**Chapter-7**

# DYNAMIC PROGRAMMING

**Prof. Ritesh Upadhyay**

Faculty of Engineering and Technology

Ganpat University
॥ विद्यया समाजोत्कर्षः ॥

## Introduction of Dynamic Programming

- Dynamic Programming is the most powerful design technique for solving optimization problems.
- Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into sub problems and stores the results of sub problems to avoid computing the same results again.
- Dynamic Programming solves each sub problems just once and stores the result in a table so that it can be repeatedly retrieved if needed again.
- In this chapter we will try to solve the problems for which we failed to get the optimal solutions using other techniques (say, Divide & Conquer and Greedy methods).
- The term Programming is not related to coding but it is from literature, and means filling tables (similar to Linear Programming).

## What is Dynamic Programming strategy?

Dynamic programming and memoization work together. The main difference between dynamic programming and divide and conquer is that in the case of divide and conquer(DC), sub problems are independent, whereas in DP there can be an overlap of sub problems. By using memoization [maintaining a table of sub problems already solved], dynamic programming reduces the exponential complexity to polynomial complexity (O(n^2), O(n^3), etc.) for many problems.

The major components of DP are:

- **Recursion**: Solves sub problems recursively.
- **Memoization**: Stores already computed values in table (Memoization means caching).

Dynamic Programming = Recursion + Memoization

## Characteristics/ Properties of Dynamic Programming

Following are the two main properties of a problem that suggests that the given problem can be solved using Dynamic programming.

1. **Optimal Substructure**: If an optimal solution contains optimal sub solutions then a problem exhibits optimal substructure.
2. **Overlapping sub-problems**: When a recursive algorithm would visit the same sub-problems repeatedly, then a problem has overlapping sub-problems.

# Dynamic Programming Approaches

Basically there are two approaches for solving DP problems:

1. Bottom-up dynamic programming
2. Top-down dynamic programming

**Bottom-up Dynamic Programming :** In this method, we evaluate the function starting with the smallest possible input argument value and then we step through possible values, slowly increasing the input argument value. While computing the values we store all computed values in a table (memory). As larger arguments are evaluated, pre-computed values for smaller arguments can be used

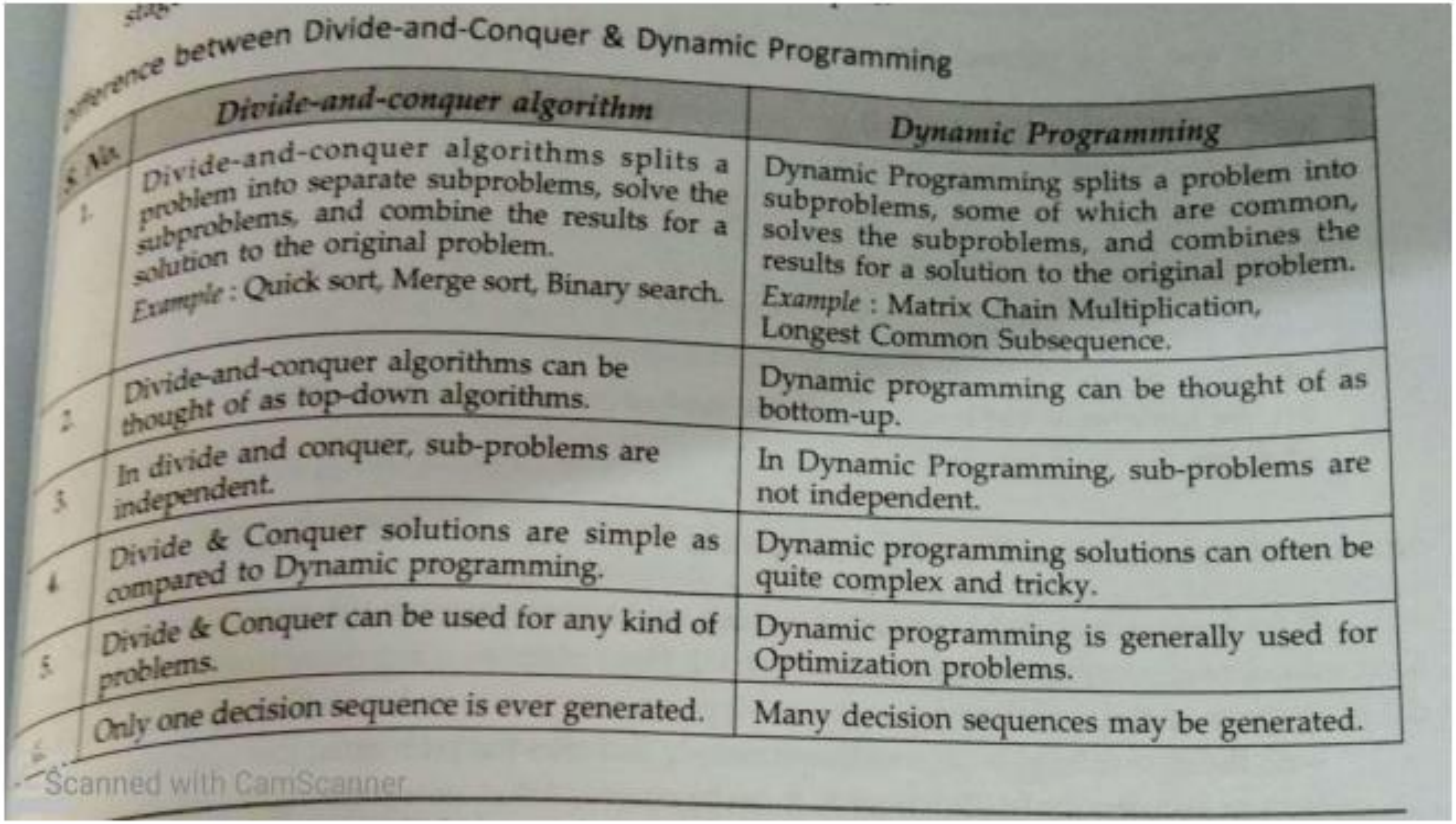
**Top-down Dynamic Programming:** In this method, the problem is broken into sub problems; each of these sub problems is solved; and the solutions remembered, in case they need to be solved. Also, we save each computed value as the final action of the recursive function, and as the first action we check if pre-computed value exists.

## Examples of Dynamic Programming Algorithms

- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, edit distance.
- Algorithms on graphs can be solved efficiently: Bellman-Ford algorithm for finding the shortest distance in a graph, Floyd’s All-Pairs shortest path algorithm, etc.

1. Chain matrix multiplication
2. Subset Sum Problem
3. 0/1 Knapsack
4. Making change problem using Dynamic Programming
5. Binomial coefficients Problem
6. Longest Common sub-sequences
7. Travelling salesman problem, and many more

## Differentiate between Divide & Conquer Method vs Dynamic Programming.



Difference between Divide-and-Conquer & Dynamic Programming

| S. No. | *Divide-and-conquer algorithm* | *Dynamic Programming* |
|---|---|---|
| 1. | Divide-and-conquer algorithms splits a problem into separate subproblems, solve the subproblems, and combine the results for a solution to the original problem. *Example* : Quick sort, Merge sort, Binary search. | Dynamic Programming splits a problem into subproblems, some of which are common, solves the subproblems, and combines the results for a solution to the original problem. *Example* : Matrix Chain Multiplication, Longest Common Subsequence. |
| 2. | Divide-and-conquer algorithms can be thought of as top-down algorithms. | Dynamic programming can be thought of as bottom-up. |
| 3. | In divide and conquer, sub-problems are independent. | In Dynamic Programming, sub-problems are not independent. |
| 4. | Divide & Conquer solutions are simple as compared to Dynamic programming. | Dynamic programming solutions can often be quite complex and tricky. |
| 5. | Divide & Conquer can be used for any kind of problems. | Dynamic programming is generally used for Optimization problems. |
| 6. | Only one decision sequence is ever generated. | Many decision sequences may be generated. |

## Differentiate between Greedy Technique 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. |
| It is guaranteed that Dynamic Programming will generate an optimal solution using Principle of Optimality | 3. In Greedy Method, there is no such guarantee of getting Optimal Solution. |
| 4. Example: 0/1 Knapsack | 4. Example: Fractional Knapsack |

## Fibonacci sequence/Series by DP

Fibonacci sequence is the sequence of numbers in which every next item is the total of the previous two items. And each number of the Fibonacci sequence is called Fibonacci number.

Example: 0 ,1,1,2,3,5,8,13,21,...................... is a Fibonacci sequence.

The Fibonacci numbers F(n) are defined as follows:

```
int Fib (int n)
{
 If (n ≤1)
return n;
return Fib (n - 2) + Fib (n - 1);
}
```

# Fibonacci sequence/Series by DP

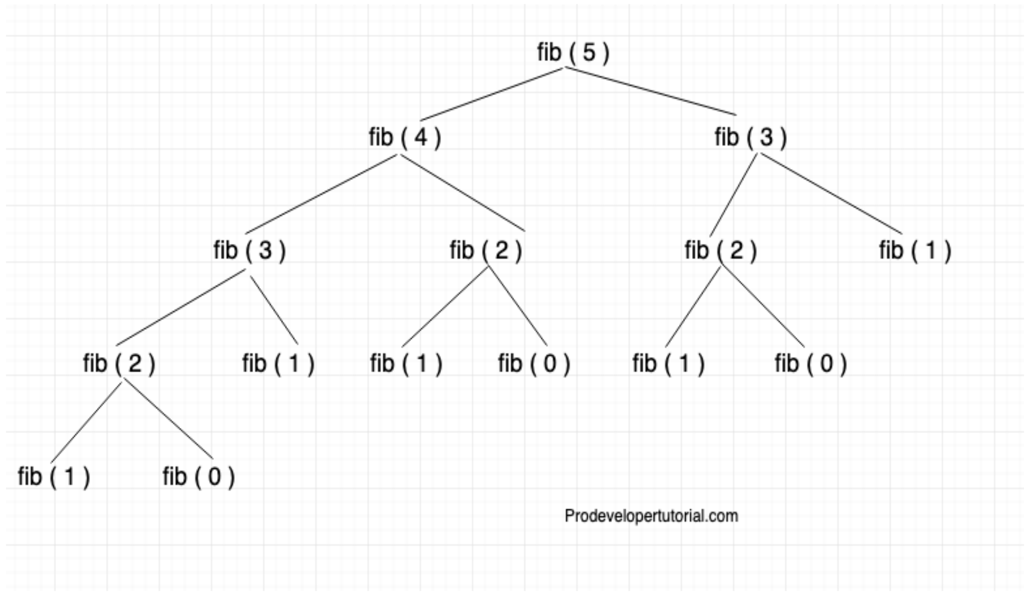
Lets take the example of Fib(5) as shown below:

The recurrence relation of fib(n)= 2T(n-1)+1 => O($2^n$)

Now this time complexity is reduced by DP.

O($2^n$) $\longrightarrow$ O(n)

In DP as Top-down Approach(Memoization

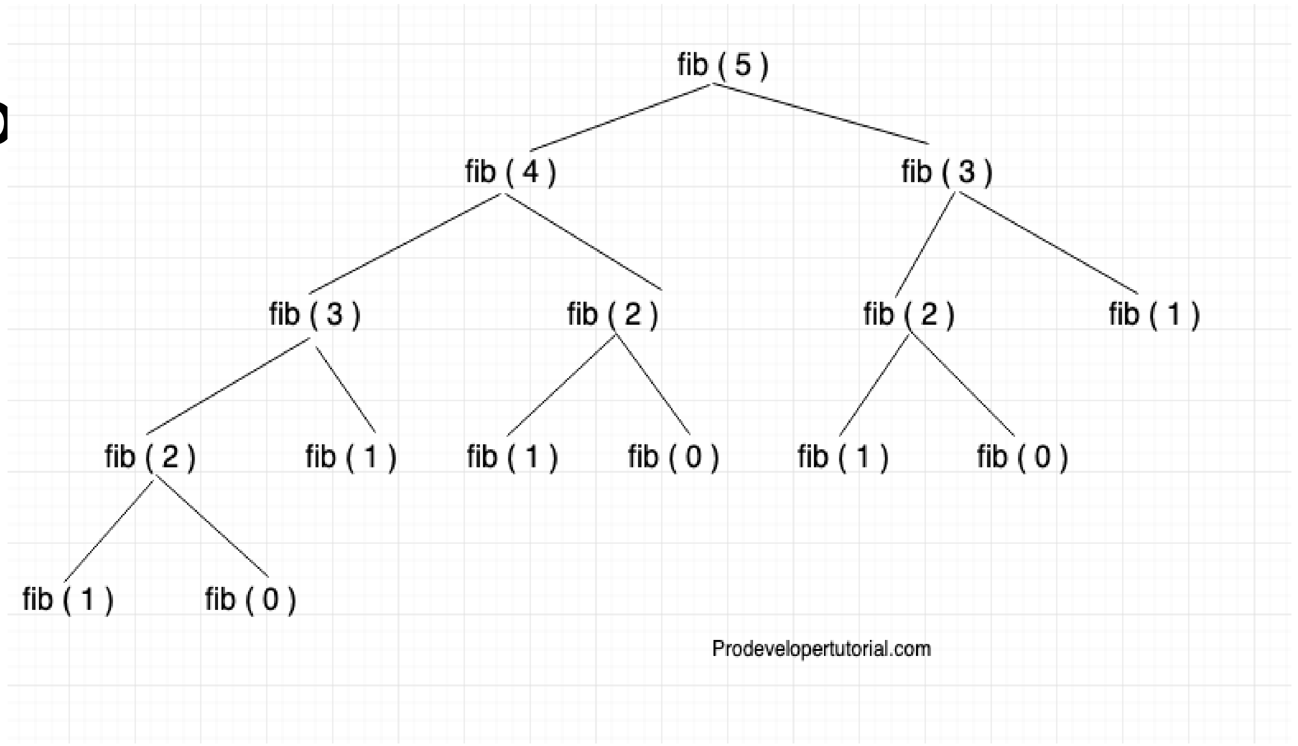


# Fibonacci sequence/Series by DP

**DP as Top-down Approach(Memoization)**

**fib(n) ----> n+1 calls required**

**Time complexity O(n) using D**



## Fibonacci sequence/Series by DP

- **DP as Bottom-up Approach(Tabulation Method)**
- **In this approach we uses iteration method**

```
int Fib (int n)
{
 If (n ≤1)
return n;
F[0]=0; F[1]=1;
For(int i=2;i<=n;i++)
{F[i]= F[i-2]+F[i-1]; }
Return F[n];
}
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |

## 0/1 Knapsack Problem by Dynamic Programming

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.

Consider-

Knapsack weight capacity = w

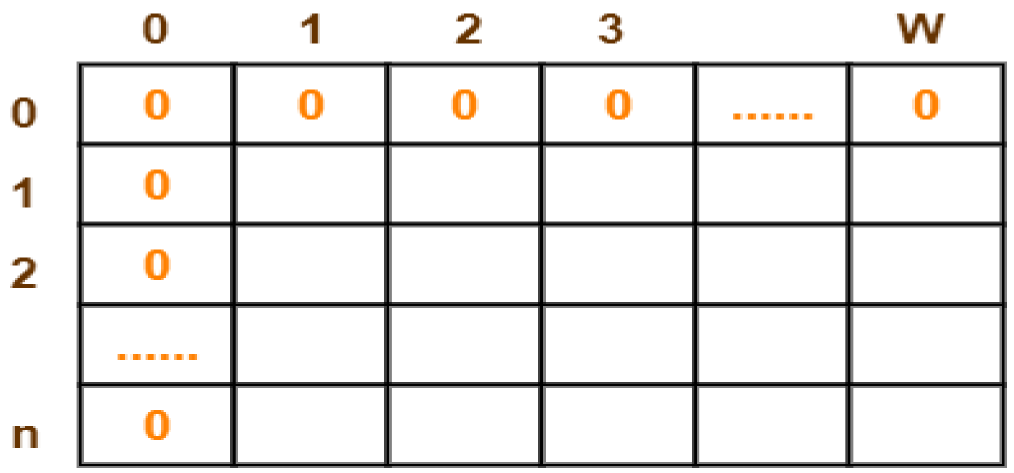
Number of items each having some weight and value = n

0/1 knapsack problem is solved using dynamic programming in the following steps-

## 0/1 Knapsack Problem Using Dynamic Programming

**Step-01:**

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



| | 0 | 1 | 2 | 3 | | W |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | …… | 0 |
| 1 | 0 | | | | | |
| 2 | 0 | | | | | |
| | …… | | | | | |
| n | 0 | | | | | |

T-Table

## 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-

$$T(i, j) = \max \{ T(i-1, j), \text{value}_i + T(i-1, j - \text{weight}_i) \}$$

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.

## PRACTICE PROBLEM BASED ON 0/1 KNAPSACK PROBLEM

Problem- For the given set of items and knapsack capacity = 8 kg, find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach.

| ITEM | WEIGHT | VALUE |
|---|---|---|
| 1 | 2 | 1 |
| 2 | 3 | 2 |
| 3 | 4 | 5 |
| 4 | 5 | 6 |

**OR**

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

Consider- n = 4 w = 8 kg

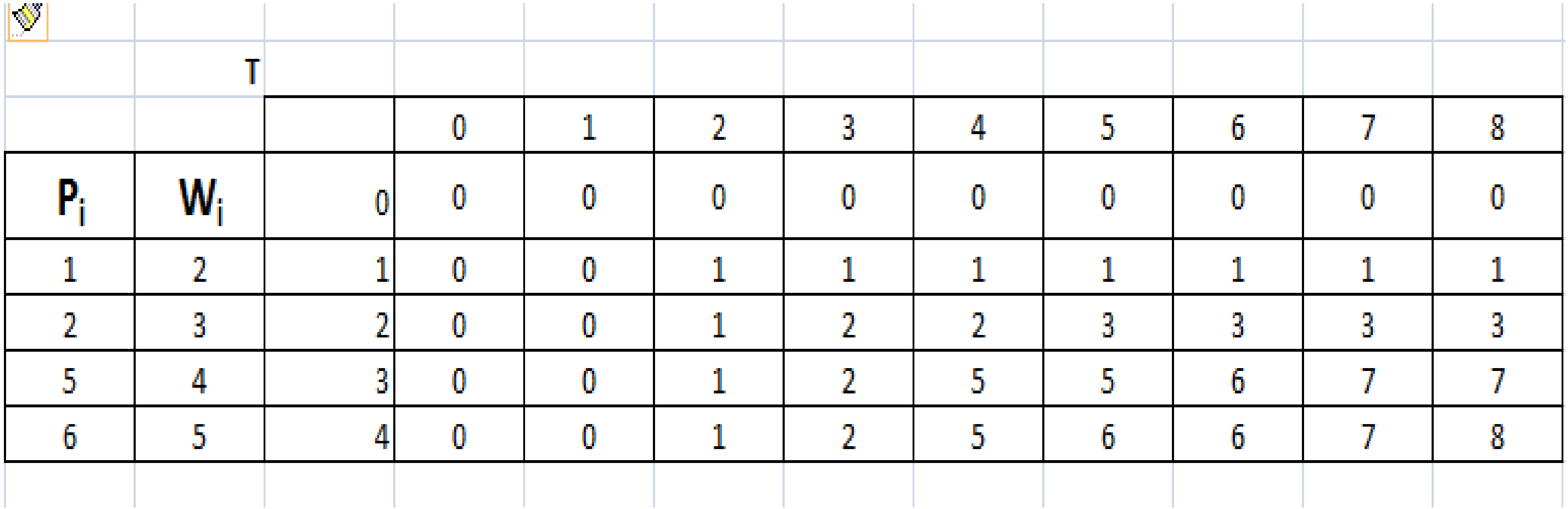
(w1, w2, w3, w4) = (2, 3, 4, 5)

(b1, b2, b3, b4) = (1, 2, 5, 6)

## PRACTICE PROBLEM BASED ON 0/1 KNAPSACK PROBLEM

So in the Problem maximum profit is 8 and we have to take the objects like

X1= 0,  X2= 1,  X3= 0,  X4= 1



T

| | | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $P_i$ | $W_i$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 2 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 3 | 2 | 0 | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 3 |
| 5 | 4 | 3 | 0 | 0 | 1 | 2 | 5 | 5 | 6 | 7 | 7 |
| 6 | 5 | 4 | 0 | 0 | 1 | 2 | 5 | 6 | 6 | 7 | 8 |

## Example of 0/1 Knapsack Problem:

Problem: The maximum weight the knapsack can hold is W is 8. There are four items to choose from. Their weights and values are presented in the following table:

| ITEM | WEIGHT | VALUE/PROFIT |
|---|---|---|
| 1 | 3 | 2 |
| 2 | 4 | 3 |
| 3 | 6 | 1 |
| 4 | 5 | 4 |

## Solution of 0/1 Knapsack Problem:

Solution: $T(i, j) = \max \{ T(i-1, j), \text{value}_i + T(i-1, j - \text{weight}_i) \}$

$T(4,6) = \max\{(T[3][6]), 1+T[3][6-6])\}$

$= \max \{4, 1+0\} \Rightarrow 4$

$X_i = \{1,0,0,1\}$

J →

| | | **T** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $P_i$ | $W_i$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 3 | 1 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 |
| 3 | 4 | 2 | 0 | 0 | 0 | 2 | 3 | 3 | 3 | 5 | 5 |
| 4 | 5 | 3 | 0 | 0 | 0 | 2 | 3 | 4 | 4 | 5 | 6 |
| 1 | 6 | 4 | 0 | 0 | 0 | 2 | 3 | 4 | 4 | 5 | 6 |

I ↓

← Max profit

## Computing a Binomial Coefficient by Dynamic Programming

Following are common definition of Binomial Coefficients.

1. A binomial coefficient C(n, k) can be defined as the coefficient of $X^k$ in the expansion of $(1 + X)^n$.
2. A binomial coefficient C(n, k) also gives the number of ways, disregarding order, that k objects can be chosen from among n objects; more formally, the number of k-element subsets (or k-combinations) of an n-element set.

Binomial coefficients are coefficients of the binomial formula:

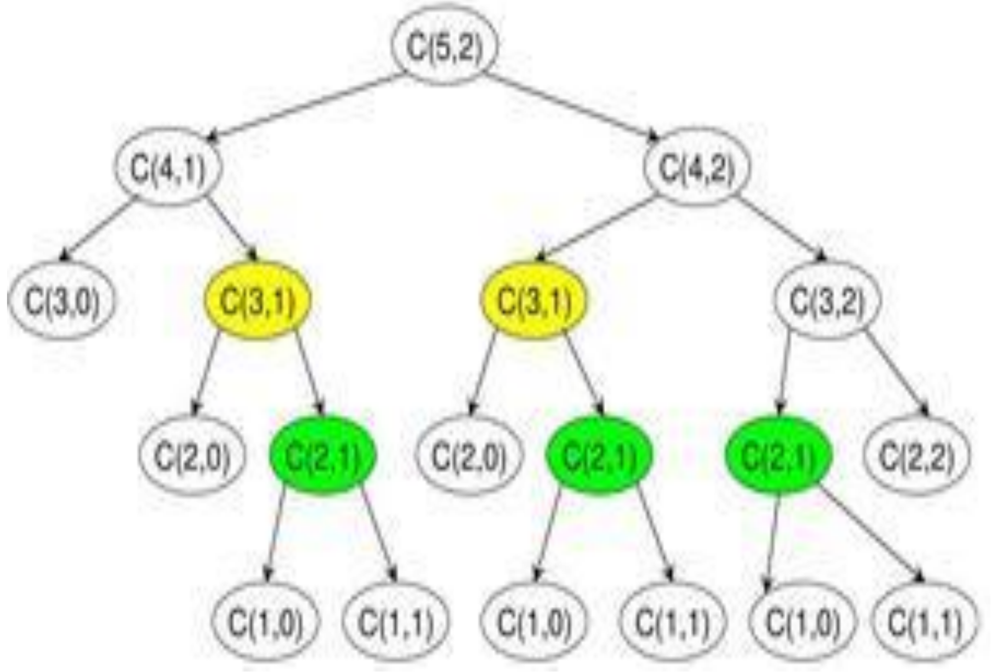
**Binomial coefficients are coefficients of the binomial formula:**

$$(a + b)^n = C(n,0)a^n b^0 + \ldots + C(n,k)a^{n-k}b^k + \ldots + C(n,n)a^0 b^n$$

$$\binom{n}{k} = \begin{cases} 1 & \text{, if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{, if } 0 < k < n \\ 0 & \text{, otherwise} \end{cases}$$

## Computing a Binomial Coefficient by DP

```
int binomialCoeff(int n, int k)
{   if (k > n)
      return 0;
    if (k == 0 || k == n)
        return 1;
return binomialCoeff  (n - 1, k – 1 ) + binomialCoeff(n - 1, k);
}
```



Since the same sub-problems are called again, this problem has Overlapping Sub problems property. So the Binomial Coefficient  problem has  dynamic programming problem. Like other typical Dynamic Programming(DP). problems, re-computations of the same sub-problems can be avoided by constructing a temporary 2D-array C[][] in a bottom-up manner. Following is Dynamic Programming based implementation.

## Computing a Binomial Coefficient by Dynamic Programming

```
int binomialCoeff(int n, int k)
{
    int C[n + 1][k + 1];
    int i, j;
    for (i = 0; i <= n; i++) {
        for (j = 0; j <= min(i, k); j++) {
            if (j == 0 || j == i)
                C[i][j] = 1;
            else
                C[i][j] = C[i - 1][j - 1] + C[i - 1][j];
        }
    }
    return C[n][k];
}
```

Time Complexity : O(n*k)

### Pascal’s Triangle

j

i

| n/k | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | | | | | | | |
| 1 | 1 | 1 | | | | | | |
| 2 | 1 | 2 | 1 | | | | | |
| 3 | 1 | 3 | 3 | 1 | | | | |
| 4 | 1 | 4 | 6 | 4 | 1 | | | |
| 5 | 1 | 5 | 10 | 10 | 5 | 1 | | |
| 6 | 1 | 6 | 15 | 20 | 15 | 6 | 1 | |
| 7 | 1 | 7 | 21 | 35 | 35 | 21 | 7 | 1 |

## Making Coin Change problem by Dynamic Programming

**Limitation of greedy algorithm:**

- Works only in limited number of instances
- Ex: Given the set of coins= {1, 4, 6}, generate the change of Rs. 8
- Greedy algorithms gives: 6+1+1 = 3 coins
- But optimum solution is: 4+4 = 2 coins

**Solution using dynamic programming:** We want to generate the change of "N" Rs.

- Coins of n different denominations /values are represented as: di , 1 ≤ i ≤ n
- Setup a table: C[1..n, 0..N]

**ROW**: for each available denomination.

**COLUMN** : for each amount from 0 to N units

## Making Coin Change problem by Dynamic Programming

Algorithm

```
for( i=o ; i<= coinslength; i++)
    {
        for( j=o; j<= w; j++)
        {
            if(coins[i]> j)
                {  c[i][j]= c[i-1][j] }
Else {
C[i][j] = min(a[i-1][j],  1+c[i][j-coins[i]])
}
```

Time complexity = O(n*w)

# Making Coin Change problem by Dynamic Programming

W= 10

Coins ={ 1,5,6,9}

j / w

i/coin

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 5 | 0 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2 |
| 6 | 0 | 1 | 2 | 3 | 4 | 1 | 1 | 2 | 3 | 4 | 2 |
| 9 | 0 | 1 | 2 | 3 | 4 | 1 | 1 | 2 | 3 | 1 | 2 |

If coin value > w then just copy the above value.

C[i][j] = min(a[i-1][j],  1+c[i][j-coins[i])

## Dynamic Programming - Matrix-chain Multiplication

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications. We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

(ABC)D = (AB)(CD) = A(BCD) = ....

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a $10 \times 30$ matrix, B is a $30 \times 5$ matrix, and C is a $5 \times 60$ matrix. Then,

$(AB)C = (10\times30\times5) + (10\times5\times60) = 1500 + 3000 = 4500$ operations

$A(BC) = (30\times5\times60) + (10\times30\times60) = 9000 + 18000 = 27000$ operations.

# Matrix Chain Multiplication

- Suppose we have a sequence or chain A1, A2, …, An of n matrices to be multiplied
- That is, we want to compute the product A1A2…An
- There are many possible ways (parenthesizations) to compute the product
- Example: consider the chain A1, A2, A3, A4 of 4 matrices
- Let us compute the product A1A2A3A4
- There are 5 possible ways:

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

## Algorithm to Multiply 2 Matrices

Input: Matrices $A_{p\times q}$ and $B_{q\times}r$ (with dimensions $p\times q$ and $q\times r$)

Result: Matrix $C_{p\times r}$ resulting from the product $A\cdot B$

```
MATRIX-MULTIPLY($A_{p\times q}$ , $B_{q\times r}$)
1. for $i \leftarrow 1$ to $p$
2.     for $j \leftarrow 1$ to $r$
3.         $C[i, j] \leftarrow 0$
4.         for $k \leftarrow 1$ to $q$
5.             $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
6. return C
```

# Matrix Chain Multiplication Algorithm

Let $A_{i,j}$ be the result of multiplying matrices i through j. It can be seen that the dimension of $A_{i,j}$ is $p_{i-1}$ x $p_j$ matrix.

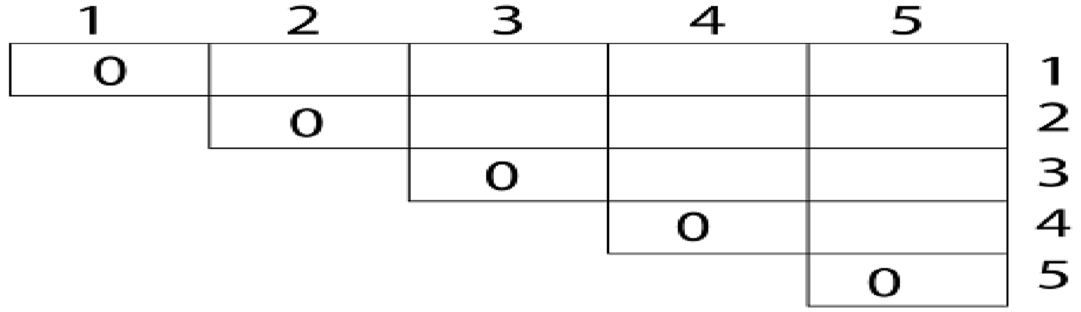
Dynamic Programming solution involves breaking up the problems into sub-problems whose solution can be combined to solve the global problem.

the minimum cost of parenthesizing the product $A_i\ A_{i+1} \ldots\ldots A_j$ becomes

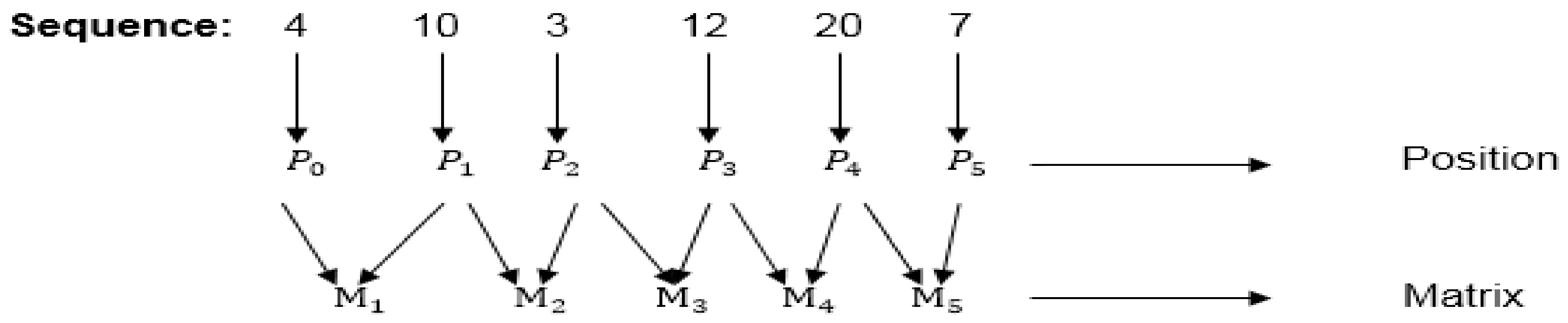
$$m\,[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\{m\,[i,k] + m\,[k+1,j] + p_{i-1}\,p_k p_j\} & \text{if } i < j \\ i \le k < j & \end{cases}$$

## Example of  Matrix Chain Multiplication

- Lets  take the 5 matrices $A_1$, $A_2$, $A_3$, $A_4$ $A_5$with the dimensions of 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.



| 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|
| 0 | | | | | 1 |
| | 0 | | | | 2 |
| | | 0 | | | 3 |
| | | | 0 | | 4 |
| | | | | 0 | 5 |



## Calculation of Product of 2 matrices:

m (1,2) = $m_1$ x $m_2$

= 4 x 10 x 10 x 3

= 4 x 10 x 3 = 120

m (2, 3) = $m_2$ x m

= 10 x 3 x 3 x 12

= 10 x 3 x 12 = 360

m (3, 4) = $m_3$ x $m_4$

= 3 x 12 x 12 x 20

= 3 x 12 x 20 = 720

m (4,5) = $m_4$ x $m_5$

= 12 x 20 x 20 x 7

= 12 x 20 x 7 = 1680

COLUMNS

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

ROWS

Now product of 3 matrices:

## M [1, 3] = $M_1$ $M_2$ $M_3$

1. There are two cases by which we can solve this multiplication: ( $M_1$ x $M_2$) + $M_3$, $M_1$+ ($M_2$x $M_3$)
2. After solving both cases we choose the case in which minimum output is there.

$$M\,[1, 3] = \min \begin{cases} M\,[1,2] + M\,[3,3] + p_0\,p_2p_3 = 120 + 0 + 4.3.12 \;=\; 264 \\ M\,[1,1] + M\,[2,3] + p_0\,p_1p_3 = 0 + 360 + 4.10.12 \;=\; 840 \end{cases}$$

As Comparing both output **264** is minimum in both cases so we insert **264** in table and ( $M_1$ x $M_2$) + $M_3$ this combination is chosen for the output making.

## M [2, 4] = $M_2$ $M_3$ $M_4$

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.

$$M\,[2, 4] = \min \begin{cases} M[2,3] + M[4,4] + \; p_1p_3p_4 = 360 + 0 + 10.12.20 = 2760 \\ M[2,2] + \; M[3,4] + \; p_1p_2p_4 = 0 + 720 + 10.3.20 = \; 1320 \end{cases}$$

As Comparing both output **1320** is minimum in both cases so we insert **1320** in table and $M_2$+($M_3$ x $M_4$) this combination is chosen for the output making.

Now product of 3 matrices:

**M [3, 5] = $M_3$ $M_4$ $M_5$**

1. There are two cases by which we can solve this multiplication: ( $M_3$ x $M_4$) + $M_5$, $M_3$+ ( $M_4$x$M_5$)
2. After solving both cases we choose the case in which minimum output is there.

$$M\,[3,5] = \min \begin{cases} M[3,4] + M[5,5] + p_2p_4p_5 = 720 + 0 + 3\,.20.7 \;=\; 1140 \\ M[3,3] + M\,[4,5] + p_2p_3p_5 = 0 + 1680 + 3.12.7 = 1932 \end{cases}$$

As Comparing both output 1140 is minimum in both cases so we insert 1140 in table and ( M3 x M4) + M5this 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 |

$\longrightarrow$

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

## Now Product of 4 matrices:

M [1, 4] = $M_1$ $M_2$ $M_3$ $M_4$

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

1. ( $M_1$ x $M_2$ x $M_3$) $M_4$
2. $M_1$ x($M_2$ x $M_3$ x $M_4$)
3. ($M_1$ x$M_2$) x ( $M_3$ x $M_4$)

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 ($M_1$ x$M_2$) x ($M_3$ x $M_4$) combination is taken out in output making,

# Matrix Chain Multiplication

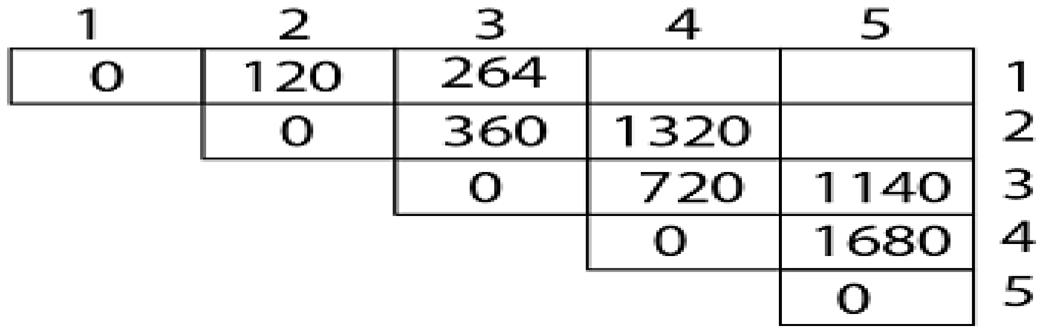
**M [2, 5] = $M_2$ $M_3$ $M_4$ $M_5$**

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

1. $(M_2 \times M_3 \times M_4) \times M_5$
2. $M_2 \times (M_3 \times M_4 \times M_5)$
3. $(M_2 \times M_3) \times (M_4 \times M_5)$

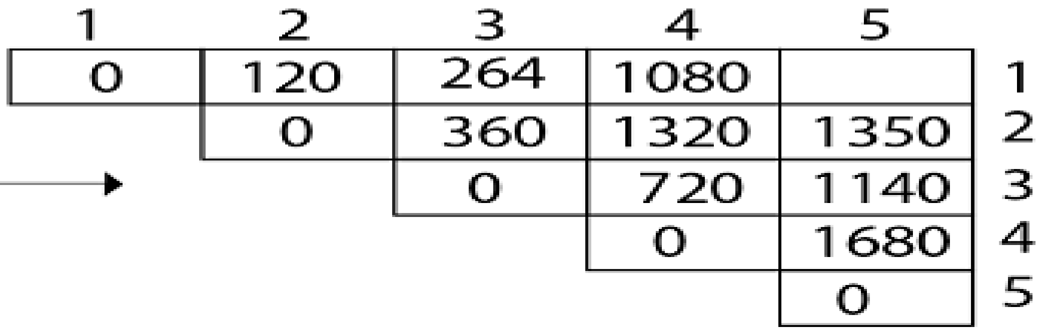
After solving these cases we choose the case in which minimum output is there

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



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

$\longrightarrow$



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

# Now Product of 5 matrices:

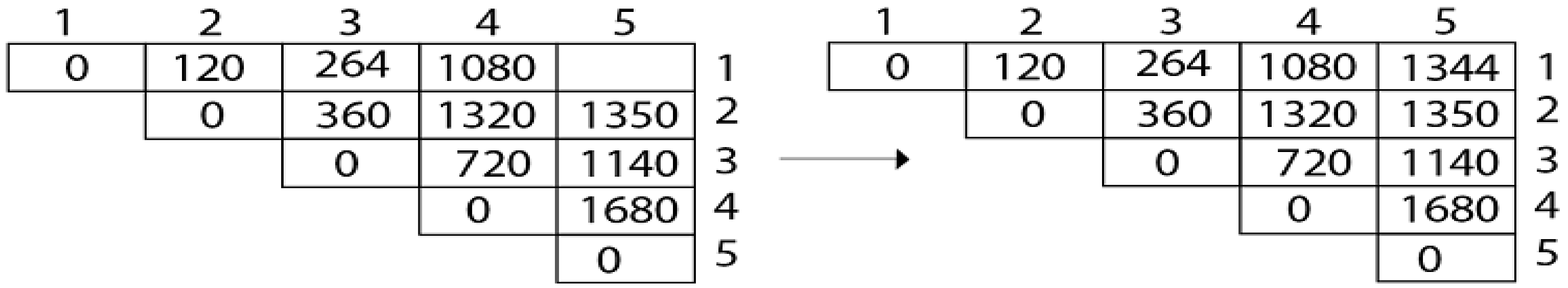
**M [1, 5] = $M_1$ $M_2$ $M_3$ $M_4$ $M_5$**

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

1. $(M_1 \times M_2 \times M_3 \times M_4) \times M_5$
2. $M_1 \times (M_2 \times M_3 \times M_4 \times M_5)$
3. $(M_1 \times M_2 \times M_3) \times M_4 \times M_5$
4. $M_1 \times M_2 \times (M_3 \times M_4 \times M_5)$

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

$$M[1,5] = \min \begin{Bmatrix} M[1,4] + M[5,5] + p_0p_4p_5 = 1080 + 0 + 4.20.7 = 1544 \\ M[1,3] + M[4,5] + p_0p_3p_5 = 264 + 1680 + 4.12.7 = 2016 \\ M[1,2] + M[3,5] + p_0p_2p_5 = 120 + 1140 + 4.3.7 = 1344 \\ M[1,1] + M[2,5] + p_0p_1p_5 = 0 + 1350 + 4.10.7 = 1630 \end{Bmatrix}$$



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

$\longrightarrow$

| 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 |

Thank
You