Parul® University

NAAC A++

Dynamic Programming:

Chapter-4

Mrs. Bhumi Shah

Assistant Professor

Department of Computer Science and Engineering

Content

1. Principal of Optimality:
- 0/1 Knapsack Problem,
- Making Change Problem.
2. Chain matrix multiplication,
   Longest Common Subsequence.
3. All pair shortest paths.

INDEX

Principal of Optimality

- A dynamic-programming algorithm solves every sub-problem just once and then **saves its answer in a table**.
- It avoids the work of **re-computing the answer** every time the sub problem is encountered.
- The dynamic programming algorithm obtains the solution using **principle of optimality**.
- The principle of optimality states that “in an optimal sequence of decisions or choices, each subsequence must also be optimal.
- If it is **not possible to apply the principle of optimality** then it is almost **impossible to obtain the solution** using the dynamic programming approach.

Principal of Optimality

- A dynamic-programming algorithm solves every sub-problem just once and then **saves its answer in a table**.
- It avoids the work of **re-computing the answer** every time the sub problem is encountered.
- The dynamic programming algorithm obtains the solution using **principle of optimality**.
- The principle of optimality states that “in an optimal sequence of decisions or choices, each subsequence must also be optimal.
- If it is **not possible to apply the principle of optimality** then it is almost **impossible to obtain the solution** using the dynamic programming approach.

0/1 Knapsack Problem - Dynamic Programming Solution

- Solve the following knapsack problem using dynamic programming technique.

1. $W = 11$ and $n = 5$

Object i	1	2	3	4	5
v_i	1	6	18	22	28
w_i	1	2	5	6	7

0/1 Knapsack Problem - Dynamic Programming Solution

- We need to generate table $V(1 \dots n, 0 \dots W)$

1. where, $n =$ number of objects.
2. $W =$ capacity of knapsack.

To generate table $V[i][j]$ use following steps:

Step-1: Make $V[i][0] = 0 \; for \; 0 < i \leq n$

Step-2: if $j < w_i$ then

$$V[i][j] = V[i-1][j]$$

Step-3: if $j \geq w_i$ then

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

0/1 Knapsack Problem - Dynamic Programming Solution

Problem Statement − A thief is robbing a store and can carry a maximal weight of ***W*** into his knapsack. There are ***n*** items and weight of i^{th} item is wi and the profit of selecting this item is ***pi***. What items should the thief take?

Let i be the highest-numbered item in an optimal solution **S** for **W** dollars. Then **S = S {i}** is an optimal solution for **W** **w_i** dollars and the value to the solution S is V_i plus the value of the sub-problem.

We can express this fact in the following formula: define **c[i, w]** to be the solution for items **1,2, , i** and the maximum weight **w**.

0/1 Knapsack Problem - Dynamic Programming Solution

The algorithm takes the following inputs:

- The maximum weight W
- The number of items n
- The two sequences v = <v1, v2, , vn> and w = <w1, w2, , wn>

The set of items to take can be deduced from the table, starting at c[n, w] and tracing backwards where the optimal values came from.

If c[i, w] = c[i-1, w], then item i is not part of the solution, and we continue tracing with c[i-1, w]. Otherwise, item i is part of the solution, and we continue tracing with c [i-1, w-W].

0/1 Knapsack Problem - Algorithm

```
Dynamic-0-1-knapsack (v, w, n, W)
for w = 0 to W do
  c[0, w] = 0
for i = 1 to n do
  c[i, 0] = 0
  for w = 1 to W do
    if wi  w then
      if vi + c[i-1, w-wi] then
        c[i, w] = vi + c[i-1, w-wi]
      else c[i, w] = c[i-1, w]
    else
      c[i, w] = c[i-1, w]
```

0/1 Knapsack Problem - Example

Let us consider that the capacity of the knapsack is W = 8 and the items are as shown in the following table.

Item	A	B	C	D
Profit	2	4	7	10
Weight	1	3	5	7

Solution

Using the greedy approach of 0-1 knapsack, the weight that's stored in the knapsack would be A+B = 4 with the maximum profit 2 + 4 = 6. But, that solution would not be the optimal solution.

Therefore, dynamic programming must be adopted to solve 0-1 knapsack problems.

0/1 Knapsack Problem - Example

Let us consider that the capacity of the knapsack is W = 8 and the items are as shown in the following table.

Item	A	B	C	D
Profit	2	4	7	10
Weight	1	3	5	7

Solution

Using the greedy approach of 0-1 knapsack, the weight that's stored in the knapsack would be A+B = 4 with the maximum profit 2 + 4 = 6. But, that solution would not be the optimal solution.

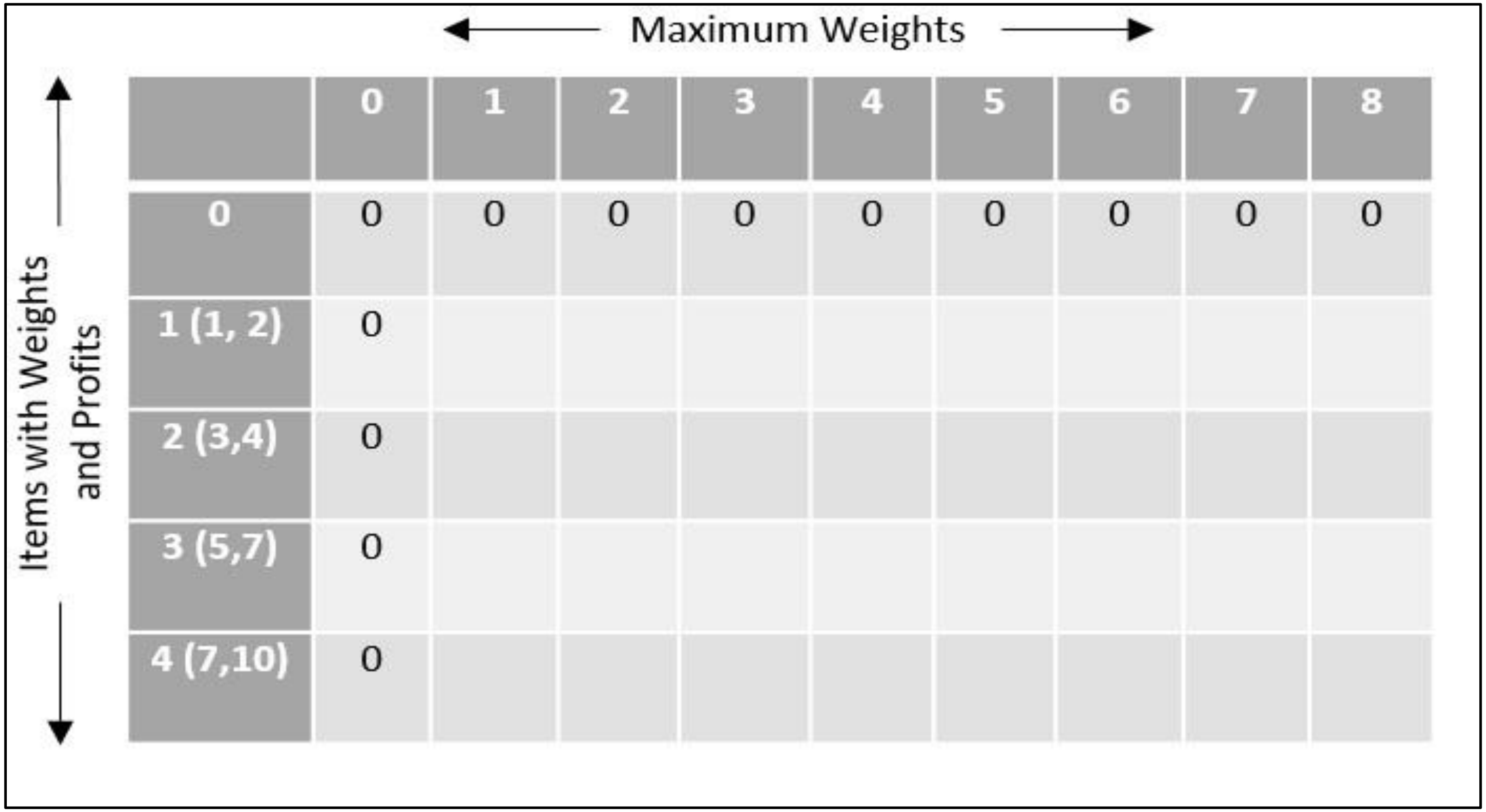
Therefore, dynamic programming must be adopted to solve 0-1 knapsack problems.

0/1 Knapsack Problem - Example

Step 1

- Construct an adjacency table with maximum weight of knapsack as rows and items with respective weights and profits as columns.
- Values to be stored in the table are cumulative profits of the items whose weights do not exceed the maximum weight of the knapsack (designated values of each row)
- So we add zeroes to the 0th row and 0th column because if the weight of item is 0, then it weighs nothing; if the maximum weight of knapsack is 0, then no item can be added into the knapsack.

0/1 Knapsack Problem - Example

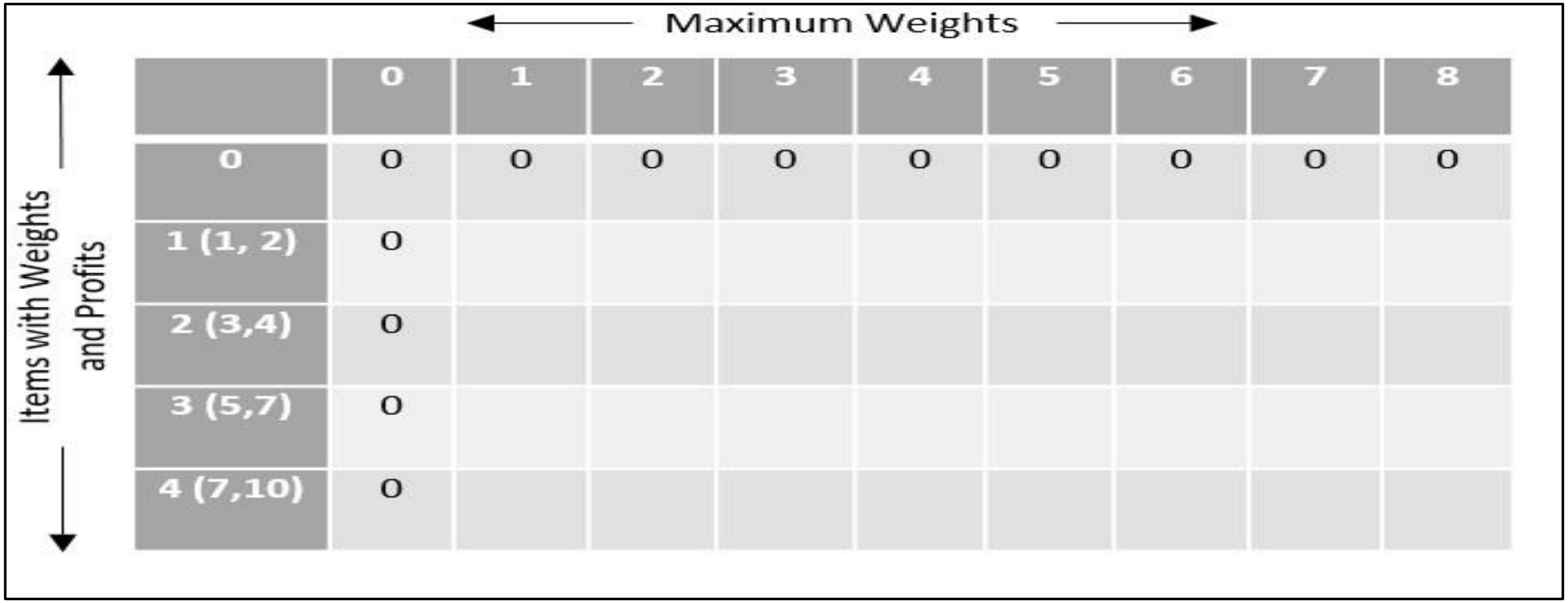


Maximum Weights

Items with Weights and Profits

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 (1, 2)	0								
2 (3,4)	0								
3 (5,7)	0								
4 (7,10)	0								

0/1 Knapsack Problem - Example



Maximum Weights

Items with Weights and Profits

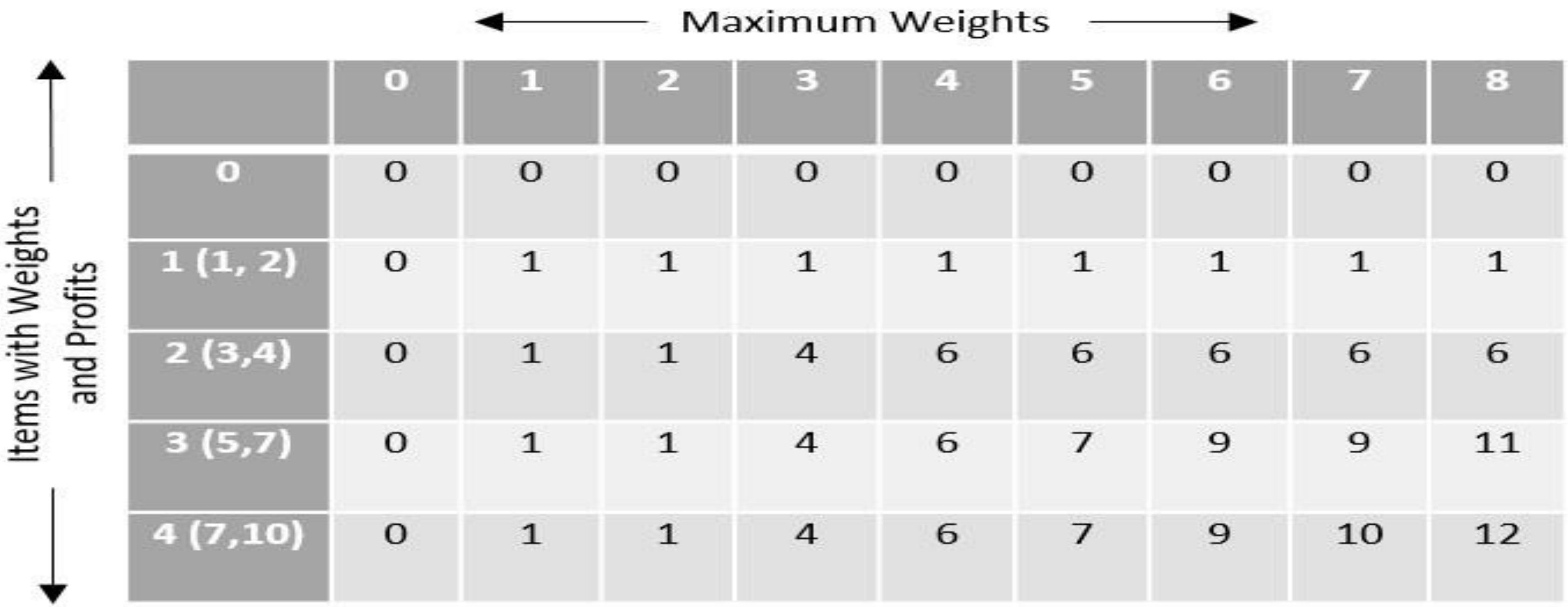
	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 (1, 2)	0								
2 (3,4)	0								
3 (5,7)	0								
4 (7,10)	0								

he remaining values are filled with the maximum profit achievable with respect to the items and weight per column that can be stored in the knapsack.

The formula to store the profit values is −

c[i,w]=max{c[i−1,w−w[i]]+P[i]}

0/1 Knapsack Problem - Example



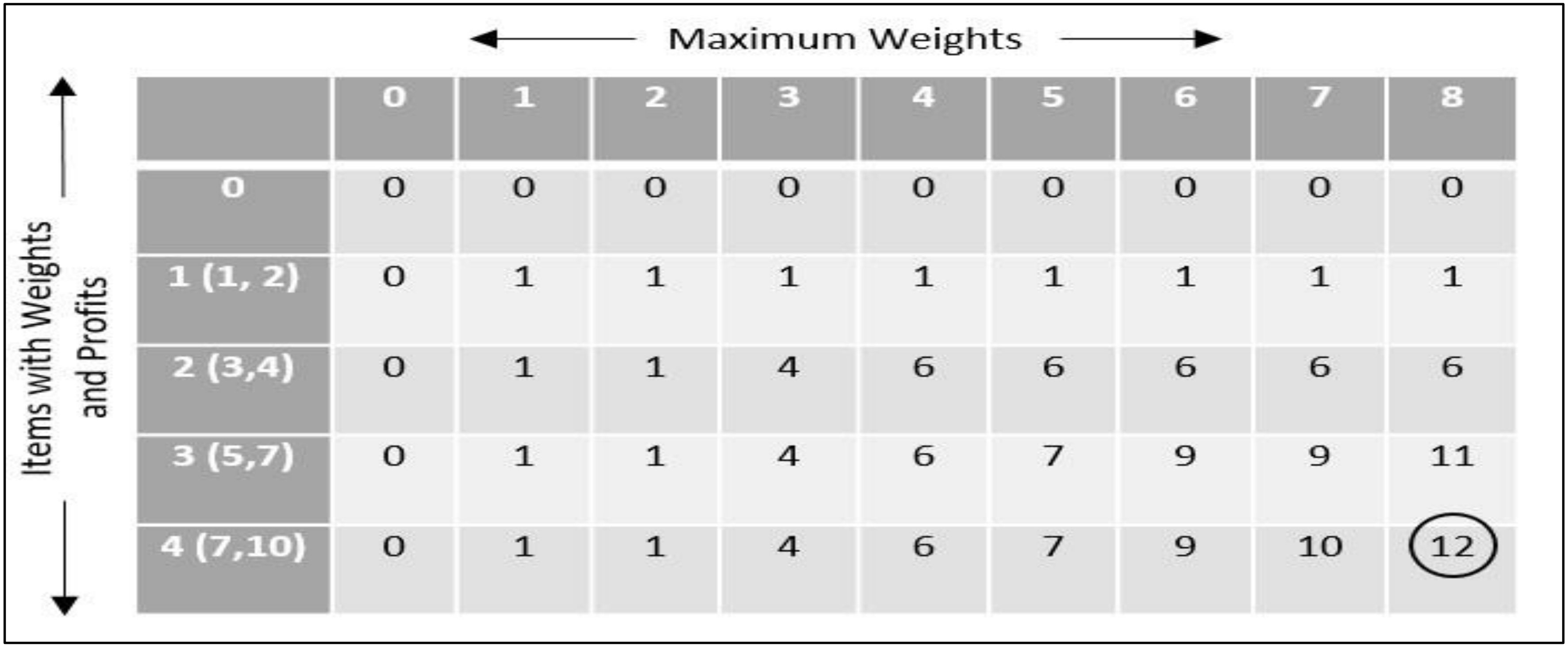
Maximum Weights

Items with Weights and Profits

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 (1, 2)	0	1	1	1	1	1	1	1	1
2 (3,4)	0	1	1	4	6	6	6	6	6
3 (5,7)	0	1	1	4	6	7	9	9	11
4 (7,10)	0	1	1	4	6	7	9	10	12

To find the items to be added in the knapsack, recognize the maximum profit from the table and identify the items that make up the profit, in this example, its {1, 7}.

0/1 Knapsack Problem - Example



Maximum Weights

Items with Weights and Profits

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 (1, 2)	0	1	1	1	1	1	1	1	1
2 (3,4)	0	1	1	4	6	6	6	6	6
3 (5,7)	0	1	1	4	6	7	9	9	11
4 (7,10)	0	1	1	4	6	7	9	10	12

The optimal solution is {1, 7} with the maximum profit is 12.

Making Change problem

- We need to generate a table $c[n][N]$, where

1. n = number of denominations
2. N= amount for which you need to make a change.

To generate table $\boldsymbol{c}[\boldsymbol{i}][\boldsymbol{j}]$ use following steps:

Step-1: Make $c[i][0] = 0 \; for \; 0 < i \leq n$

Repeat step-2 to step-4 for the remaining matrix values

Step-2: If $i = 1$ then $c[i][j] = 1 + c[1][j - d_1]$

Step-3: If $j < d_i$ then $c[i][j] = c[i-1][j]$

Step-4: Otherwise $c[i][j] = min(c[i-1][j], 1 + c[i][j - d_i])$

Optimal Sub-structure

Making Change problem

Denominations: $d_1 = 1, d_2 = 4, d_3 = 6.$ Make a change of Rs. 8.

Step-1: Make $\underline{c[i][0] = 0}\ for\ 0 < i \leq n$

$j \longrightarrow$ Amoun

$i = 1$										
$i = 2$						1	2			
$i = 3$										

$min(c[1][4], 1 + c[2][0]) = min(4,1 + 0) = min(4,1) = 1$

$min(c[1][5], 1 + c[2][1]) = min(5,1 + 1) = min(5,2) = 2$

Making Change problem

- Denominations: $d_1 = 1, d_2 = 4, d_3 = 6$. Make a change of Rs. 8.

Step-1: Make $c[i][0] = 0\ for\ 0 < i \leq n$

Step-2: If $i = 1$ then $c[i][j] = 1 + c[1][j - d_1]$, here $d_1 = 1$

Step-3: If $j < d_i$ then $c[i][j] = c[i-1][j]$

Step-4: Otherwise $c[i][j] = min(c[i-1][j], 1 + c[i][j - d_i])$

$j \longrightarrow$ Amount

		0	1	2	3	4	5	6	7	8
$i = 1$	$d_1 = 1$	0	1	2	3	4	5	6	7	8
$i = 2$	$d_2 = 4$	0	1	2	3	1	2			
$i = 3$	$d_3 = 6$	0								

Making Change problem

- We can also find the coins to be included in the solution set as follows:

1. Start looking at c[3, 8] = c[2, 8] ⟹ So, **not to include** a coin with denomination 6.
2. Next go to c[2,8] ≠ c[1, 8] but c[2,8] = 1 + c[2,4]
   - So, include a coin with denomination 4
3. Now, got to c[2,4] ≠ c[1,4] but c[2,4] = 1+ c[2,0]
   - So, again include a coin with denomination 4
4. Go to c[2,0] = c[1,0] and stop.

$$c[i][j] = min(c[i-1][j], \mathbf{1 + c[i][j-d_i]})$$

	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

Solution contains 2 coins with denomination 4

Parul® University | NAAC GRADE A++

https://paruluniversity.ac.in/

Parul® University

NAAC A++

Dynamic Programming:

Chapter-4

Mrs. Bhumi Shah

Assistant Professor

Department of Computer Science and Engineering

Content

1. Principal of Optimality:
- 0/1 Knapsack Problem,
- Making Change Problem.

2. Chain matrix multiplication,
   Longest Common Subsequence.

3. All pair shortest paths.

INDEX

Matrix Chain Multiplication Algorithm

- Matrix Chain Multiplication is an algorithm that is applied to determine the lowest cost way for multiplying matrices.
- The actual multiplication is done using the standard way of multiplying the matrices, i.e., it follows the basic rule that the number of rows in one matrix must be equal to the number of columns in another matrix.
- Hence, multiple scalar multiplications must be done to achieve the product.
- To brief it further, consider matrices A, B, C, and D, to be multiplied; hence, the multiplication is done using the standard matrix multiplication. There are multiple combinations of the matrices found while using the standard approach since matrix multiplication is associative

Matrix Chain Multiplication Algorithm

For instance, there are five ways to multiply the four matrices given above –

- (A(B(CD)))
- (A((BC)D))
- ((AB)(CD))
- ((A(BC))D)
- (((AB)C)D)

Matrix Chain Multiplication Algorithm

- Now, if the size of matrices A, B, C, and D are **l m, m n, n p, p q** respectively, then the number of scalar multiplications performed will be ***lmnpq***. But the cost of the matrices change based on the rows and columns present in it. Suppose, the values of l, m, n, p, q are 5, 10, 15, 20, 25 respectively, the cost of (A(B(CD))) is 5 100 25 = 12,500; however, the cost of (A((BC)D)) is 10 25 37 = 9,250.
- So, **dynamic programming approach of the matrix chain multiplication is adopted in order to find the combination with the lowest cost.**

Matrix Chain Multiplication Algorithm

Count the number of parenthesizations. Find the number of ways in which the input matrices can be multiplied using the formulae −

$$P(n) = \begin{cases} 1 & if\ n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & if\ n \geq 2 \end{cases}$$

(or)

$$P(n) = \begin{cases} \frac{2(n-1)C_{n-1}}{n} & if\ n \geq 2 \\ 1 & if\ n = 1 \end{cases}$$

•Once the parenthesization is done, the optimal substructure must be devised as the first step of dynamic programming approach **so the final product achieved is optimal**. In matrix chain multiplication, the optimal substructure is found by dividing the sequence of matrices ***A[i.j]*** into two parts ***A[i,k] and A[k+1,j]***. It must be ensured that the parts are divided in such a way that optimal solution is achieved.

Matrix Chain Multiplication Algorithm

Using the formula,

$$C[i,j] = \begin{cases} 0 & if\, i = j \\ \min\limits_{i<k<j} \{C[i,k] + C[k+1,j] + d_{i-1}d_kd_j & if\, i < j \end{cases}$$

find

- The lowest cost parenthesization of the sequence of matrices by constructing cost tables and corresponding ***k*** values table.
- Once the lowest cost is found, print the corresponding parenthesization as the output.

Pseudocode to find the lowest cost of all the possible parenthesizations –

MATRIX-CHAIN-MULTIPLICATION(p)**the lowest cost parenthesization of the sequence of matrices by constructing cost tables and corresponding k values table.**
Once the lowest cost is found, print the corresponding parenthesization as the output.

```
  n = p.length  1
  let m[1n, 1n] and s[1n  1, 2n] be new matrices
  for i = 1 to n
    m[i, i] = 0
  for l = 2 to n // l is the chain length
    for i = 1 to n - l + 1
      j = i + l - 1
      m[i, j] = ∞
      for k = i to j - 1
        q = m[i, k] + m[k + 1, j] + pi-1pkpj
        if q < m[i, j]
          m[i, j] = q
          s[i, j] = k
return m and s
```

Pseudocode to print the optimal output parenthesizing –

```
PRINT-OPTIMAL-OUTPUT(s, i, j )
    if i == j
        print Ai
    else print (
        PRINT-OPTIMAL-OUTPUT(s, i, s[i, j])
        PRINT-OPTIMAL-OUTPUT(s, s[i, j] + 1, j)
        print )
```

Example

A sequence of matrices A, B, C, D with dimensions 5 10, 10 15, 15 20, 20 25 are set to be multiplied. Find the lowest cost parenthesization to multiply the given matrices using matrix chain multiplication.
Find the count of parenthesization of the 4 matrices, i.e. n = 4.

Using the formula, P(n)={1∑n−1k=1P(k)P(n−k)ifn=1ifn≥2

Since n = 4 2, apply the second case of the formula −

P(n)=∑k=1n−1P(k)P(n−k)

P(4)=∑k=13P(k)P(4−k)

P(4)=P(1)P(3)+P(2)P(2)+P(3)P(1)

Example

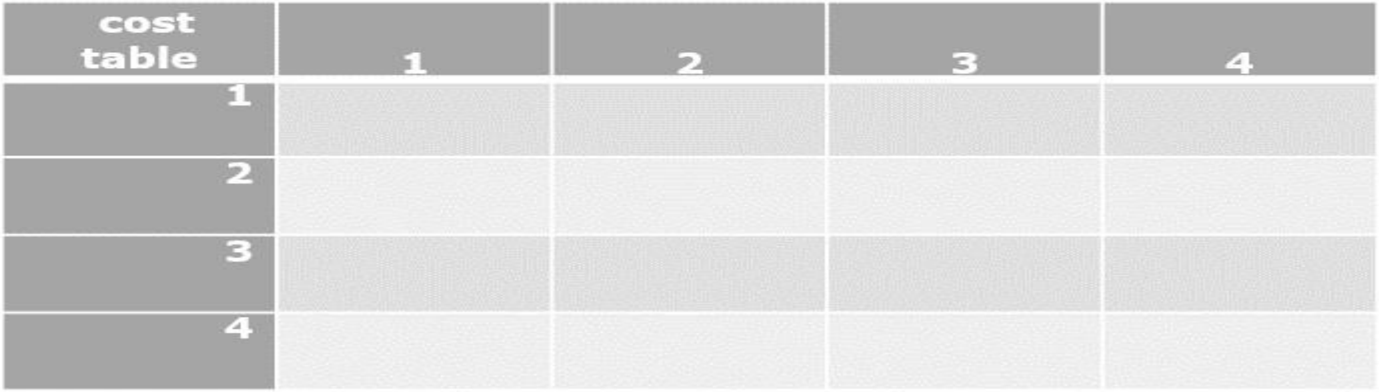
If $P(1) = 1$ and $P(2)$ is also equal to 1, $P(4)$ will be calculated based on the $P(3)$ value. Therefore, $P(3)$ needs to determined first.

$P(3)=P(1)P(2)+P(2)P(1)$
$=1+1=2$
Therefore,
$P(4)=P(1)P(3)+P(2)P(2)+P(3)P(1)$
$=2+1+2=5$
Among these 5 combinations of parenthesis, the matrix chain multiplication algorithm must find the lowest cost parenthesis.

Example

Step 1

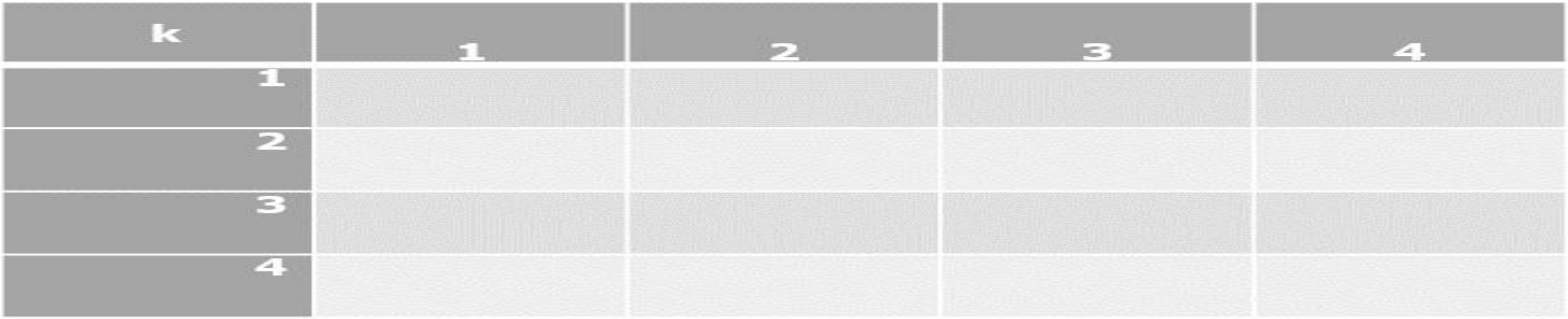
The table above is known as a **cost table**, where all the cost values calculated from the different combinations of parenthesis are stored.



cost table	1	2	3	4
1				
2				
3				
4				

Another table is also created to store the **k** values obtained at the minimum cost of each combination.

Example



k	1	2	3	4
1				
2				
3				
4				

Step 2

Applying the dynamic programming approach formula find the costs of various parenthesizations,

$$C[i,j] = \begin{cases} 0 & if\, i = j \\ \min\limits_{i \le k < j} \{ C[i,k] + C\,[k+1, j] + d_{i-1} d_k d_j & if\, i < j \end{cases}$$

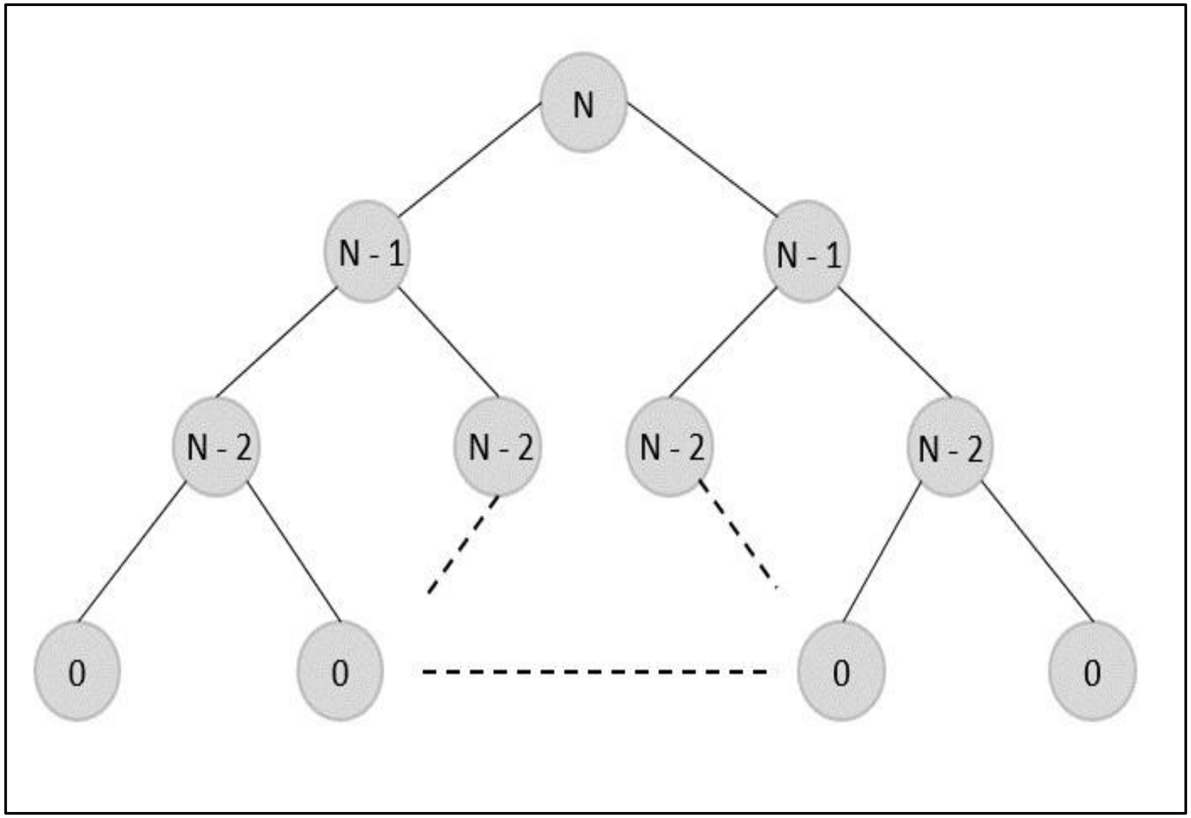
Example

C[1,1]=0

C[2,2]=0

C[3,3]=0

C[4,4]=0



Example

Step 3

Applying the dynamic approach formula only in the upper triangular values of the cost table, since i < j always.

$$C[1,2] = \min_{1 \le k < 2} \{ C[1,1] + C[2,2] + d_0 d_1 d_2 \}$$

- $C[1,2] = 0 + 0 + (5 \times 10 \times 15)$
- $C[1,2] = 750$

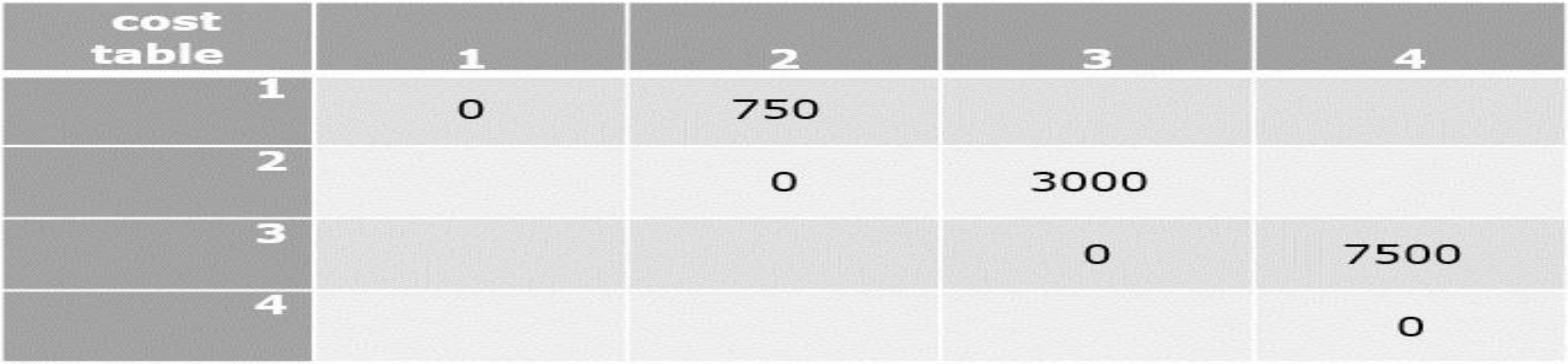
$$C[2,3] = \min_{2 \le k < 3} \{ C[2,2] + C[3,3] + d_1 d_2 d_3 \}$$

- $C[2,3] = 0 + 0 + (10 \times 15 \times 20)$
- $C[2,3] = 3000$

$$C[3,4] = \min_{3 \le k < 4} \{ C[3,3] + C[4,4] + d_2 d_3 d_4 \}$$

- $C[3,4] = 0 + 0 + (15 \times 20 \times 25)$
- $C[3,4] = 7500$

Example



cost table	1	2	3	4
1	0	750		
2		0	3000	
3			0	7500
4				0

Step 4

Find the values of [1, 3] and [2, 4] in this step. The cost table is always filled diagonally step-wise.

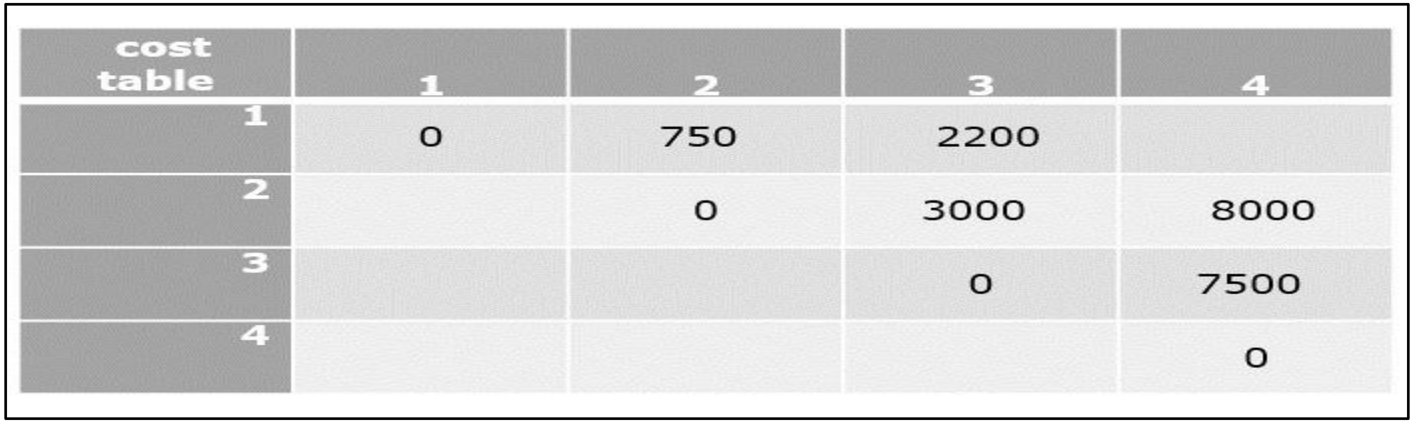
$$C[2,4] = \min_{2 \leq k < 4} \{ C[2,2] + C[3,4] + d_1 d_2 d_4, C[2,3] + C[4,4] + d_1 d_3 d_4 \}$$

Example

- C[2,4]=min{(0+7500+(10×15×20)),(3000+5000)}C[2,4]=min{(0+7500+(10×15×20)),(3000+5000)}
- C[2,4]=8000

$$C[1,3] = \min_{1\leq k<3} \{ C[1,1] + C[2,3] + d_0 d_1 d_3, C[1,2] + C[3,3] + d_0 d_2 d_3 \}$$

- C[1,3]=min{(0+3000+1000),(1500+0+750)}C[1,3]=min{(0+3000+1000),(1500+0+750)}
- C[1,3]=2250



cost table	1	2	3	4
1	0	750	2200	
2		0	3000	8000
3			0	7500
4				0

Example

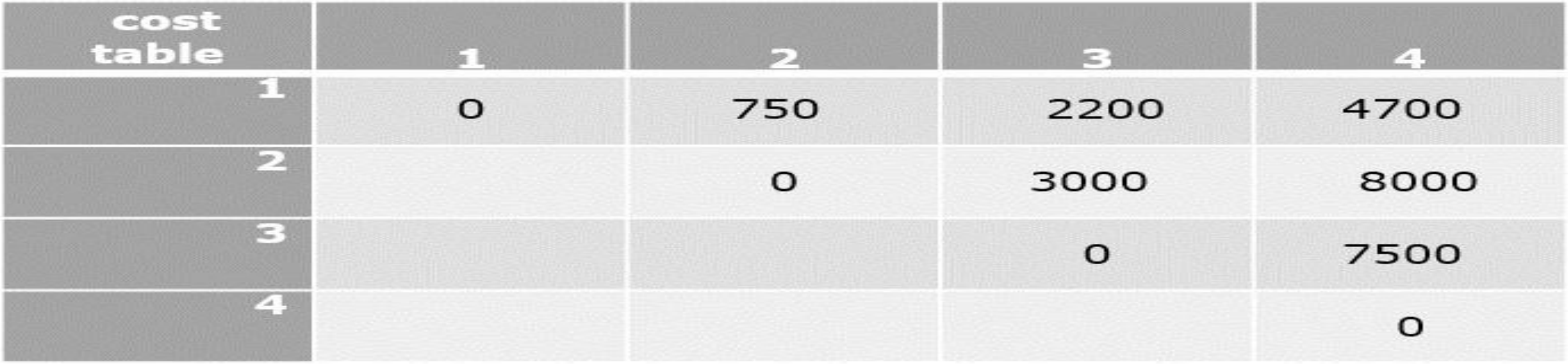
Step 5

Now compute the final element of the cost table to compare the lowest cost parenthesization.

$$C[1,4] = \min_{1\le k<4} \left\{ C[1,1]+C[2,4]+d_0d_1d_4, C[1,2]+C[3,4]+d_1d_2d_4, C[1,3]+C[4,4]+d_1d_3d_4 \right\}$$

- C[1,4]=min{0+8000+1250,750+7500+1875,2200+0+2500}C[1,4]=min{0+8000+1250,750+7500+1875,2200+0+2500}
- C[1,4]=4700

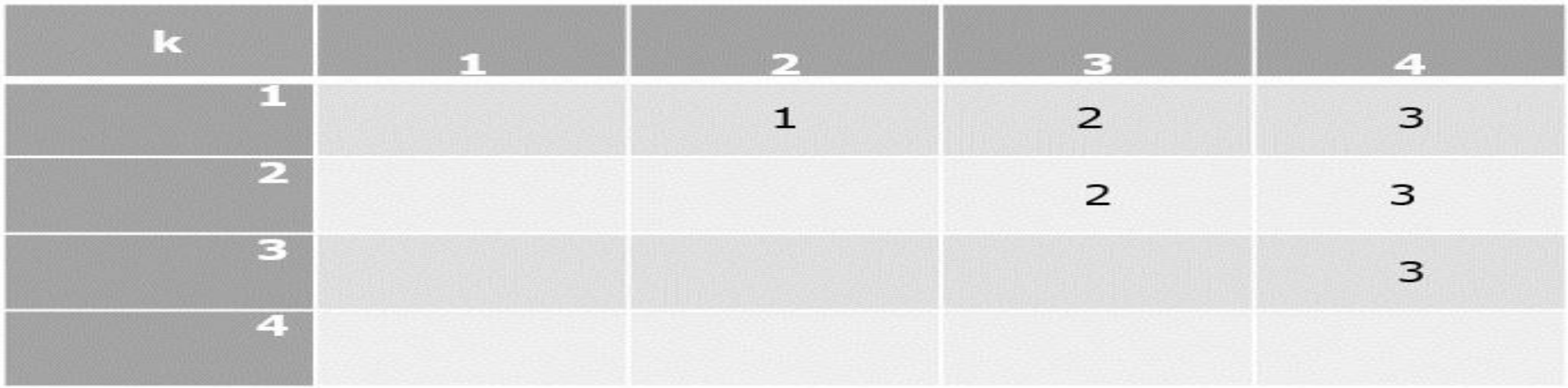
Example



cost table	1	2	3	4
1	0	750	2200	4700
2		0	3000	8000
3			0	7500
4				0

Now that all the values in cost table are computed, the final step is to parethesize the sequence of matrices. For that, k table needs to be constructed with the minimum value of k corresponding to every parenthesis.

Example



k	1	2	3	4
1		1	2	3
2			2	3
3				3
4				

Parenthesization

Based on the lowest cost values from the cost table and their corresponding k values, let us add parenthesis on the sequence of matrices.

The lowest cost value at [1, 4] is achieved when k = 3, therefore, the first parenthesization must be done at 3.

Example

(ABC)(D)

The lowest cost value at [1, 3] is achieved when k = 2, therefore the next parenthesization is done at 2.

((AB)C)(D)

The lowest cost value at [1, 2] is achieved when k = 1, therefore the next parenthesization is done at 1. But the parenthesization needs at least two matrices to be multiplied so we do not divide further.

((AB)(C))(D)

Since, the sequence cannot be parenthesized further, the final solution of matrix chain multiplication is ((AB)C)(D).

Longest Common Subsequence

- A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.
- Given two sequences $\boldsymbol{X}$ and $\boldsymbol{Y}$, we say that a sequence $\boldsymbol{Z}$ is a common subsequence of $\boldsymbol{X}$ and $\boldsymbol{Y}$ if $\boldsymbol{Z}$ is a subsequence of both $\boldsymbol{X}$ and $\boldsymbol{Y}$.
- E.g., if $\boldsymbol{X}$ = <$\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, $\boldsymbol{B}$, $\boldsymbol{D}$, $\boldsymbol{A}$, $\boldsymbol{B}$> and $\boldsymbol{Y}$= <$\boldsymbol{B}$, $\boldsymbol{D}$, $\boldsymbol{C}$, $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{A}$> then <$\boldsymbol{B}$, $\boldsymbol{C}$, $\boldsymbol{A}$> is a subsequence.
- Use dynamic programming technique to find the longest common subsequence (LCS).

Longest Common Subsequence

- We need to generate table $\boldsymbol{c(1..m, 1..n)}$

  where $\boldsymbol{m}$ = length of string $\boldsymbol{S1}$ and $\boldsymbol{n}$ = length of string $\boldsymbol{S2}$.

To generate table $\boldsymbol{c[i][j]}$ use following steps:

Step-1: Make $\boldsymbol{c[i][0] = 0}$ and $\boldsymbol{c[0][j] = 0}$

Step-2: if $\boldsymbol{x_i = y_j}$ then $\boldsymbol{c[i, j] \leftarrow c[i-1, j-1] + 1}$

Step-3: else $\boldsymbol{c[i, j] \leftarrow \max(c[i-1, j], c[i, j-1])}$

Longest Common Subsequence

if $x_i = y_j$ **then** $c[i,j] \leftarrow c[i-1,j-1]+1$ **else** $c[i,j] \leftarrow \max(c[i-1,j], c[i,j-1])$

	y_j	$\boxed{A}$	B	C	B	D	A	B
x_i		○						
$\boxed{B}$	○	↑	↖	←	↖	←	←	↖
D								
C								
A								
B								
A								

Longest Common Subsequence

$if\ \ x_i = y_j$ **then** $c[i,j] \leftarrow c[i-1, j-1] + 1$ **else** $c[i,j] \leftarrow$ **max**$(c[i-1,j], c[i,j-1])$

	y_j	$\boxed{A}$	B	C	B	D	A	B
x_i	0	0	0	0	0	0	0	0
$\boxed{B}$	0	↑ 0	↖ 1	← 1	↖ 1	← 1	← 1	↖ 1
D	0	↑	↑	↑	↑	↖	←	←
C	0							
A	0							
B	0							
A								

Longest Common Subsequence

$if\ \ x_i = y_j$ **then** $c[i,j] \leftarrow c[i-1, j-1] + 1$ **else** $c[i,j] \leftarrow$ **max(**$c[i-1, j]$, $c[i, j-1]$**)**

	y_j	$\boxed{A}$	B	C	B	D	A	B
x_i	0	0	0	0	0	0	0	0
B	0	↑ 0	↖ 1	← 1	↖ 1	← 1	← 1	↖ 1
$\boxed{D}$	0	↑ 0	↑ 1	↑ 1	↑ 1	↖ 2	← 2	← 2
C	0	↑	↑	↖	←	↑	↑	↑
A	0	↖	↑	↑	↑	↑	↖	←
B	0	↑	↖	↑	↖	←	↑	↖
A	0	↖	↑	↑	↑	↑	↖	↑

Longest Common Subsequence

$X = < A, B, C, B, D, A, B >$ and $Y = < B, D, C, A, B, A >$ **and LCS =** $< B, D, A, B >$

	y_j	A	B	C	$\boxed{B}$	$\boxed{D}$	$\boxed{A}$	$\boxed{B}$
x_i	0	0	0	0	0	0	0	0
B	0	↑ 0	↖ 1	← 1	↖ 1	← 1	← 1	↖ 1
D	0	↑ 0	↑ 1	↑ 1	↑ 1	↖ 2	← 2	← 2
C	0	↑ 0	↑ 1	↖ 2	← 2	↑ 2	↑ 2	↑ 2
A	0	↖ 1	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3
B	0	↑ 1	↖ 2	↑ 2	↖ 3	← 3	↑ 3	↖ 4
A	0	↖ 1	↑ 2	↑ 2	↑ 3	↑ 3	↖ 4	↑ 4

Parul® University | NAAC GRADE A++

https://paruluniversity.ac.in/

Dynamic Programming:

Chapter-4

Mrs. Bhumi Shah

Assistant Professor

Department of Computer Science and Engineering

Content

1. Principal of Optimality:
- 0/1 Knapsack Problem,
- Making Change Problem.
2. Chain matrix multiplication,

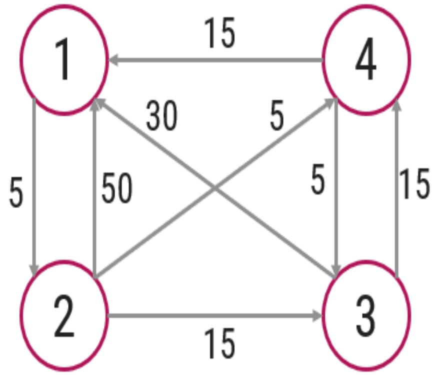
   Longest Common Subsequence.
3. All pair shortest paths.

INDEX

All pair shortest paths

- Given a directed, connected weighted graph G(V, E), for each edge ⟨u, v⟩∈E, a weight w(u, v) is associated with each edge.
- The all pairs of shortest paths problem is to find a shortest path from u to v for every pair of vertices u and v in V.
- Floyd's algorithm is used to find all pair shortest path problem from a given weighted graph.
- As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.
- At first, the output matrix is same as given cost matrix of the graph.
- As the algorithm proceeds, the output matrix will be updated with each vertex k as an intermediate vertex.
- The time complexity of this algorithm is O(n^3), where n is the number of vertices in the graph.

All pair shortest paths



Step: 1 Initialization

$$D_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & \infty & \infty \\ 2 & 50 & 0 & 15 & 5 \\ 3 & 30 & \infty & 0 & 15 \\ 4 & 15 & \infty & 5 & 0 \end{array}$$

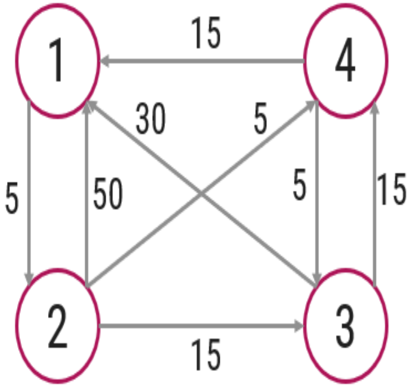
Step: 2 Calculate the distance between each node with **node 1** as an intermediate node.

$$D_1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & \infty & \infty \\ 2 & 50 & 0 & 15 & 5 \\ 3 & 30 & \infty & 0 & 15 \\ 4 & 15 & \infty & 5 & 0 \end{array}$$

For node 2	

No change

All pair shortest paths



Step: 1 Initialization

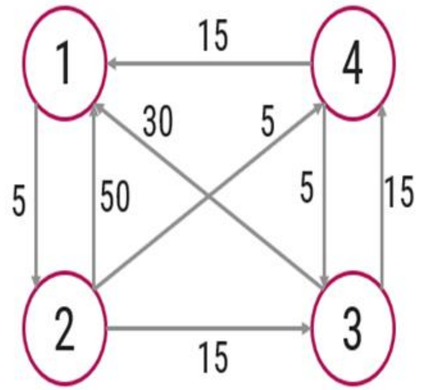
$$D_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & \infty & \infty \\ 2 & 50 & 0 & 15 & 5 \\ 3 & 30 & \infty & 0 & 15 \\ 4 & 15 & \infty & 5 & 0 \end{array}$$

Step: 2 Calculate the distance between each node with **node 1** as an intermediate node.

$$D_1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & \infty & \infty \\ 2 & 50 & 0 & 15 & 5 \\ 3 & 30 & \mathbf{35} & 0 & 15 \\ 4 & 15 & \infty & 5 & 0 \end{array}$$

For node 3

All pair shortest paths



Step: 1 Initialization

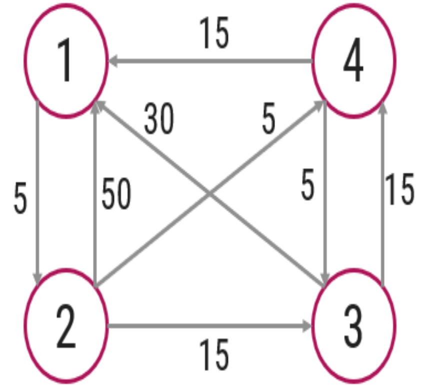
$$D_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & \infty & \infty \\ 2 & 50 & 0 & 15 & 5 \\ 3 & 30 & \infty & 0 & 15 \\ 4 & 15 & \infty & 5 & 0 \end{array}$$

Step: 2 Calculate the distance between each node with **node 1** as an intermediate node.

$$D_1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & \infty & \infty \\ 2 & 50 & 0 & 15 & 5 \\ 3 & 30 & \mathbf{35} & 0 & 15 \\ 4 & 15 & \mathbf{20} & 5 & 0 \end{array}$$

For node 4	

All pair shortest paths



Step: 1 Initialization

$$D_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & \infty & \infty \\ 2 & 50 & 0 & 15 & 5 \\ 3 & 30 & \infty & 0 & 15 \\ 4 & 15 & \infty & 5 & 0 \end{array}$$

Step: 3

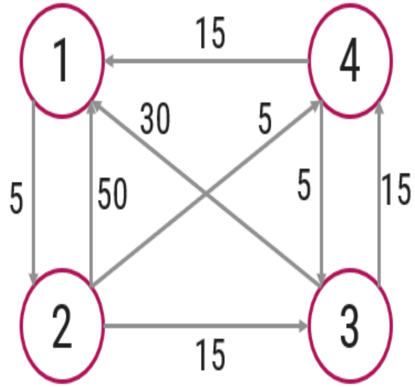
Calculate the distance between each node with **node 2** as an intermediate node.

$$D_2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & \mathbf{20} & \mathbf{10} \\ 2 & 50 & 0 & 15 & 5 \\ 3 & 30 & \mathbf{35} & 0 & 15 \\ 4 & 15 & \mathbf{20} & 5 & 0 \end{array}$$

For node 1	

No change for Node 3 & 4

All pair shortest paths



Step: 1 Initialization

$$D_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & \infty & \infty \\ 2 & 50 & 0 & 15 & 5 \\ 3 & 30 & \infty & 0 & 15 \\ 4 & 15 & \infty & 5 & 0 \end{array}$$

Step: 4

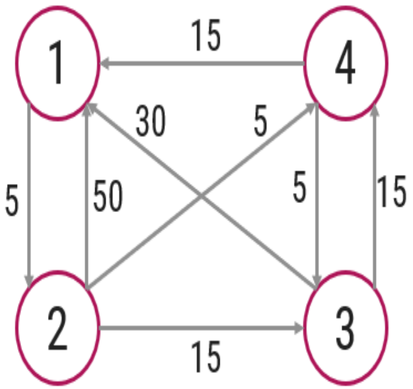
Calculate the distance between each node with **node 3** as an intermediate node.

$$D_3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & \mathbf{20} & \mathbf{10} \\ 2 & 50 & 0 & 15 & 5 \\ 3 & 30 & \mathbf{35} & 0 & 15 \\ 4 & 15 & \mathbf{20} & 5 & 0 \end{array}$$

For node 1	

No change

All pair shortest paths



Step: 1 Initialization

$$D_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & \infty & \infty \\ 2 & 50 & 0 & 15 & 5 \\ 3 & 30 & \infty & 0 & 15 \\ 4 & 15 & \infty & 5 & 0 \end{array}$$

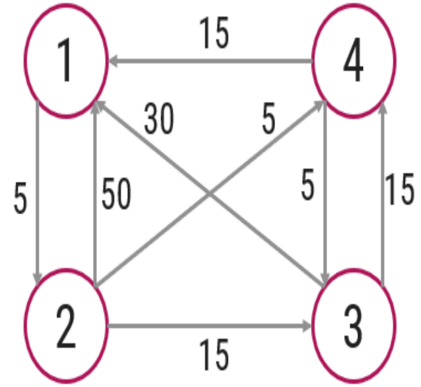
Step: 4 Calculate the distance between each node with **node 3** as an intermediate node.

$$D_3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & \mathbf{20} & \mathbf{10} \\ 2 & \mathbf{45} & 0 & 15 & 5 \\ 3 & 30 & \mathbf{35} & 0 & 15 \\ 4 & 15 & \mathbf{20} & 5 & 0 \end{array}$$

For node 2	

No change for Node 4

All pair shortest paths



Step: 1 Initialization

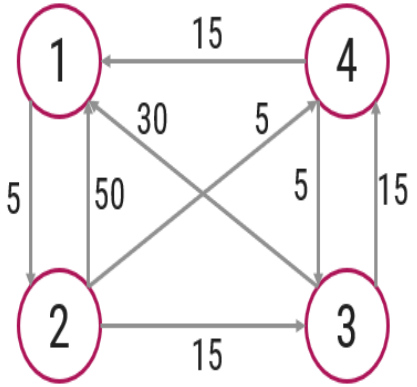
$$D_0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix} \end{array}$$

Step: 5 Calculate the distance between each node with **node 4** as an intermediate node.

$$D_4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 5 & \mathbf{15} & \mathbf{10} \\ \mathbf{45} & 0 & 15 & 5 \\ 30 & \mathbf{35} & 0 & 15 \\ 15 & \mathbf{20} & 5 & 0 \end{pmatrix} \end{array}$$

For node 1	

All pair shortest paths



Step: 1 Initialization

$$D_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & \infty & \infty \\ 2 & 50 & 0 & 15 & 5 \\ 3 & 30 & \infty & 0 & 15 \\ 4 & 15 & \infty & 5 & 0 \end{array}$$

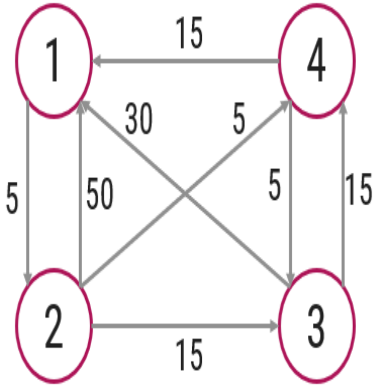
Step: 5 Calculate the distance between each node with **node 4** as an intermediate node.

$$D_4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & \mathbf{15} & \mathbf{10} \\ 2 & \mathbf{20} & 0 & \mathbf{10} & 5 \\ 3 & 30 & \mathbf{35} & 0 & 15 \\ 4 & 15 & \mathbf{20} & 5 & 0 \end{array}$$

For node 2

No change for Node 3

All pair shortest paths



Step: 1 Initialization

$$D_0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix} \end{array}$$

Step: 5

Calculate the distance between each node with **node 4** as an intermediate node.

$$D_4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 5 & \mathbf{15} & \mathbf{10} \\ \mathbf{20} & 0 & \mathbf{10} & 5 \\ 30 & \mathbf{35} & 0 & 15 \\ 15 & \mathbf{20} & 5 & 0 \end{pmatrix} \end{array}$$

Final Solution

Floyd’s Algorithm

```
function Floyd(L[1..n, 1..n]):array [1..n, 1..n]
  array D[1..n, 1..n]
  D ← L
  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
```

Parul®
University

NAAC GRADE A++

https://paruluniversity.ac.in/