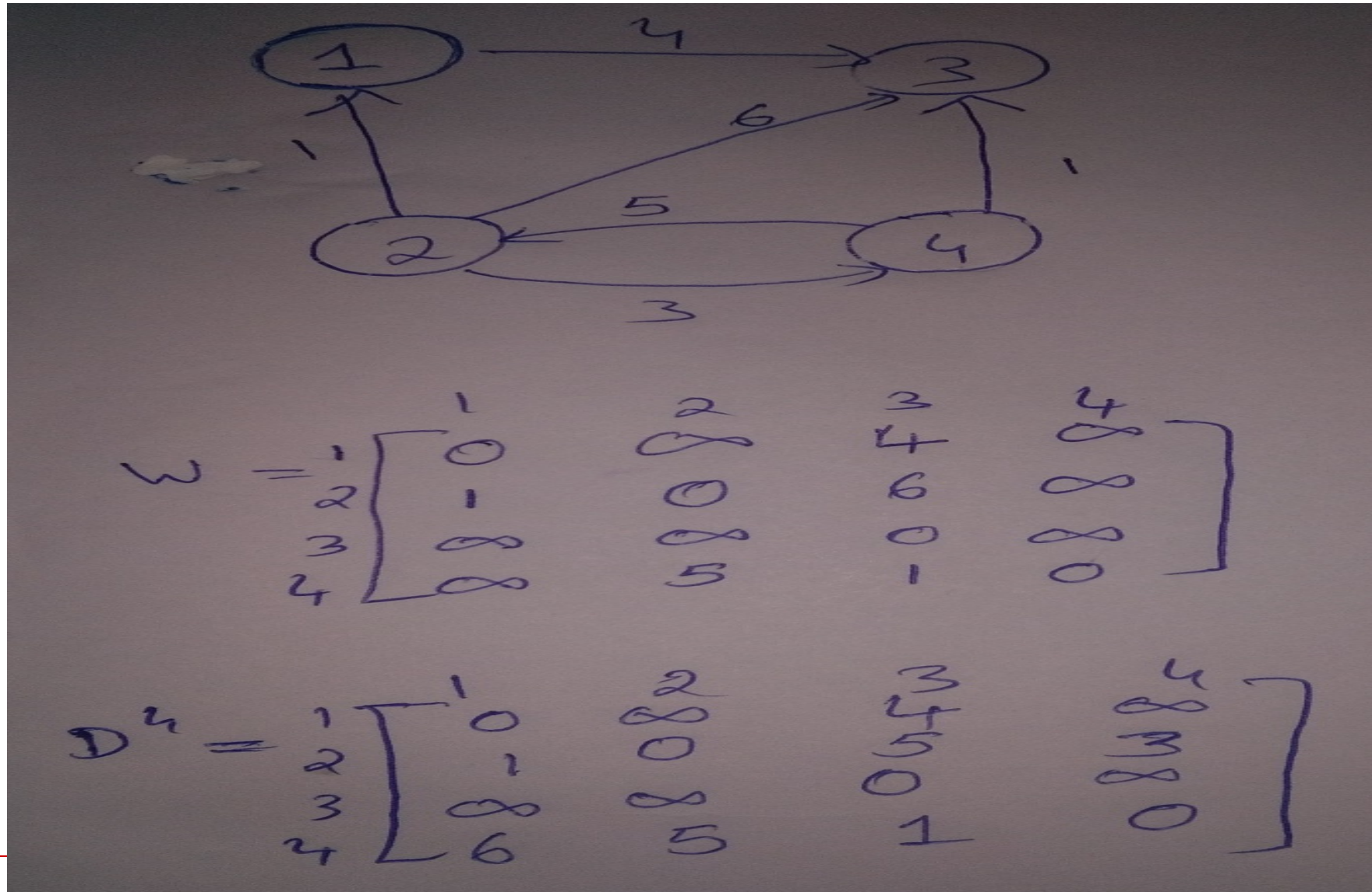
Lab Programs

11. Implement Warshall's algorithm using dynamic programming.
12. Implement 0/1 Knapsack problem using dynamic programming.
13. Implement All Pair Shortest paths problem using Floyd's algorithm.

Solve using Floyd's Algorithm



Solve using Floyd's Algorithm



Thanks for Listening

-
- Coin-Row Problem: <https://www.youtube.com/watch?v=mSyiRGSAq7k>
 - Change-Making Problem: <https://www.youtube.com/watch?v=jgiZlGzXMBw>
 - Coin-Collection Problem: https://www.youtube.com/watch?v=94FEC_uNwVM