Design and Analysis of Algorithm
Unit-4

Dynamic Programming

Dynamic programming

- Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.
- In contrast to the divide-and-conquer method, dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems.
- A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table.

Optimization problems

- We typically apply dynamic programming to ***optimization problems***.

- Such problems can have many possible solutions.
- Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.
- We call such a solution *an* optimal solution to the problem.

Dynamic programming

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1) Characterize the structure of an optimal solution.
2) Recursively define the value of an optimal solution.
3) Compute the value of an optimal solution, typically in a bottom-up fashion.
4) Construct an optimal solution from computed information.

Dynamic programming

Using dynamic programming, we shall solve the following problems:-

- Matrix-chain multiplication
- Longest common subsequence problem
- 0-1 Knapsack problem
- All pairs shortest path problem

Matrix-chain multiplication problem

This problem is stated as following:-

Given a sequence (chain) < A_1, A_2, ……., A_n > of n matrices where for i = 1, 2, 3, ……., n, matrix A_i has dimension p_{i-1} x p_i, fully parenthesize the product A_1A_2 ……. A_n in a way that minimizes the number of scalar multiplications.

Example: Find all parenthesization of matrix for n=4.

Solution:

1) ((A1A2)(A3A4))
2) (((A1A2)A3)A4)
3) (A1(A2(A3A4)))
4) ((A1(A2A3))A4)
5) (A1((A2A3)A4))

Matrix-chain multiplication problem

Example: Consider the following chain of matrices < A1, A2, A3 > .
The dimension of matrices are the following:-

A1 = 10x100, A2 = 100x5, A3= 5x50

Find the optimal parenthesization of these matrices.

Solution: All the parenthesization of these matrices are:-

1) ((A1A2)A3)
2) (A1(A2A3))

Number of scalar multiplications in (1)

= 10*100*5 + 10*5*50

= 5000 + 2500 = 7500

Number of scalar multiplications in (2)

= 100*5*50 + 10*100*50

= 25000 + 50000 = 75000

Therefore, the solution ((A1A2)A3) is optimal solution.

Number of parenthesizations

Let P(n) denote the number of parenthesizations for n matrices. It is computed as following:-

P(n) = 1 if n=1

$= \sum_{k=1}^{n-1} P(k)P(n-k)$ if n >=2

Dynamic programming approach for Matrix-chain multiplication problem

Step 1: In this step, we find the optimal substructure and then use it to construct an optimal solution to the problem.

Let $A_{i..j}$ → Matrix that results from evaluating the product $A_iA_{i+1}.....A_j$. Where $i \leq j$.

- If $i < j$, then to parenthesize the product $A_iA_{i+1}.....A_j$, we must split the product between A_k and A_{k+1} for some integer k in the range $i \leq k < j$. That is, for some value of k, we first compute the matrices $A_{i..k}$ and $A_{k+1..j}$ and then multiply them together to produce the final product $A_{i..j}$.
- The cost of parenthesizing this way is the cost of computing the matrix $A_{i..k}$, plus the cost of computing $A_{k+1..j}$, plus the cost of multiplying them together.

Matrix-chain multiplication problem

Step 2:

- Let m[i, j] be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$.
- m[i,j] is recursively defined as following:-

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

- We define s[i, j] to be the value of k at which m[i, k] + m[k+1, j] + $p_{i-1}p_kp_j$ is minimum.
- We compute the optimal solution using matrix s.
- And matrix m is used to compute the value of optimal solution.

Matrix-chain multiplication problem

Step 3: In this step, we compute the matrices m and s.

Following algorithm is used to compute the matrices m and s. This algorithm assumes that matrix A_i has dimensions $p_{i-1} x p_i$ for i = 1, 2, ..., n. Its input is a sequence $p = < p_0, p_1, ..., p_n >$, where p.*length* = n+1.

```
MATRIX-CHAIN-ORDER(p)
 1   n = p.length − 1
 2   let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3   for i = 1 to n
 4       m[i, i] = 0
 5   for l = 2 to n                    // l is the chain length
 6       for i = 1 to n − l + 1
 7           j = i + l − 1
 8           m[i, j] = ∞
 9           for k = i to j − 1
10               q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11               if q < m[i, j]
12                   m[i, j] = q
13                   s[i, j] = k
14   return m and s
```

Matrix-chain multiplication problem

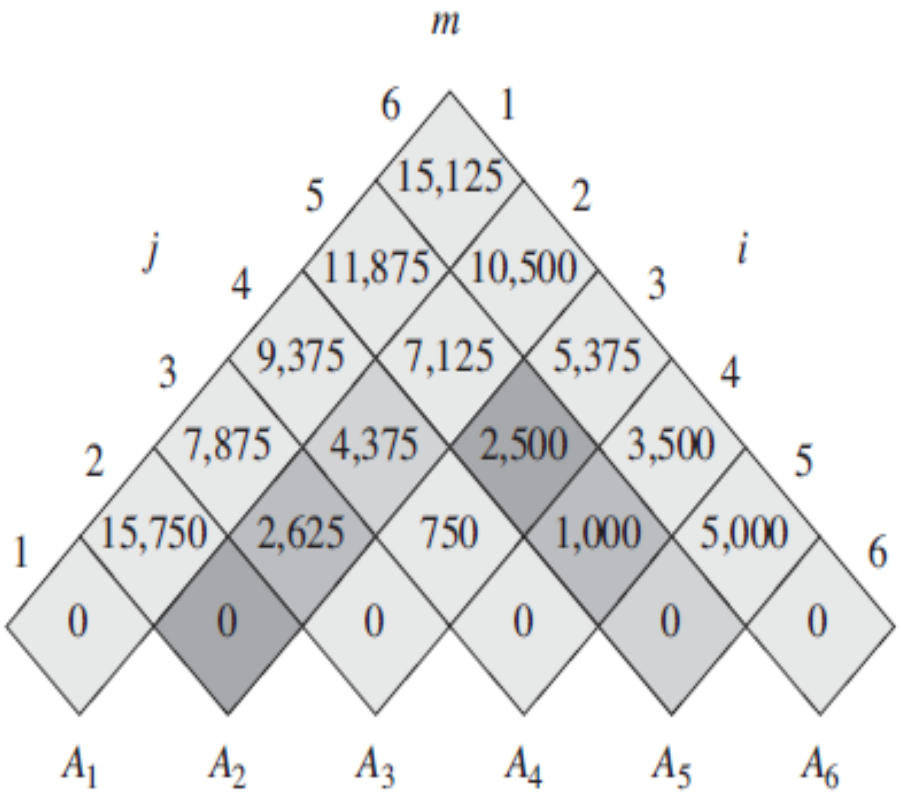
Example: Consider the following matrix chain multiplication problem.

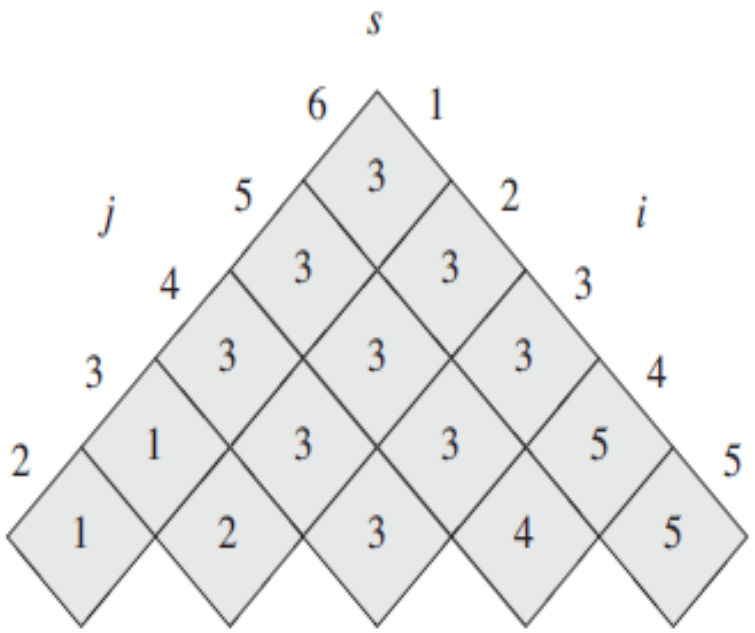
$A_1 \rightarrow$ 30x35, $A_2 \rightarrow$ 35x15, $A_3 \rightarrow$ 15x5,

$A_4 \rightarrow$ 5x10, $A_5 \rightarrow$ 10x20, $A_6 \rightarrow$ 20x25.

Find the optimal solution and its value.

Solution:





Matrix-chain multiplication problem

Step 4: Constructing an optimal solution

Following algorithm is used to create optimal solution i.e. it finds optimal parenthesization.

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

- The initial call PRINT-OPTIMAL-PARENS(s, 1, n) prints an optimal parenthesization of $< A_1, A_2, \ldots\ldots, A_n >$.

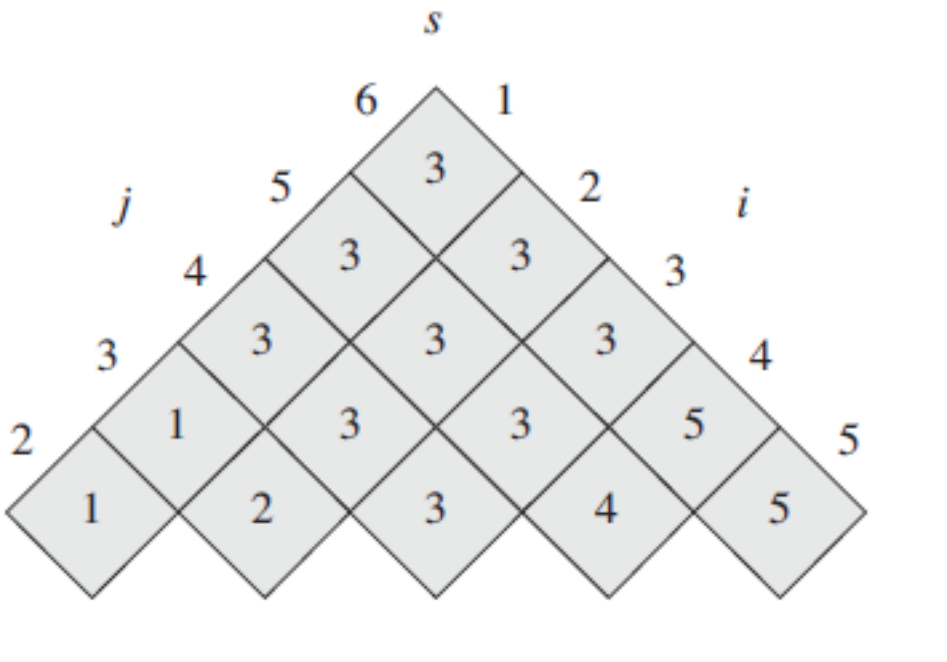
Time complexity of this algorithm is O(n).

Matrix-chain multiplication problem

Step 4: Constructing an optimal solution for previous example

The initial call PRINT-OPTIMAL-PARENS(s, 1, 6) prints an optimal parenthesization .

Optimal solution will be
((A1(A2A3))((A4A5)A6))



Longest Common Subsequence Problem (LCS problem)

In the ***longest-common-subsequence problem***, we are given two sequences $X = < x_1, x_2, \ldots\ldots\ldots, x_m >$ and $Y = < y_1, y_2, \ldots\ldots\ldots, y_n >$ and wish to find a maximum length common subsequence of X and Y.

Longest Common Subsequence Problem (LCS problem)

Example:

- If X = < A, B, C, B, D, A, B > and Y= < B, D, C, A, B, A >, the sequence < B, C, A > is a common subsequence of both X and Y .
- The sequence < B, C, A > is not a *longest* common subsequence (LCS) of X and Y , however, since it has length 3 and the sequence < B, C, B, A >, which is also common to both X and Y , has length 4.
- The sequence < B, C, B, A > is an LCS of X and Y , as is the sequence < B, D, A, B >, since X and Y have no common subsequence of length 5 or greater.

Solving the LCS problem using dynamic programming

Step-1: Characterizing a longest common subsequence

***Prefix of* a sequence:**

Given a sequence $X = < x_1, x_2, \ldots\ldots, x_m >$, we define the i^{th} ***prefix*** of X, for i = 0, 1, 2, ..., m, as $X_i = < x_1, x_2, \ldots\ldots, x_i >$.

For example, if $X = < A, B, C, B, D, A, B >$, then $X_4 = <A, B, C, B >$ and X_0 is the empty sequence.

Solving the LCS problem using dynamic programming

Optimal substructure of an LCS

Let $X = < x_1, x_2, \ldots\ldots, x_m >$ and $Y = < y_1, y_2, \ldots\ldots, y_n >$ be sequences, and let $Z = < z_1, z_2, \ldots\ldots, z_k >$ be any LCS of X and Y .

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y .

3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Solving the LCS problem using dynamic programming

Step 2: A recursive solution

Let c[i, j] be the length of an LCS of the sequences X_i and Y_j. If either i = 0 or j = 0, one of the sequences has length 0, and so the LCS has length 0. The optimal substructure of the LCS problem gives the recursive formula:-

c[i,j] = 0 if i = 0 or j = 0

= c[i-1,j-1]+1, if i, j > 0 and $x_i = y_j$

= max{ c[i-1, j], c[i,j-1] }, if i, j > 0 and $x_i \neq y_j$

Let b is the matrix which stores one of the following three arrows, ←, ↑ and ↖.

b[i,j] = ← , if c[i,j] = c[i,j-1]

b[i,j] = ↑ , if c[i,j] = c[i-1,j]

b[i,j] = ← , if c[i,j] = c[i-1,j-1]

LCS problem using dynamic programming

Step-3: Computing the length of an LCS

Following procedure is used to compute the table b and c.

```
LCS-LENGTH(X, Y)
 1   m = X.length
 2   n = Y.length
 3   let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4   for i = 1 to m
 5       c[i, 0] = 0
 6   for j = 0 to n
 7       c[0, j] = 0
 8   for i = 1 to m
 9       for j = 1 to n
10           if x_i == y_j
11               c[i, j] = c[i − 1, j − 1] + 1
12               b[i, j] = "↖"
13           elseif c[i − 1, j] ≥ c[i, j − 1]
14               c[i, j] = c[i − 1, j]
15               b[i, j] = "↑"
16           else c[i, j] = c[i, j − 1]
17               b[i, j] = "←"
18   return c and b
```

LCS problem using dynamic programming

Step-3: Computing the length of an LCS.

Example: Find LCS of the following sequences:-

X = < A, B, C, B, D, A, B > and Y = < B, D, C, A, B, A >.

Solution: We compute the table corresponding to b and c as following:-

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

LCS problem using dynamic programming

Step-4: Constructing an LCS

- The following recursive procedure prints out an LCS of X and Y in the proper, forward order. The initial call is PRINT-LCS(b, X, m, n).

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

LCS problem using dynamic programming

Step-4:

Time complexity of this algorithm = O(m+n).

The LCS of sequences taken in previous example = BCBA.

0-1 Knapsack using dynamic programming

- Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W.

- Based on the nature of the items, Knapsack problems are categorized as

1. Fractional Knapsack
2. 0-1 Knapsack

0-1 Knapsack using dynamic programming

Fractional Knapsack

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of $\mathbf{i^{th}}$ item.

0-1 Knapsack

In this version of Knapsack problem, we cannot break an item, either pick the complete item or don't pick it (0-1 property).

0-1 Knapsack using dynamic programming

Dynamic programming approach for solving 0-1 Knapsack problem is explained as following:-

Let c[i,w] denotes the value of the solution for items 1,2,3,…,i and maximum weight w.

It is defined as

$c[i,w] = 0 \quad$,if i = 0 or w = 0

$= c[i-1, w] \quad$, if i > 0 and $w_i > w$

$= \max\{ v_i + c[i-1,w-w_i], c[i-1,w] \}$, if i > 0 and $w_i \leq w$

0-1 Knapsack using dynamic programming

Dynamic programming algorithm for solving 0-1 Knapsack problem is the following:-

Dynamic-0-1-Knapsack(v, w, n, W)

```
for l← 0 to W
        c[0,l] ← 0
for i← 1 to n
        c[i,0] ← 0
        for l← 1 to W
                if w_i ≤ l then
                        if v_i + c[i-1,l-w_i] > c[i-1,l] then
                                c[i,l] ← v_i + c[i-1,l-w_i]
                        else
                                c[i,l] ← c[i-1,l]
                else
                        c[i,l] ← c[i-1,l]
return c
```

Time complexity of this algorithm = O(nW)

0-1 Knapsack using dynamic programming

Example: Solve the following 0-1 knapsack.

$n = 4,\ W = 5$

$(v_1, v_2, v_3, v_4) = (3, 4, 5, 6)$

$(w_1, w_2, w_3, w_4) = (2, 3, 4, 5)$

Solution:

Here, we have to calculate $c[4, 5]$.

$c[4,5] = \max\{ v_4 + c[3,0], c[3,5] \} = \max\{ 6 + 0, c[3,5] \}$

$= \max\{ 6, c[3,5] \} = \max\{ 6, 7\} = 7$

$c[3,5] = \max\{ v_3 + c[2, 1], c[2, 5] \} = \max\{ 5 + 0, c[2, 5] \}$

$= \max\{ 5, c[2, 5] \} = \max\{ 5, 7\} = 7$

$c[2,5] = \max\{ v_2 + c[1, 2], c[1, 5] \} = \max\{ 4 + 3, 3\} = 7$

Therefore, the value of the optimal solution is 7. And the optimal solution is (1, 2).

0-1 Knapsack using dynamic programming

Solution by Tabular Method

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

This table is used to find the value of the optimal solution.

Therefore, value of optimal solution = 7

0-1 Knapsack using dynamic programming

Solution by Tabular Method

	1	2	3	4	5
1	↑	←	←	←	←
2	↑	↑	←	←	←
3	↑	↑	↑	←	↑
4	↑	↑	↑	↑	↑

This table is used to find the optimal solution.

Therefore, optimal solution = (1, 2)

0-1 Knapsack using dynamic programming

Example: Solve the following 0-1 knapsack.

$n = 6, \quad W = 100$

$(v_1, v_2, v_3, v_4, v_5, v_6) = (40, 35, 20, 4, 10, 6)$

$(w_1, w_2, w_3, w_4, w_5, w_6) = (100, 50, 40, 20, 10, 10)$

Solution:

	0	10	20	30	40	50	60	70	80	90	100
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	40
2	0	0	0	0	0	35	35	35	35	35	40
3	0	0	0	0	20	35	35	35	35	55	55
4	0	0	4	4	20	35	35	39	39	55	55
5	0	10	10	14	20	35	45	45	49	55	65
6	0	10	16	16	20	35	45	51	51	55	65

0-1 Knapsack using dynamic programming

From table in the previous slide, the value of optimal solution will be 65.

Following is the table to compute the optimal solution.

The optimal solution will be (2, 3, 5).

	10	20	30	40	50	60	70	80	90	100
1	↑	↑	↑	↑	↑	↑	↑	↑	↑	←
2	↑	↑	↑	↑	←	←	←	←	←	↑
3	↑	↑	↑	←	↑	↑	↑	↑	←	←
4	↑	←	←	↑	↑	↑	←	←	↑	↑
5	←	←	←	↑	↑	←	←	←	↑	←
6	↑	←	←	↑	↑	↑	←	←	↑	↑

Floyd's Warshall Algorithm for shortest path

- This algorithm is used to solve the all-pairs shortest-paths problem on a directed graph G =(V,E).
- This algorithm is based on the dynamic programming approach.
- Let $\mathbf{d^{(k)}_{ij}}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set {1, 2, ……., k}.
- When k = 0, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence $\mathbf{d^{(0)}_{ij} = w_{ij}}$.

Floyd's Warshall Algorithm for shortest path

Recursive formula to compute $d^{(k)}_{ij}$

We define $d^{(k)}_{ij}$ as the following:-

$$d^{(k)}_{ij} = w_{ij} \quad \text{if } k = 0$$

$$= \min\{ d^{(k-1)}_{ij} , d^{(k-1)}_{ik} + d^{(k-1)}_{kj} \} \quad \text{if } k \geq 1.$$

- Because for any path, all intermediate vertices are in the set {1, 2, ..., n}, therefore the matrix $D^{(n)} = (d^{(n)}_{ij})$ gives the final answer.

Floyd's Warshall Algorithm for shortest path

Constructing a shortest path

- We define $\pi^{(k)}_{ij}$ as the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set {1, 2, ..., k}.
- When **k = 0**, a shortest path from i to j has no intermediate vertices at all. Therefore,

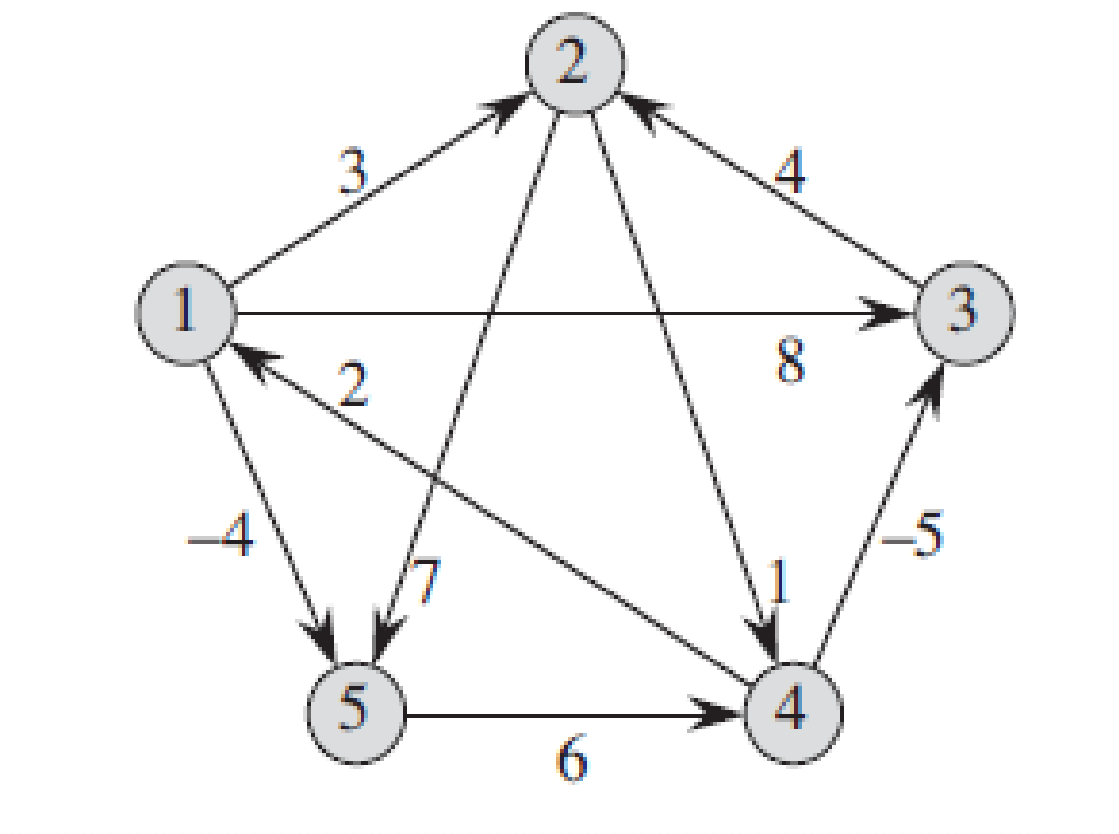
$$\pi^{(0)}_{ij} = \text{NIL} \quad \text{if } i=j \text{ or } w_{ij} = \infty$$
$$= i \quad \text{if } i \neq j \text{ and } w_{ij} < \infty.$$

- **For k ≥ 1.**

$$\pi^{(k)}_{ij} = \pi^{(k-1)}_{ij}, \quad \text{if } d^{(k-1)}_{ij} \leq d^{(k-1)}_{ik} + d^{(k-1)}_{kj}$$
$$= \pi^{(k-1)}_{kj}, \quad \text{if } d^{(k-1)}_{ij} > d^{(k-1)}_{ik} + d^{(k-1)}_{kj}$$

Floyd's Warshall Algorithm for shortest path

Example: Apply the Floyd's Warshall algorithm in the following graph :-



Floyd's Warshall Algorithm for shortest path

Solution: Weighted matrix of this graph is the following:-

$$W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Therefore,

$$D^{(o)} = W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(o)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

Floyd's Warshall Algorithm for shortest path

Here, we have to calculate $D^{(5)}$ and $\Pi^{(5)}$. It is calculated as following:-

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

Floyd's Warshall Algorithm for shortest path

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Floyd's Warshall Algorithm for shortest path

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

- Matrix $D^{(5)}$ stores the shortest distance between each pair of vertices.
- Matrix $\Pi^{(5)}$ is used to find shortest path between each pair of vertices.

Floyd's Warshall Algorithm for shortest path

```
Floyd-Warshall(W, n)
    D(0) = W
    for i←1 to n
        for j← 1 to n
            if (i=j or wij=∞)
                π(0)ij = Nil
            if (i≠j and wij< ∞)
                π(0)ij = i
    for k←1 to n
        for i←1 to n
            for j← 1 to n
                if (d(k-1)ij ≤ d(k-1)ik + d(k-1)kj )
                    d(k)ij = d(k-1)ij
                    π(k)ij = π(k-1)ij
                else
                    d(k)ij = d(k-1)ik + d(k-1)kj
                    π(k)ij = π(k-1)kj
```

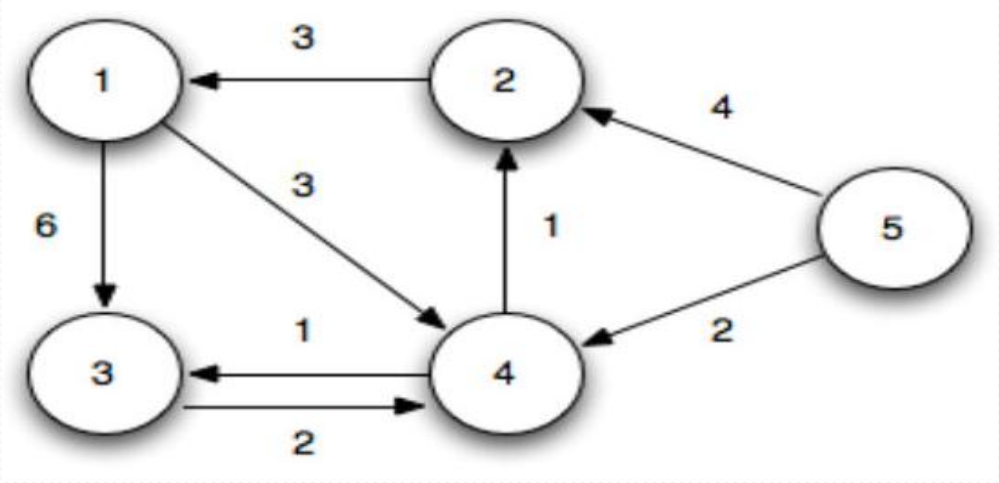
Time complexity of this algorithm is $\theta(n^3)$.

AKTU Examination Questions

1. Define principal of optimality. When and how dynamic programming is applicable.
2. Give Floyd Warshall algorithm to find the shortest path for all pairs of vertices in a graph. Give the complexity of the algorithm. Explain with example.
3. Difference between Greedy Technique and Dynamic programming.
4. Write down an algorithm to compute Longest Common Subsequence (LCS) of two given strings and analyze its time complexity.
5. What is dynamic programming? How is this approach different from recursion? Explain with example.
6. Solve the following 0/1 knapsack problem using dynamic programming. P=[11,21,31,33] w=[2,11,22,15] c=40, n=4.

AKTU Examination Questions

7. Define Floyd Warshall Algorithm for all pair shortest path and apply the same on following graph:



7. Define the terms—LCS, Matrix Chain multiplication & Bellman-Ford algorithm.
8. Explain the Floyd Warshall algorithm with an example.
9. Find an optimal parenthesization of a matrix chain product whose sequence of dimensions is {10, 5, 3, 12, 6}.

Thank You.