Design and Analysis of Algorithms (CSC-314)

B.Sc. CSIT

Unit-5: Dynamic Programming

Introduction:

- Dynamic programming is an optimization method which was developed by Richard Bellman in 1950.
- Dynamic programming is used to solve the multistage optimization problem in which dynamic means reference to time and programming means planning or tabulation.
- Dynamic programming approach consists of three steps for solving a problem that is as follows:
  - The given problem is divided into subproblems as same as in divide and conquer rule. However dynamic programming is used when the subproblems are not independent of each other but they are interrelated. i.e. they are also called as overlapping problems.
  - To avoid this type of recomputation of overlapping subproblem a table is created in which whenever a subproblem is solved, then its solution will be stored in the table so that in future its solution can be reused.
  - The solution of the subproblem is combined in a bottom of manner to obtain the optimal solution of a given problem.

Unit-5: Dynamic Programming

Introduction:

- Dynamic Programming is mainly an optimization over plain recursion.
- Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming.
- The idea is to simply store the results of sub-problems, so that we do not have to re-compute them when needed later.
- This simple optimization reduces time complexities from exponential to polynomial.

Unit-5: Dynamic Programming

Introduction:

- Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler sub-problems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

- Let's take the example of the Fibonacci numbers. As we all know, Fibonacci numbers are a series of numbers in which each number is the sum of the two preceding numbers. The first few Fibonacci numbers are 0, 1, 1, 2, 3, 5, and 8, and they continue on from there.

Unit-5: Dynamic Programming

Introduction:

- Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler sub-problems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

- Let's take the example of the Fibonacci numbers. As we all know, Fibonacci numbers are a series of numbers in which each number is the sum of the two preceding numbers. The first few Fibonacci numbers are 0, 1, 1, 2, 3, 5, and 8, and they continue on from there.

Unit-5: Dynamic Programming

Introduction:

- If we are asked to calculate the nth Fibonacci number, we can do that with the following equation,

  - Fib(n) = Fib(n-1) + Fib(n-2), for n > 1

- As we can clearly see here, to solve the overall problem (i.e. Fib(n)), we broke it down into two smaller subproblems (which are Fib(n-1) and Fib(n-2)). This shows that we can use DP to solve this problem.
- If we write simple recursive solution for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of sub-problems, time complexity reduces to linear.

Unit-5: Dynamic Programming

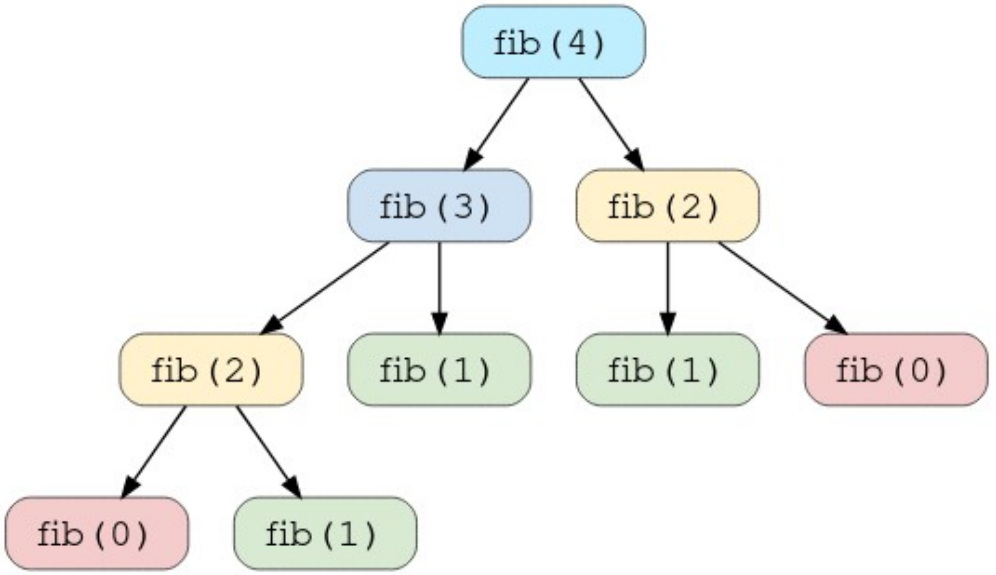
Introduction: Characteristics of Dynamic Programming

- Overlapping Sub-problems
  - Subproblems are smaller versions of the original problem.
  - Any problem has overlapping sub-problems if finding its solution involves solving the same subproblem multiple times.
  - Take the example of the Fibonacci numbers; to find the fib(4), we need to break it down into the following sub-problems:

Unit-5: Dynamic Programming

Introduction:Characteristics of Dynamic Programming

- Overlapping Sub-problems



Recursion tree for calculating Fibonacci numbers

- We can clearly see the overlapping sub-problem pattern here, as fib(2) has been evaluated twice and fib(1) has been evaluated three times.

Unit-5: Dynamic Programming

Introduction: Characteristics of Dynamic Programming

- Optimal Substructure Property
  - Any problem has optimal substructure property if its overall optimal solution can be constructed from the optimal solutions of its sub-problems.
  - For Fibonacci numbers, as we know,
    - Fib(n) = Fib(n-1) + Fib(n-2)
  - This clearly shows that a problem of size ‘n’ has been reduced to sub-problems of size ‘n-1’ and ‘n-2’. Therefore, Fibonacci numbers have optimal substructure property.

Unit-5: Dynamic Programming

Introduction: Greedy Algorithm vs Dynamic Programming

Dynamic Programming	Greedy Method
1. Dynamic Programming is used to obtain the optimal solution.	1. Greedy Method is also used to get the optimal solution.
2. In Dynamic Programming, we choose at each step, but the choice may depend on the solution to sub-problems.	2. In a greedy Algorithm, we make whatever choice seems best at the moment and then solve the sub-problems arising after the choice is made.
3. Less efficient as compared to a greedy approach	3. More efficient as compared to a greedy approach
4. Example: 0/1 Knapsack	4. Example: Fractional Knapsack
5. It is guaranteed that Dynamic Programming will generate an optimal solution using Principle of Optimality.	5. In Greedy Method, there is no such guarantee of getting Optimal Solution.

Unit-5: Dynamic Programming

Introduction: Greedy Algorithm vs Dynamic Programming

BASIS FOR COMPARISON	GREEDY METHOD	DYNAMIC PROGRAMMING
Basic	Generates a single decision sequence.	Many decision sequence may be generated.
Reliability	Less reliable	Highly reliable
Follows	Top-down approach.	Bottom-up approach.
Solutions	Contain a particular set of feasible set of solutions.	There is no special set of feasible set of solution.
Efficiency	More	Less
Overlapping subproblems	Cannot be handled	Chooses the optimal solution to the subproblem.
Example	Fractional knapsack, shortest path.	0/1 Knapsack

Concept of Matrix Chain Multiplication:

- Matrix chain multiplication (or Matrix Chain Ordering Problem, MCOP) is an optimization problem that to find the most efficient way to multiply a given sequence of matrices.
- The problem is not actually to perform the multiplications but merely to decide the sequence of the matrix multiplications involved.
- The matrix multiplication is associative as no matter how the product is parenthesized, the result obtained will remain the same.
- For example, for four matrices A, B, C, and D, we would have:
  - ((AB)C)D = ((A(BC))D) = (AB)(CD) = A((BC)D) = A(B(CD))
-

Concept of Matrix Chain Multiplication:

- However, the order in which the product is parenthesized affects the number of simple arithmetic operations needed to compute the product.
- For example, if A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix,
  - then computing (AB)C needs (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations
  - while computing A(BC) needs (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.
- Clearly, the first method is more efficient.
- Let A i….j denote the result of multiplying matrices i through j. It is easy to see that A i…j is a Pi−1 x p j matrix.
- So for some k total cost is sum of cost of computing A i…k , cost of computing A k+1…j , and cost of multiplying A i…k and A k+1…j .

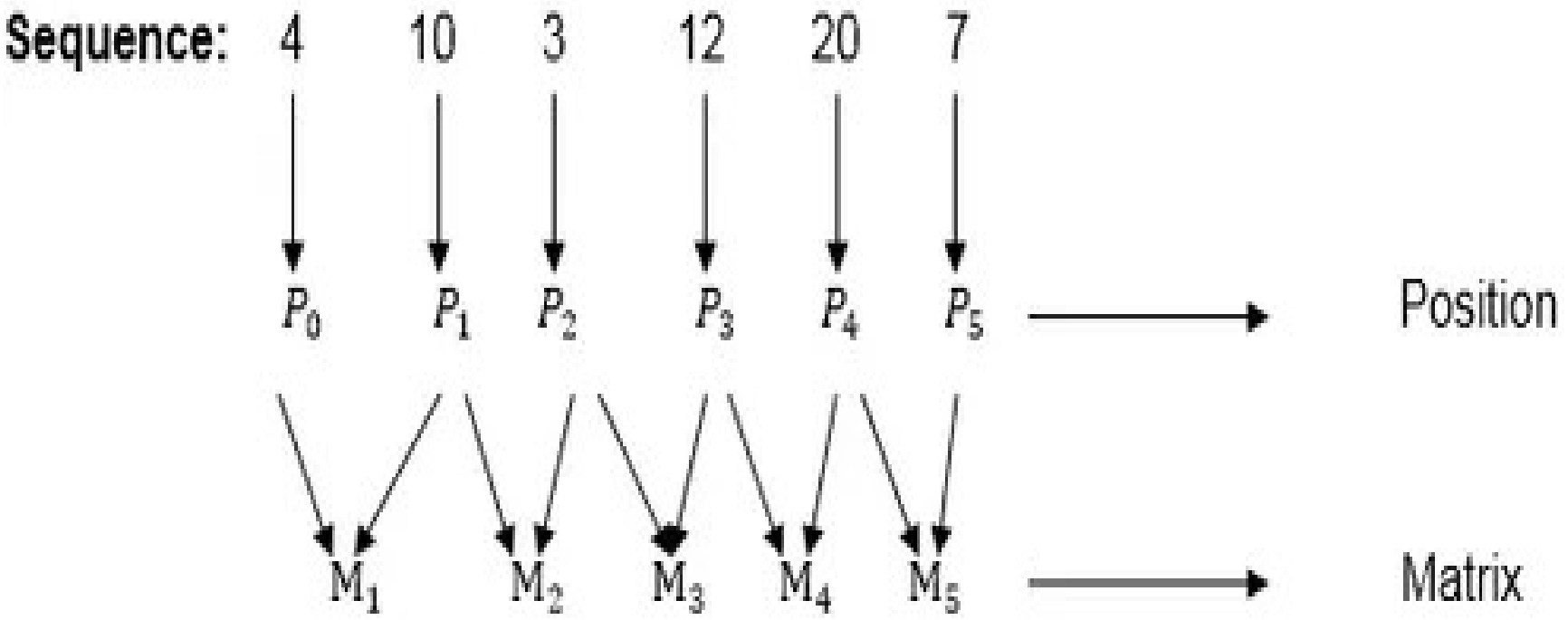
Concept of Matrix Chain Multiplication:

- Example : We are given the sequence {4, 10, 3, 12, 20, and 7}. The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7. We need to compute M [i,j], 0 ≤ i, j≤ 5. We know M [i, i] = 0 for all i.
- **Solution:**
- Let us proceed with working away from the diagonal. We compute the optimal solution for the product of 2 matrices.

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

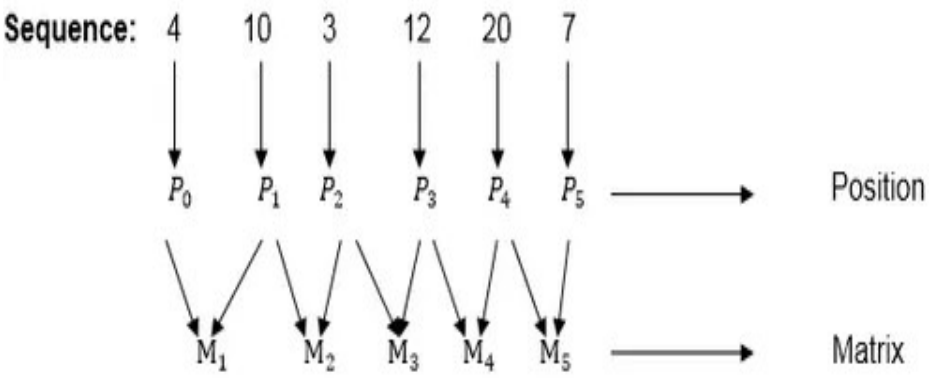
Concept of Matrix Chain Multiplication:

- Example-1 : We are given the sequence {4, 10, 3, 12, 20, and 7}. The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7. We need to compute M [i,j], 0 ≤ i, j≤ 5. We know M [i, i] = 0 for all i.
- **Solution:**



Concept of Matrix Chain Multiplication:

- Example-1 : We are given the sequence {4, 10, 3, 12, 20, and 7}. The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7. We need to compute M [i,j], 0 ≤ i, j≤ 5. We know M [i, i] = 0 for all i.
- **Solution:**
  - Here P0 to P5 are Position and M1 to M5 are matrix of size (pi to pi-1)
  - On the basis of sequence, we make a formula , for Mi ------> p[i] as column and p[i-1] as row .
  - In Dynamic Programming, initialization of every method done by '0'.So we initialize it by '0'.It will sort out diagonally.
  - We have to sort out all the combination but the minimum output combination is taken into consideration.
  - M1: 4810
  - M2: 10*3
  - M3: 3*12
  - M4: 12*20
  - M5: 20*7



Unit-5: Dynamic Programming

Concept of Matrix Chain Multiplication:

- **Solution:**
-

Calculation of Product of 2 matrices:

1. m (1,2) = m1 x m2
   = 4 x 10 x 10 x 3
   = 4 x 10 x 3 = 120

2. m (2, 3) = m2 x m3
   = 10 x 3 x 3 x 12
   = 10 x 3 x 12 = 360

3. m (3, 4) = m3 x m4
   = 3 x 12 x 12 x 20
   = 3 x 12 x 20 = 720

4. m (4,5) = m4 x m5
   = 12 x 20 x 20 x 7
   = 12 x 20 x 7 = 1680

Unit-5: Dynamic Programming

Concept of Matrix Chain Multiplication:

Solution:

- We initialize the diagonal element with equal i,j value with '0'.
- After that second diagonal is sorted out and we get all the values corresponded to it
- Now the third diagonal will be solved out in the same way.

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

Unit-5: Dynamic Programming

Concept of Matrix Chain Multiplication:

- **Solution**

Now product of 3 matrices:

M [1, 3] = M1 M2 M3

1. There are two cases by which we can solve this multiplication: (M1 x M2) + M3, M1+ (M2x M3)
2. After solving both cases we choose the case in which minimum output is there.

$$M[1,3] = \min \begin{cases} M[1,2] + M[3,3] + p_0 p_2 p_3 = 120 + 0 + 4.3.12 = 264 \\ M[1,1] + M[2,3] + p_0 p_1 p_3 = 0 + 360 + 4.10.12 = 840 \end{cases}$$

M [1, 3] =264

As Comparing both output 264 is minimum in both cases so we insert 264 in table and (M1 x M2) + M3 this combination is chosen for the output making.

M [2, 4] = M2 M3 M4

1. There are two cases by which we can solve this multiplication: (M2x M3)+M4, M2+(M3 x M4)
2. After solving both cases we choose the case in which minimum output is there.

$$M[2,4] = \min \begin{cases} M[2,3] + M[4,4] + p_1 p_3 p_4 = 360 + 0 + 10.12.20 = 2760 \\ M[2,2] + M[3,4] + p_1 p_2 p_4 = 0 + 720 + 10.3.20 = 1320 \end{cases}$$

M [2, 4] = 1320

As Comparing both output 1320 is minimum in both cases so we insert 1320 in table and M2+(M3 x M4) this combination is chosen for the output making.

Unit-5: Dynamic Programming

Concept of Matrix Chain Multiplication:

- **Solution**

Now product of 3 matrices:

> *M [3, 5] = M3 M4 M5*

1. There are two cases by which we can solve this multiplication: (M3 x M4) + M5, M3+ (M4xM5)
2. After solving both cases we choose the case in which minimum output is there.

$$M[3,5] = \min \begin{cases} M[3,4] + M[5,5] + p_2p_4p_5 = 720 + 0 + 3.20.7 = 1140 \\ M[3,3] + M[4,5] + p_2p_3p_5 = 0 + 1680 + 3.12.7 = 1932 \end{cases}$$

> *M [3, 5] = 1140*

As Comparing both output 1140 is minimum in both cases so we insert 1140 in table and (M3 x M4) + M5 this combination is chosen for the output making.

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

Unit-5: Dynamic Programming

Concept of Matrix Chain Multiplication:

- **Solution**

Now product of 4 matrices:

M [1, 4] = M1 M2 M3 M4

There are three cases by which we can solve this multiplication:

1. (M1 x M2 x M3) M4
2. M1 x(M2 x M3 x M4)
3. (M1 xM2) x (M3 x M4)

After solving these cases we choose the case in which minimum output is there

$$M[1,4] = \min \begin{cases} M[1,3] + M[4,4] + p_0p_3p_4 = 264 + 0 + 4.12.20 = 1224 \\ M[1,2] + M[3,4] + p_0p_2p_4 = 120 + 720 + 4.3.20 = 1080 \\ M[1,1] + M[2,4] + p_0p_1p_4 = 0 + 1320 + 4.10.20 = 2120 \end{cases}$$

M [1, 4] =1080

As comparing the output of different cases then '1080' is minimum output, so we insert 1080 in the table and (M1 xM2) x (M3 x M4) combination is taken out in output making,

Unit-5: Dynamic Programming

Concept of Matrix Chain Multiplication:

- **Solution**

Now product of 4 matrices:

> *M [2, 5] = M2 M3 M4 M5*

There are three cases by which we can solve this multiplication:

1. (M2 x M3 x M4)x M5
2. M2 x(M3 x M4 x M5)
3. (M2 x M3)x (M4 x M5)

After solving these cases we choose the case in which minimum output is there

$$M[2,5] = \min \begin{cases} M[2,4] + M[5,5] + p_1p_4p_5 = 1320 + 0 + 10.20.7 = 2720 \\ M[2,3] + M[4,5] + p_1p_3p_5 = 360 + 1680 + 10.12.7 = 2880 \\ M[2,2] + M[3,5] + p_1p_2p_5 = 0 + 1140 + 10.3.7 = 1350 \end{cases}$$

> *M [2, 5] = 1350*

As comparing the output of different cases then '1350' is minimum output, so we insert 1350 in the table and M2 x(M3 x M4 xM5)combination is taken out in output making.

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

Unit-5: Dynamic Programming

Concept of Matrix Chain Multiplication:

- **Solution**

Now product of 5 matrices:

> *M [1, 5] = M1 M2 M3 M4 M5*

There are five cases by which we can solve this multiplication:

1. (M1 x M2 xM3 x M4)x M5
2. M1 x(M2 xM3 x M4 xM5)
3. (M1 x M2 xM3)x M4 xM5
4. M1 x M2x(M3 x M4 xM5)

After solving these cases we choose the case in which minimum output is there

$$M[1,5] = \min \begin{cases} M[1,4] + M[5,5] + p_0 p_4 p_5 = 1080 + 0 + 4.20.7 = 1544 \\ M[1,3] + M[4,5] + p_0 p_3 p_5 = 264 + 1680 + 4.12.7 = 2016 \\ M[1,2] + M[3,5] + p_0 p_2 p_5 = 120 + 1140 + 4.3.7 = 1344 \\ M[1,1] + M[2,5] + p_0 p_1 p_5 = 0 + 1350 + 4.10.7 = 1630 \end{cases}$$

> *M [1, 5] = 1344*

As comparing the output of different cases then '1344' is minimum output, so we insert 1344 in the table and M1 x M2 x(M3 x M4 x M5)combination is taken out in output making.

Unit-5: Dynamic Programming

Concept of Matrix Chain Multiplication:

- **Solution**

Finally:

Final Output is:

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

- . .

(M1M2M3M4M5)

(M1M2) (M3M4M5)

(M3M4) (M5)

Unit-5: Dynamic Programming

Concept of Matrix Chain Multiplication:

- Example: We are given the sequence {3,4,5,2 and 3}. The matrices M1,M2,M3,M4 have size of 3*4, 4*5, 5*2, 2*3. Compute the optimal sequence for the computation of multiplication operation.
-

Unit-5: Dynamic Programming

Concept of Matrix Chain Multiplication:

- **Recursive definition of optimal solution:** let m[j,j] denotes minimum number of scalar multiplications needed to compute $A_{i...j}$.

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

Unit-5: Dynamic Programming

Concept of Matrix Chain Multiplication:Pseudo Code

```
Matrix-Chain-Multiplication(p)
{
        n =length[p]
        for( i= 1 i<=n i++)
        {
                m[i, i]= 0
        }
    for(l=2; l<= n; l++)
    {
        for( i= 1; i<=n-l+1; i++)
        {
                j = i + l − 1
            m[i, j] = ∞
            for(k= i; k<= j-1; k++)
            {
                        c= m[i, k] + m[k + 1, j] + p[i−1] * p[k] * p[j]
                        if c < m[i, j]
                        {
                                m[i, j] = c
                                s[i, j] = k
                        }
            }
        }
    }
return m and s
}
```

Unit-5: Dynamic Programming

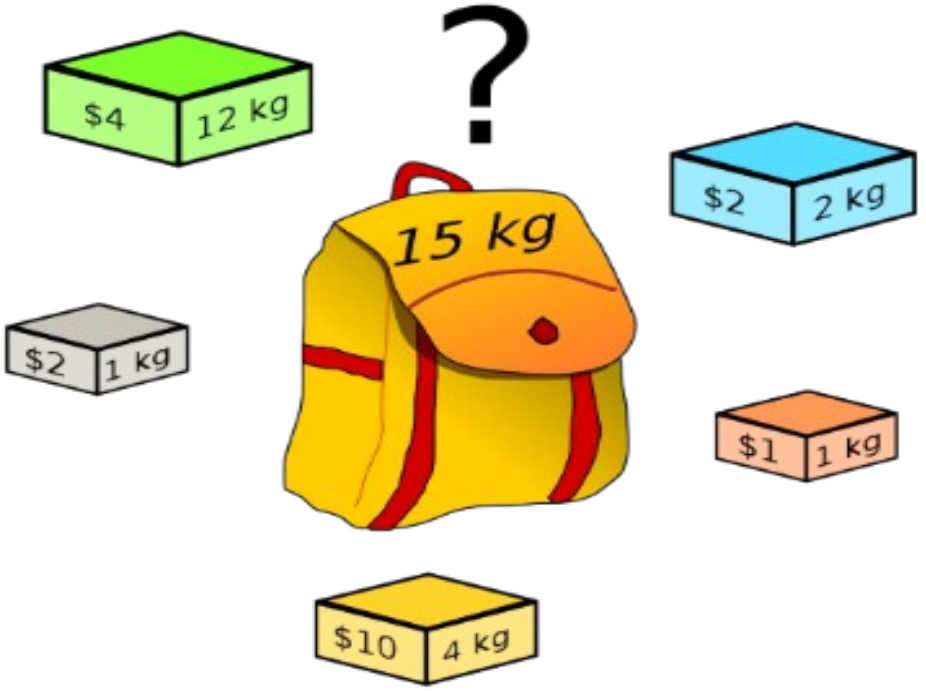
Concept of Matrix Chain Multiplication:Analysis

- The above algorithm can be easily analyzed for running time as O(n*n*n), due to three nested loops.
- The space complexity is O(n*n)

Unit-5: Dynamic Programming

0/1 Knapsack Problem:

- The problem states-
  - Which items should be placed into the knapsack such that-
  - The value or profit obtained by putting the items into the knapsack is maximum.
  - And the weight limit of the knapsack does not exceed.



Knapsack Problem

Unit-5: Dynamic Programming

0/1 Knapsack Problem:

- In 0/1 Knapsack Problem,
  - As the name suggests, items are indivisible here.
  - We can not take the fraction of any item.
  - We have to either take an item completely or leave it completely.
  - It is solved using dynamic programming approach.

Statement: A thief has a bag or knapsack that can contain maximum weight W of his loot. There are n items and the weight of i^{th} item is Wi and it worth Vi . An amount of item can be put into the bag is 0 or 1 i.e. xi is 0 or 1. Here the objective is to collect the items that maximize the total profit earned.

Unit-5: Dynamic Programming

0/1 Knapsack Problem Using Dynamic Programming-

- Consider-
  - Knapsack weight capacity = w
  - Number of items each having some weight and value = n
  - 0/1 knapsack problem is solved using dynamic programming in the following steps-
- **Step-01:**

  - Draw a table say 'T' with (n+1) number of rows and (w+1) number of columns.
  - Fill all the boxes of 0^{th} row and 0^{th} column with zeroes as shown-

	0	1	2	3	.....	W
0	0	0	0	0	.....	0
1	0					
2	0					
	.....					
n	0					

T-Table

Unit-5: Dynamic Programming

0/1 Knapsack Problem Using Dynamic Programming-

Step-02:

Start filling the table row wise top to bottom from left to right.

Use the following formula-

$$T(i, j) = \max \{ T(i-1, j), value_i + T(i-1, j - weight_i) \}$$

Here, T(i , j) = maximum value of the selected items if we can take items 1 to i and have weight restrictions of j.

- This step leads to completely filling the table.
- Then, value of the last box represents the maximum possible value that can be put into the knapsack.

Unit-5: Dynamic Programming

0/1 Knapsack Problem Using Dynamic Programming-

Step-03:

To identify the items that must be put into the knapsack to obtain that maximum profit,

- Consider the last column of the table.
- Start scanning the entries from bottom to top.
- On encountering an entry whose value is not same as the value stored in the entry immediately above it, mark the row label of that entry.
- After all the entries are scanned, the marked labels represent the items that must be put into the knapsack.

Unit-5: Dynamic Programming

0/1 Knapsack Problem Using Dynamic Programming-Example

- Find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach. Consider-

n = 4

w = 5 kg

(w1, w2, w3, w4) = (2, 3, 4, 5)

(b1, b2, b3, b4) = (3, 4, 5, 6)

Solution-

Given-

Knapsack capacity (w) = 5 kg

Number of items (n) = 4

Unit-5: Dynamic Programming

0/1 Knapsack Problem Using Dynamic Programming-Example

Step-01:

- Draw a table say 'T' with (n+1) = 4 + 1 = 5 number of rows and (w+1) = 5 + 1 = 6 number of columns.
- Fill all the boxes of 0^{th} row and 0^{th} column with 0.

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

T-Table

Unit-5: Dynamic Programming

0/1 Knapsack Problem Using Dynamic Programming-Example

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$$T(i, j) = \max \{ T(i-1, j), value_i + T(i-1, j - weight_i) \}$$

Finding T(1,1)-

We have,

- i = 1
- j = 1
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,1) = max { T(1-1 , 1) , 3 + T(1-1 , 1-2) }

T(1,1) = max { T(0,1) , 3 + T(0,-1) }

T(1,1) = T(0,1) { Ignore T(0,-1) }

T(1,1) = 0

Unit-5: Dynamic Programming

0/1 Knapsack Problem Using Dynamic Programming-Example

Finding T(1,2)-

We have,

- i = 1
- j = 2
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,2) = max { T(1-1 , 2) , 3 + T(1-1 , 2-2) }

T(1,2) = max { T(0,2) , 3 + T(0,0) }

T(1,2) = max {0 , 3+0}

T(1,2) = 3

Finding T(1,3)-

We have,

- i = 1
- j = 3
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,3) = max { T(1-1 , 3) , 3 + T(1-1 , 3-2) }

T(1,3) = max { T(0,3) , 3 + T(0,1) }

T(1,3) = max {0 , 3+0}

T(1,3) = 3

Unit-5: Dynamic Programming

0/1 Knapsack Problem Using Dynamic Programming-Example

Finding T(1,4)-

We have,

- i = 1
- j = 4
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,4) = max { T(1-1 , 4) , 3 + T(1-1 , 4-2) }

T(1,4) = max { T(0,4) , 3 + T(0,2) }

T(1,4) = max {0 , 3+0}

T(1,4) = 3

Finding T(1,5)-

We have,

- i = 1
- j = 5
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,5) = max { T(1-1 , 5) , 3 + T(1-1 , 5-2) }

T(1,5) = max { T(0,5) , 3 + T(0,3) }

T(1,5) = max {0 , 3+0}

T(1,5) = 3

Unit-5: Dynamic Programming

0/1 Knapsack Problem Using Dynamic Programming-Example

Finding T(2,1)-

We have,

- i = 2
- j = 1
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,1) = max { T(2-1 , 1) , 4 + T(2-1 , 1-3) }

T(2,1) = max { T(1,1) , 4 + T(1,-2) }

T(2,1) = T(1,1) { Ignore T(1,-2) }

T(2,1) = 0

Finding T(2,2)-

We have,

- i = 2
- j = 2
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,2) = max { T(2-1 , 2) , 4 + T(2-1 , 2-3) }

T(2,2) = max { T(1,2) , 4 + T(1,-1) }

T(2,2) = T(1,2) { Ignore T(1,-1) }

T(2,2) = 3

Unit-5: Dynamic Programming

0/1 Knapsack Problem Using Dynamic Programming-Example

Finding T(2,3)-

We have,

- i = 2
- j = 3
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,3) = max { T(2-1 , 3) , 4 + T(2-1 , 3-3) }

T(2,3) = max { T(1,3) , 4 + T(1,0) }

T(2,3) = max { 3 , 4+0 }

T(2,3) = 4

Finding T(2,4)-

We have,

- i = 2
- j = 4
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,4) = max { T(2-1 , 4) , 4 + T(2-1 , 4-3) }

T(2,4) = max { T(1,4) , 4 + T(1,1) }

T(2,4) = max { 3 , 4+0 }

T(2,4) = 4

Unit-5: Dynamic Programming

Finding T(2,5)-

We have,

- i = 2
- j = 5
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,5) = max { T(2-1 , 5) , 4 + T(2-1 , 5-3) }

T(2,5) = max { T(1,5) , 4 + T(1,2) }

T(2,5) = max { 3 , 4+3 }

T(2,5) = 7

Compute remaining :

T(3,1), T(3,2), T(3,3), T(3,4), T(3,5)

T(4,1), T(4,2), T(4,3), T(4,4), T(4,5)

And fill the table.

Unit-5: Dynamic Programming

0/1 Knapsack Problem Using Dynamic Programming-Example

Similarly, compute all the entries.

After all the entries are computed and filled in the table, we get the following table-

	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

T-Table

- The last entry represents the maximum possible value that can be put into the knapsack.
- So, maximum possible value that can be put into the knapsack = 7.

Unit-5: Dynamic Programming

0/1 Knapsack Problem Using Dynamic Programming-

The algorithm takes as input maximum weight W, the number of items n, two arrays v[] for values of items and w[] for weight of items. Let us assume that the table c[i,w] is the value of solution for items 1 to i and maximum weight w. Then we can define recurrence relation for 0/1 knapsack problem as

$$C[i,w] = \begin{cases} 0 & \text{if } i=0 \text{ or } w=0 \\ C[i-1,w] & \text{if } wi > w \\ \text{Max}\{vi + C[i-1,w-wi],\ c[i-1,w]\} & \text{if } i>0 \text{ and } w>wi \end{cases}$$

Unit-5: Dynamic Programming

0/1 Knapsack Problem Using Dynamic Programming-Pseudo Code

```
DynaKnapsack(W,n,v,w)
{
        for(w=0; w<=W; w++)
                C[0,w] = 0;
        for(i=1; i<=n; i++)
                C[i,0] = 0;
        for(i=1; i<=n; i++)
        {
                for(w=1; w<=W;w++xw1z)
                {
                        if(w[i]<w)

                {
                        if v[i] +C[i-1,w-w[i]] > C[i-1,w]
                        {
                                C[i,w] = v[i] +C[i-1,w-w[i]];
                        }
                        else
                        {
                                C[i,w] = C[i-1,w];
                        }
                }
                else
                {
                        C[i,w] = C[i-1,w];
                }
        }
    }
}
```

Unit-5: Dynamic Programming

0/1 Knapsack Problem Using Dynamic Programming

- **Analysis:**
  - For run time analysis examining the above algorithm the overall run time of the algorithm is $O(n^w)$.

Unit-5: Dynamic Programming

0/1 Knapsack Problem Using Dynamic Programming

- **Example**

- **Let the problem instance be with 7 items where v[ ] = {2,3,3,4,4,5,7} and w[] = {3,5,7,4,3,9,2} and W = 9. Find the maximum profit earned by using 0/1 knapsack problem.**

- **Consider the problem having weights and profits are:**

  Weights: {3, 4, 6, 5}

  Profits: {2, 3, 1, 4}

  The weight of the knapsack is 8 kg and The number of items is 4. Find the maximum profit earned by using 0/1 knapsack problem.

Unit-5: Dynamic Programming

Longest Common Sub-sequence problem (LCS):

- The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

- If S1 and S2 are the two given sequences then,
  - Z is the common subsequence of S1 and S2
  - if Z is a subsequence of both S1 and S2.
  - Furthermore, Z must be a strictly increasing sequence of the indices of both S1 and S2.

Unit-5: Dynamic Programming

Longest Common Sub-sequence problem (LCS):

- In a strictly increasing sequence, the indices of the elements chosen from the original sequences must be in ascending order in Z.
- If
  - S1 = {B, C, D, A, A, C, D}
- Then, {A, D, B} cannot be a subsequence of S1 as the order of the elements is not the same (ie. not strictly increasing sequence).

Unit-5: Dynamic Programming

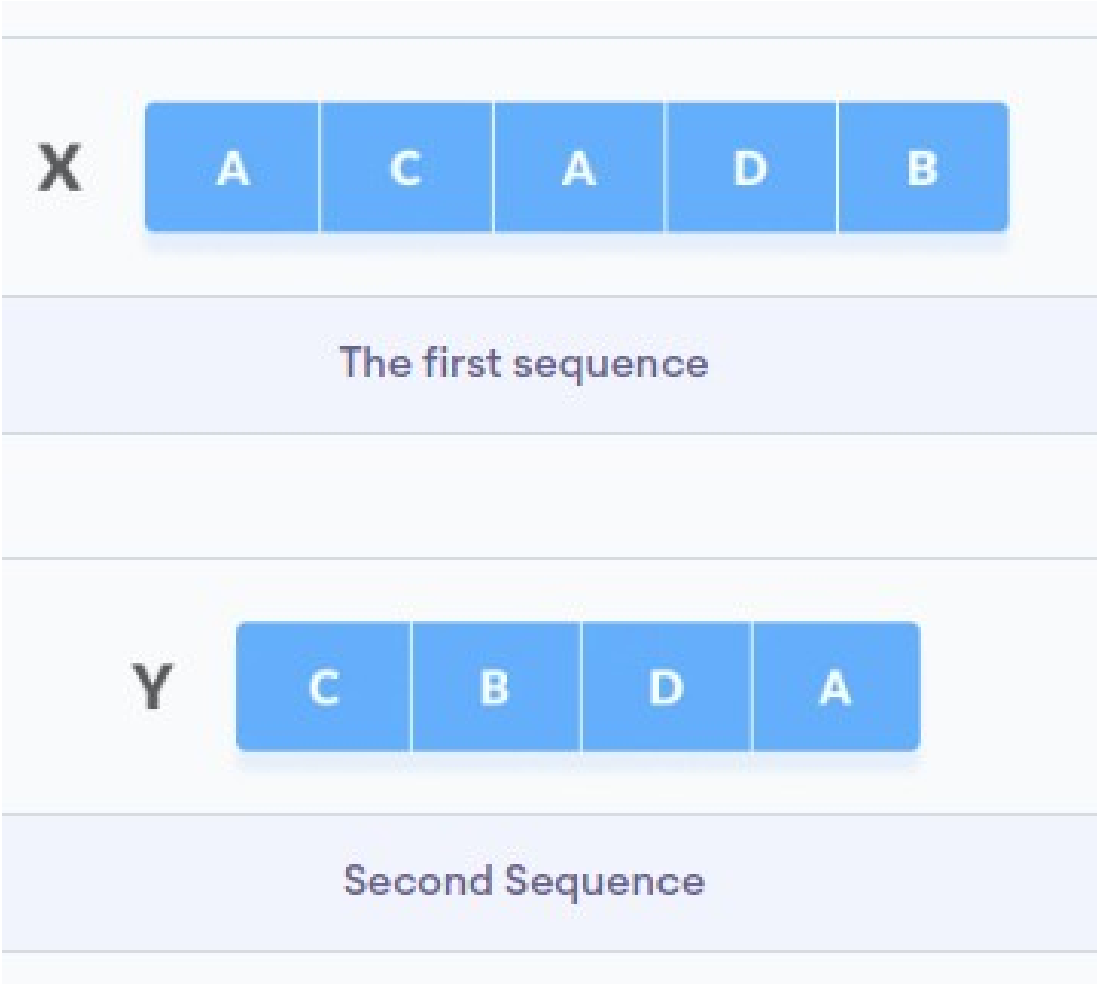
Longest Common Sub-sequence problem (LCS):

- Let us understand LCS with an example.
- If
  - S1 = {B, C, D, A, A, C, D}
  - S2 = {A, C, D, B, A, C}
- Then, common subsequences are {B, C}, {C, D, A, C}, {D, A, C}, {A, A, C}, {A, C}, {C, D}, ...
- Among these subsequences, {C, D, A, C} is the longest common subsequence. We are going to find this longest common subsequence using dynamic programming.

Unit-5: Dynamic Programming

Longest Common Sub-sequence problem (LCS):

- Let us take two sequences:



Unit-5: Dynamic Programming

Longest Common Sub-sequence problem (LCS):

- The following steps are followed for finding the longest common subsequence.
- **Step 1:** Create a table of dimension n+1*m+1 where n and m are the lengths of X and Y respectively. The first row and the first column are filled with zeros.

		C	B	D	A
	0	0	0	0	0
A	0				
C	0				
A	0				
D	0				
B	0				

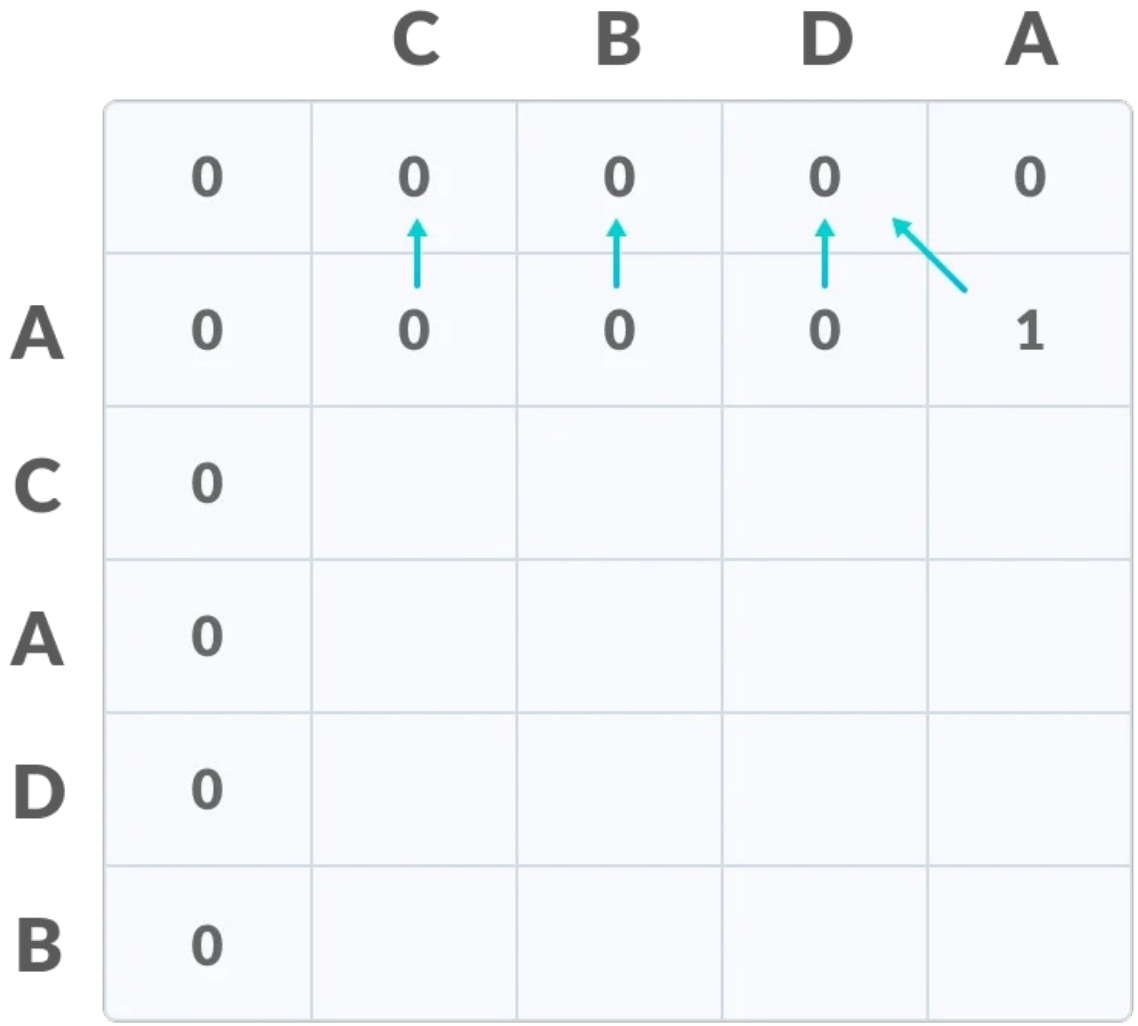
Unit-5: Dynamic Programming

Longest Common Sub-sequence problem (LCS):

- The following steps are followed for finding the longest common subsequence.
  - **Step 2:** Fill each cell of the table using the following logic.
  - **Step 2.1:** If the character corresponding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
  - **Step 2.2:** Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to vertical one.

Unit-5: Dynamic Programming

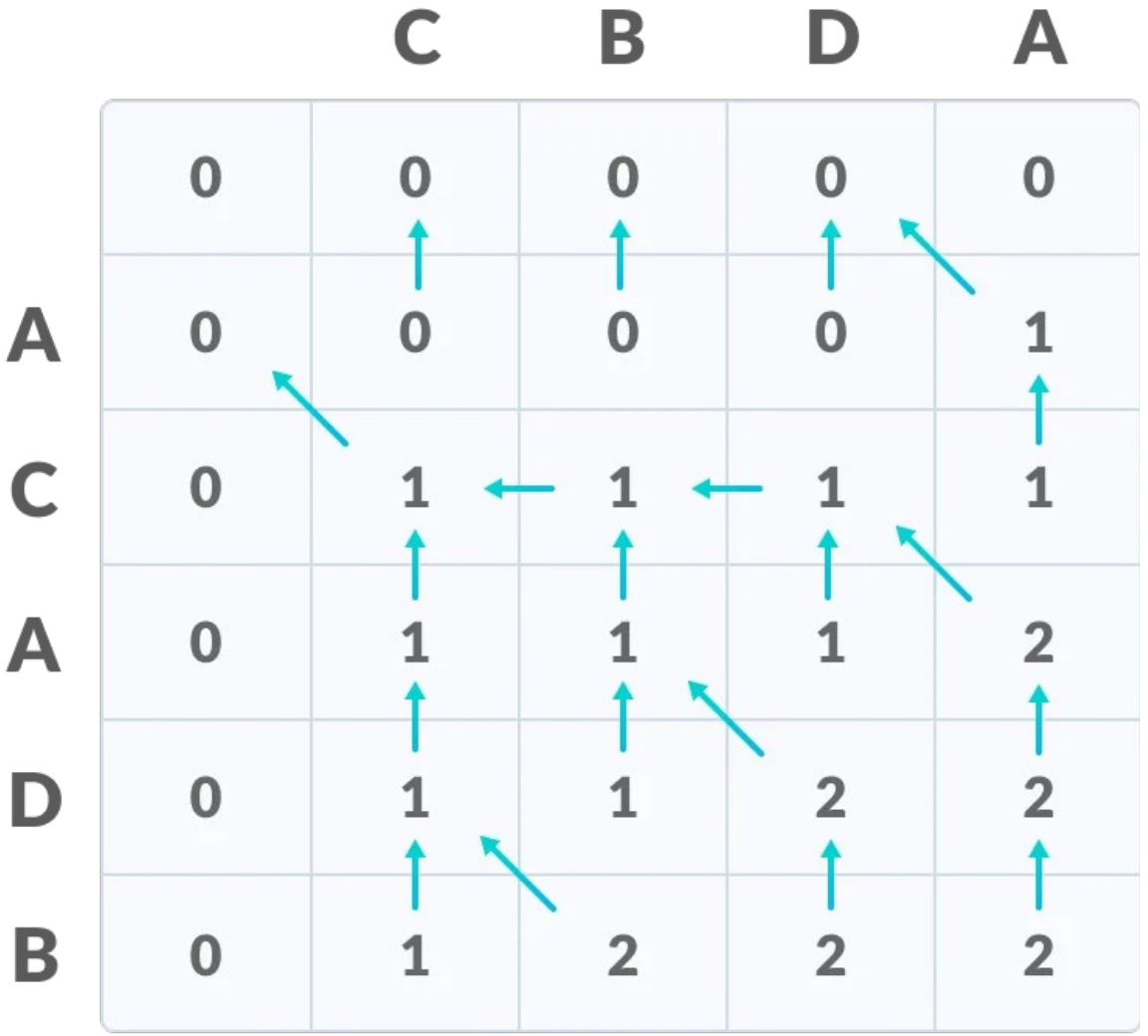
Longest Common Sub-sequence problem (LCS):



Unit-5: Dynamic Programming

Longest Common Sub-sequence problem (LCS):

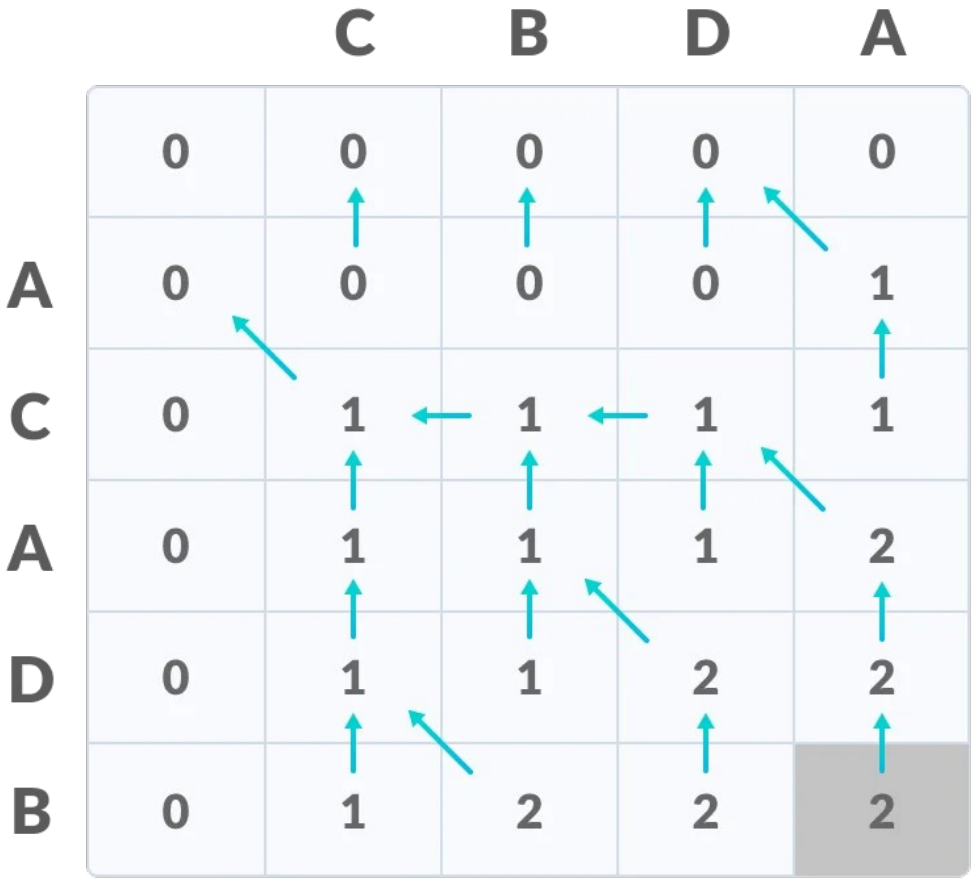
- **Step 2** is repeated until the table is filled.



Unit-5: Dynamic Programming

Longest Common Sub-sequence problem (LCS):

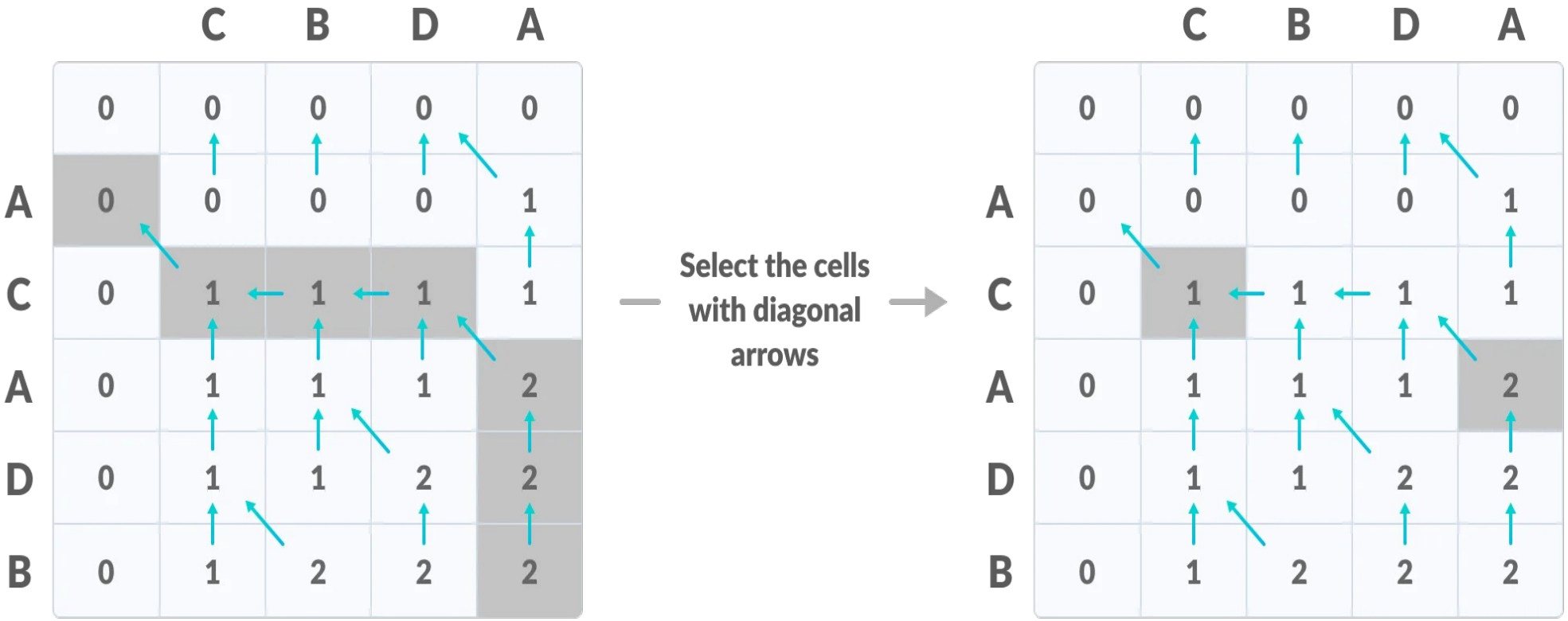
- The value in the last row and the last column is the length of the longest common subsequence.



Unit-5: Dynamic Programming

Longest Common Sub-sequence problem (LCS):

- In order to find the longest common subsequence, start from the last element and follow the direction of the arrow.



- Thus, the longest common subsequence is (C,A).

Unit-5: Dynamic Programming

Longest Common Sub-sequence problem (LCS):

- **Examples: Find the longest common sub-sequences for the following.**
  - LCS for input Sequences x= “ABCDGH” and Y= “AEDFHR” is “ADH” of length 3.
  - LCS for input Sequences X= “AGGTAB” and Y= “GXTXAYB” is “GTAB” of length 4.

Unit-5: Dynamic Programming

Floyd Warshall Algorithm:

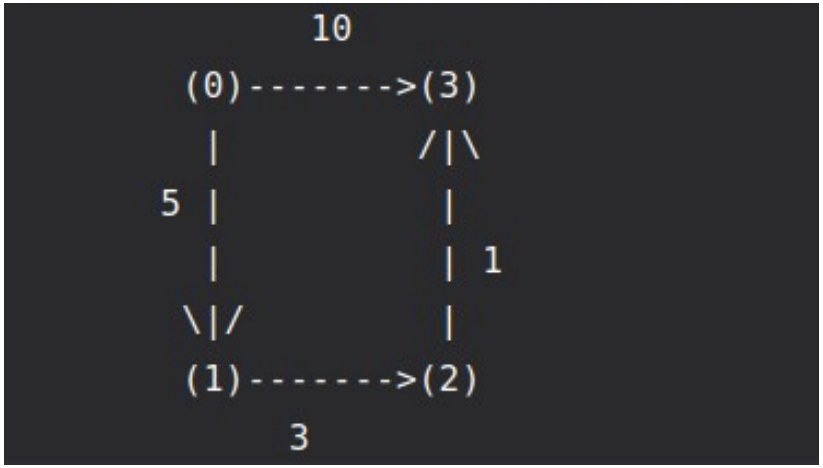
The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem.

- The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

- **Input:**

```
graph[][] = { {0,   5,  INF, 10},
              {INF,  0,  3,  INF},
              {INF, INF, 0,   1},
              {INF, INF, INF, 0} }
```

which represents the following graph



Unit-5: Dynamic Programming

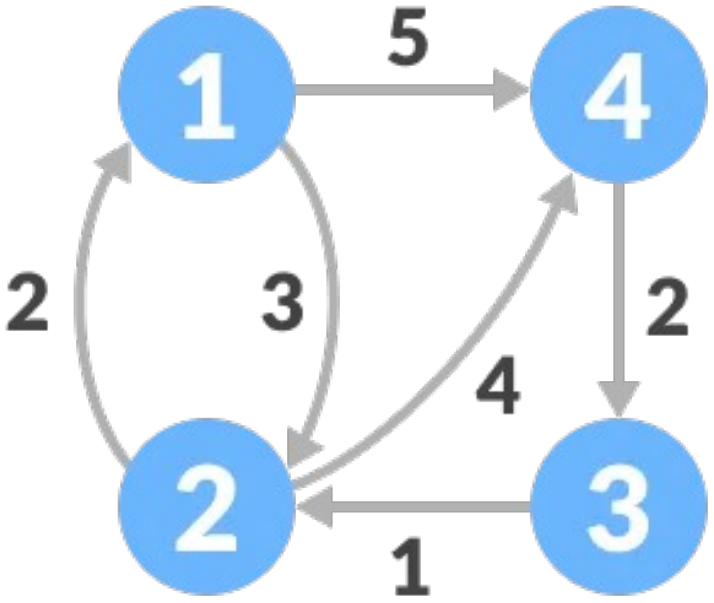
Floyd Warshall Algorithm:

- Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph.
- This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).
- A weighted graph is a graph in which each edge has a numerical value associated with it.
- Floyd-Warhshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, or WFI algorithm.
- This algorithm follows the dynamic programming approach to find the shortest paths.

Unit-5: Dynamic Programming

Floyd Warshall Algorithm:

- How Floyd-Warshall Algorithm Works?
  - Let the given graph be:



Unit-5: Dynamic Programming

Floyd Warshall Algorithm:

- Follow the steps below to find the shortest path between all the pairs of vertices.
  - Create a matrix A^0 of dimension n*n where n is the number of vertices. The row and the column are indexed as i and j respectively. (i and j are the vertices of the graph).
  - Each cell A[i][j] is filled with the distance from the i^{th} vertex to the j^{th} vertex. If there is no path from i^{th} vertex to j^{th} vertex, the cell is left as infinity.

Unit-5: Dynamic Programming

Floyd Warshall Algorithm:

- How Floyd-Warshall Algorithm Works?

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Unit-5: Dynamic Programming

Floyd Warshall Algorithm:

- How Floyd-Warshall Algorithm Works?
  - Now, create a matrix A^1 using matrix A^0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.
  - Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex.
  - A[i][j] is filled with (A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j]).
  - That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with A[i][k] + A[k][j].
  - In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k.

Floyd Warshall Algorithm:

How Floyd-Warshall Algorithm Works?

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ \infty & 1 & 0 & 8 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

For example: For $A^1[2, 4]$, the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since 4 < 7, A0[2, 4] is filled with 4.

Unit-5: Dynamic Programming

Floyd Warshall Algorithm:

- How Floyd-Warshall Algorithm Works?
  - Similarly, A^2 is created using A^1. The elements in the second column and the second row are left as they are.
  - In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in step 2.

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & & \\ 2 & 0 & 9 & 4 \\ & 1 & 0 & \\ & \infty & & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Unit-5: Dynamic Programming

Floyd Warshall Algorithm:

- How Floyd-Warshall Algorithm Works?
  - Similarly, A^3 and A^4 is also created.

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & \infty & \\ & 0 & 9 & \\ \infty & 1 & 0 & 8 \\ & & 2 & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

Unit-5: Dynamic Programming

Floyd Warshall Algorithm:

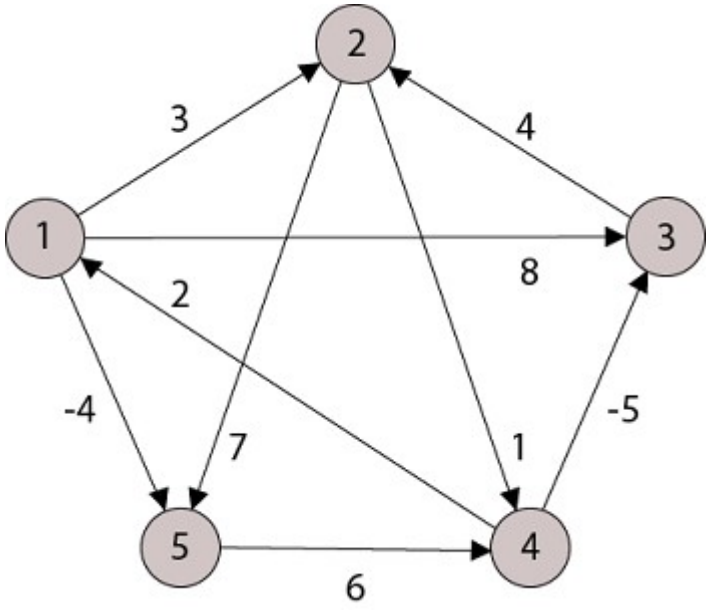
- How Floyd-Warshall Algorithm Works?
  - A^4 gives the shortest path between each pair of vertices.

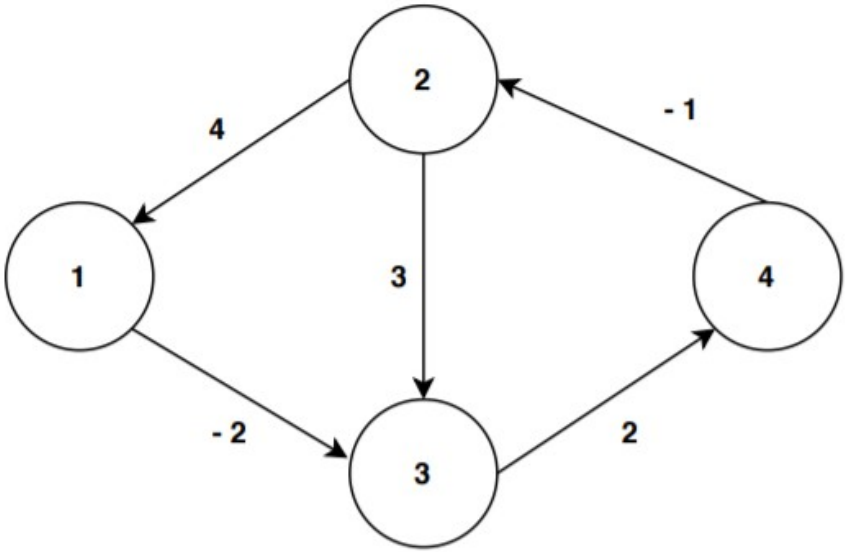
$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & & 5 \\ & 0 & & 4 \\ & & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

Unit-5: Dynamic Programming

Floyd Warshall Algorithm:

- Apply Floyd-Warshall algorithm for constructing the shortest path for the given graphs.





Unit-5: Dynamic Programming

Floyd Warshall Algorithm: Pseudo code:

Algorithm 1: Pseudocode of Floyd-Warshall Algorithm

```
Data: A directed weighted graph G(V, E)
Result: Shortest path between each pair of vertices in G
for each d ∈ V do
    distance[d][d] ← 0;
end
for each edge (s, p) ∈ E do
    distance[s][p] ← weight(s, p);
end
n = cardinality(V);
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            if distance[i][j] > distance[i][k] + distance[k][j] then
                distance[i][j] ← distance[i][k] + distance[k][j];
            end
        end
    end
end
```

Unit-5: Dynamic Programming

Floyd Warshall Algorithm:

- Floyd Warshall Algorithm Complexity
- Time Complexity
  - There are three loops. Each loop has constant complexities. So, the time complexity of the Floyd-Warshall algorithm is $O(n^3)$.
- Space Complexity
  - The space complexity of the Floyd-Warshall algorithm is $O(n^2)$.

Unit-5: Dynamic Programming

Travelling Salesman Problem:

- Problem Statement
  - A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once.
  - What is the shortest possible route that he visits each city exactly once and returns to the origin city?

Unit-5: Dynamic Programming

Travelling Salesman Problem:

- Travelling salesman problem is the most well known computational problem.
- We can use brute-force approach to evaluate every possible tour and select the best one. For n number of vertices in a graph, there are (n - 1)! num.ber of possibilities.
- Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

Unit-5: Dynamic Programming

Travelling Salesman Problem:

- Let us consider a graph G = (V, E), where V is a set of cities and E is a set of weighted edges.
- An edge e(u, v) represents that vertices u and v are connected. Distance between vertex u and v is d(u, v), which should be non-negative.
- Suppose we have started at city 1 and after visiting some cities now we are in city j.
- Hence, this is a partial tour. We certainly need to know j, since this will determine which cities are most convenient to visit next.
- We also need to know all the cities visited so far, so that we don't repeat any of them.
- Hence, this is an appropriate sub-problem.

Unit-5: Dynamic Programming

Travelling Salesman Problem:

- For a subset of cities S Є {1, 2, 3, ... , n} that includes 1, and j Є S, let C(S, j) be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.
- When |S| > 1, we define C(S, 1) = $\propto$ since the path cannot start and end at 1.
- Now, let express C(S, j) in terms of smaller sub-problems. We need to start at 1 and end at j. We should select the next city in such a way that

  - **$C(S,j)=\min C\{d(i,j)+(S-\{j\},i)\}$ where $i \in S$ and $i \neq j$ $c(S,j)$**
    $=\min C\{d(i,j)+(s-\{j\},i)\}$ where $i \in S$ and $i \neq j$

Unit-5: Dynamic Programming

Travelling Salesman Problem:

```
Algorithm: Traveling-Salesman-Problem
C ({1}, 1) = 0
for s = 2 to n do
   for all subsets S ∈ {1, 2, 3, … , n} of size s and containing 1
      C (S, 1) = ∞
   for all j ∈ S and j ≠ 1
      C (S, j) = min {C (S – {j}, i) + d(i, j) for i ∈ S and i ≠ j}
Return minj C ({1, 2, 3, …, n}, j) + d(j, i)
```

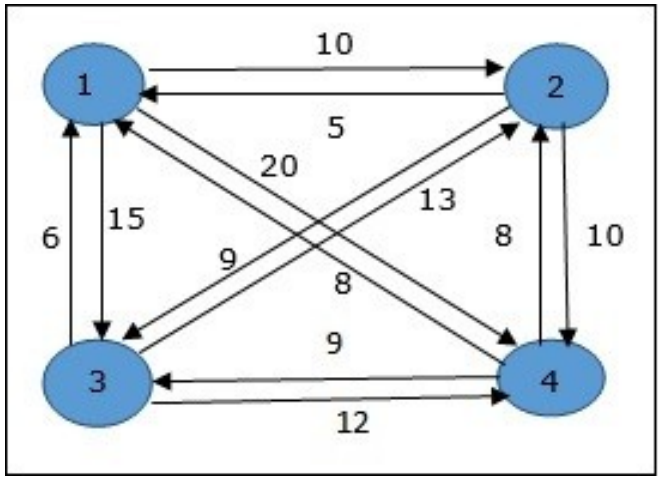
Analysis

There are at the most $2^n . n$ sub-problems and each one takes linear time to solve.

Therefore, the total running time is $O(2^n . n^2)$.

Unit-5: Dynamic Programming

Travelling Salesman Problem: Example



From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

Unit-5: Dynamic Programming

Travelling Salesman Problem: Example

- Clearly, g(i,s) =min{Cik + g(jk, S-{k})} where kbelongs to S
- S = Φ= Ci1
  - Cost(2,Φ,1)=d(2,1)=5Cost(2,Φ,1)=d(2,1)=5
  - Cost(3,Φ,1)=d(3,1)=6Cost(3,Φ,1)=d(3,1)=6
  - Cost(4,Φ,1)=d(4,1)=8Cost(4,Φ,1)=d(4,1)=8

Unit-5: Dynamic Programming

Travelling Salesman Problem: Example

- S = 1
- Cost(i,s)=min{Cost(j,s–(j))+d[i,j]}Cost(i,s)=min{Cost(j,s)−(j))+d[i,j]}
  - Cost(2,{3},1)=d[2,3]+Cost(3,Φ,1)=9+6=15 cost(2,{3},1)=d[2,3]+cost(3,Φ,1)=9+6=15
  - Cost(2,{4},1)=d[2,4]+Cost(4,Φ,1)=10+8=1cost(2{4},1)=d[2,4]+cost(4,Φ,1)=10+8=18
  - Cost({2},1)=d[3,2]+Cost(2,Φ,1)=13+5=18cost(3{2},1)=d[3,2]+cost(2,Φ,1)=13+5=18
  - Cost({4},1)=d[3,4]+Cost(4,Φ,1)=12+8=20cost(3{4},1)=d[3,4]+cost(4,Φ,1)=12+8=20
  - Cost(4,{3},1)=d[4,3]+Cost(3,Φ,1)=9+6=15cost(4,{3},1)=d[4,3]+cost(3,Φ,1)=9+6=15
  - Cost(4,{2},1)=d[4,2]+Cost(2,Φ,1)=8+5=13cost(4,{2},1)=d[4,2]+cost(2,Φ,1)=8+5=13

Unit-5: Dynamic Programming

Travelling Salesman Problem: Example

- S = 2

$$Cost(2,\{3,4\},1)$$

$$= \begin{cases} d[2,3] + Cost(3,\{4\},1) = 9 + 20 = 29 \\ d[2,4] + Cost(4,\{3\},1) = 10 + 15 = 25 = 25 Cost(2,\{3,4\},1) \\ \{d[2,3] + cost(3,\{4\},1) = 9 + 20 = 29 d[2,4] + Cost(4,\{3\},1) = 10 + 15 = 25 \end{cases}$$

$$= 25$$

$$Cost(3,\{2,4\},1)$$

$$= \begin{cases} d[3,2] + Cost(2,\{4\},1) = 13 + 18 = 31 \\ d[3,4] + Cost(4,\{2\},1) = 12 + 13 = 25 = 25 Cost(3,\{2,4\},1) \\ \{d[3,2] + cost(2,\{4\},1) = 13 + 18 = 31 d[3,4] + Cost(4,\{2\},1) = 12 + 13 = 25 \end{cases}$$

$$= 25$$

$$Cost(4,\{2,3\},1)$$

$$= \begin{cases} d[4,2] + Cost(2,\{3\},1) = 8 + 15 = 23 \\ d[4,3] + Cost(3,\{2\},1) = 9 + 18 = 27 = 23 Cost(4,\{2,3\},1) \\ \{d[4,2] + cost(2,\{3\},1) = 8 + 15 = 23 d[4,3] + Cost(3,\{2\},1) = 9 + 18 = 27 \end{cases}$$

$$= 23$$

Unit-5: Dynamic Programming

Travelling Salesman Problem: Example

- S = 3

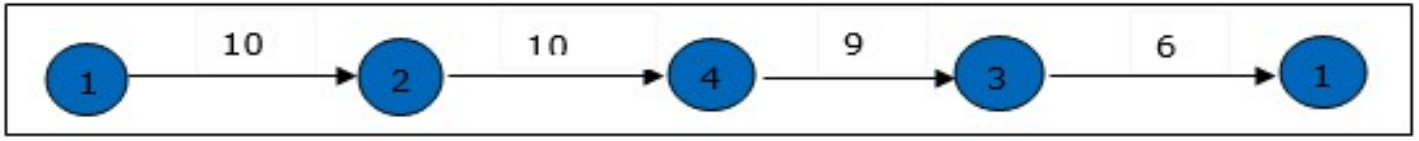
$$Cost(1,\{2,3,4\},1)$$

$$= \begin{cases} d[1,2] + Cost(2,\{3,4\},1) = 10+25 = 35 \\ d[1,3] + Cost(3,\{2,4\},1) = 15+25 = 40 \\ d[1,4] + Cost(4,\{2,3\},1) = 20+23 = 43 = 35 cost(1,\{2,3,4\}),1) \\ d[1,2] + cost(2,\{3,4\},1) = 10+25 = 35 \\ d[1,3] + cost(3,\{2,4\},1) = 15+25 = 40 \\ d[1,4] + cost(4,\{2,3\},1) = 20+23 = 43 = 35 \end{cases}$$

The minimum cost path is 35.

Start from cost **{1, {2, 3, 4}, 1}**, we get the minimum value for **d [1, 2]**. When **s = 3**, select the path from 1 to 2 (cost is 10) then go backwards. When **s = 2**, we get the minimum value for **d [4, 2]**. Select the path from 2 to 4 (cost is 10) then go backwards.

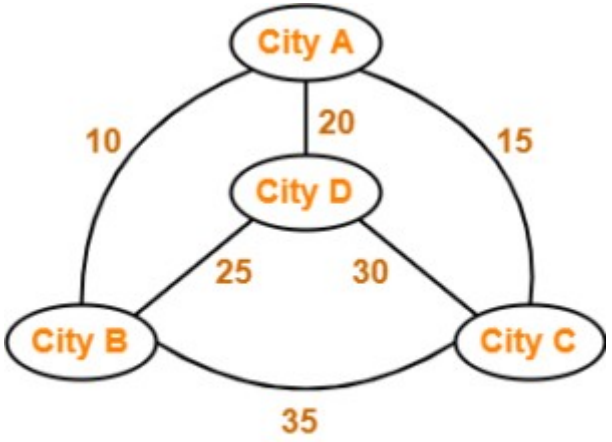
When **s = 1**, we get the minimum value for **d [4, 3]**. Selecting path 4 to 3 (cost is 9), then we shall go to then go to **s = Φ** step. We get the minimum value for **d [3, 1]** (cost is 6).



Unit-5: Dynamic Programming

Travelling Salesman Problem: Example

- For the following graph find the minimum cost of tour



Travelling Salesman Problem

Unit-5: Dynamic Programming

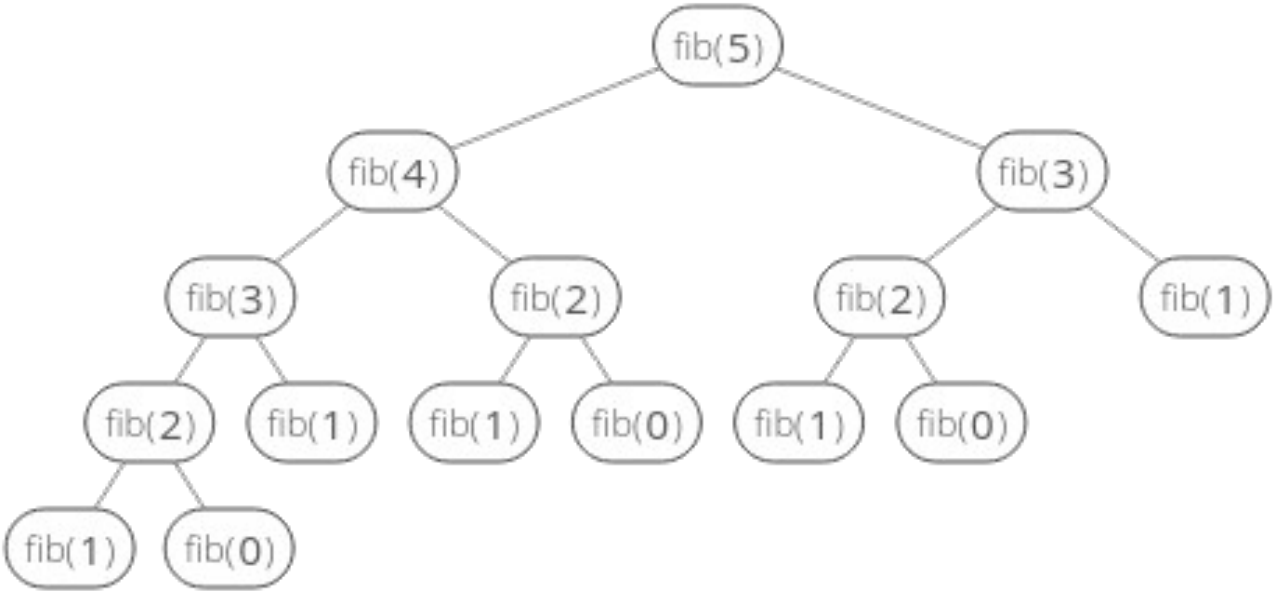
Concepts of Memoization:

- In computing, memoization or memoisation is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.
- Most of the Dynamic Programming problems are solved in two ways:
  - Tabulation: Bottom Up
  - Memoization: Top Down

Unit-5: Dynamic Programming

Concepts of Memoization:

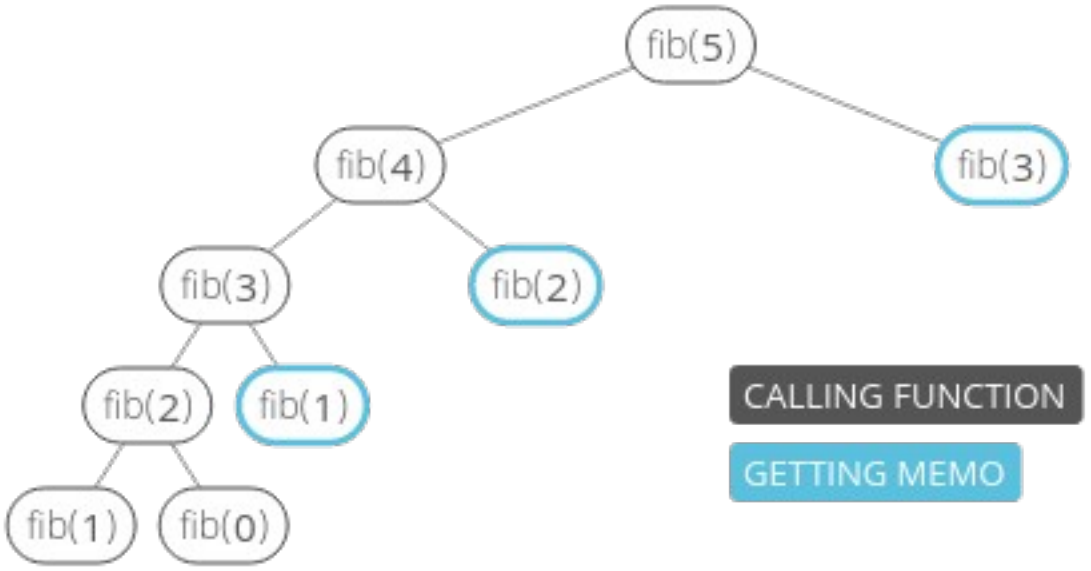
- Memoization ensures that a method doesn't run for the same inputs more than once by keeping a record of the results for the given inputs (usually in a hash map).
- We can imagine the recursive calls of this method as a tree, where the two children of a node are the two recursive calls it makes.
- We can see that the tree quickly branches out of control:



Unit-5: Dynamic Programming

Concepts of Memoization:

- To avoid the duplicate work caused by the branching, we can wrap the method in a class that stores an instance variable, memo, that maps inputs to outputs.
- Then we simply check memo to see if we can avoid computing the answer for any given input, and save the results of any calculations to memo.
- Now in our recurrence tree, no node appears more than twice:



Concepts of Memoization:

- Memoization is a common strategy for dynamic programming problems, which are problems where the solution is composed of solutions to the same problem with smaller inputs (as with the Fibonacci problem, above).
- The other common strategy for dynamic programming problems is going bottom-up, which is usually cleaner and often more efficient.

Unit-5: Dynamic Programming

- Thank You!