Design and Analysis of Algorithm

UNIT – 3 : Dynamic Programming

Dynamic Programming:

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again.

The subproblems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem.

The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler sub-problems, solving each sub-problem just once, and then storing their solutions to avoid repetitive computations.

Let's understand this approach through an example.

Example: Fibonacci series.

The following series is the Fibonacci series:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

The numbers in the above series are not randomly calculated. Mathematically, we could write each of the terms using the below formula:

F(n) = F(n-1) + F(n-2),

With the base values F(0) = 0, and F(1) = 1.

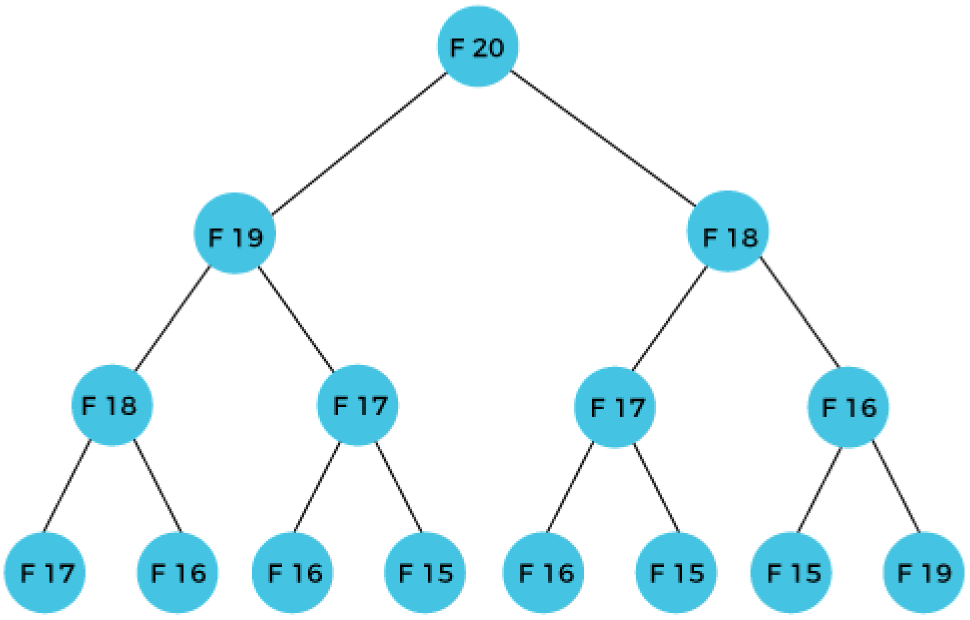
To calculate the other numbers, we follow the above relationship.

For example, F(2) is the sum **f(0)** and **f(1),** which is equal to 1.

How can we calculate F(20)?

The F(20) term will be calculated using the nth formula of the Fibonacci series.

The below figure shows that how F(20) is calculated.



As we can observe in the above figure that F(20) is calculated as the sum of F(19) and F(18).

In the dynamic programming approach, we try to divide the problem into the similar sub-problems.

We are following this approach in the above case where F(20) into the similar subproblems, i.e., F(19) and F(18). If we recap the definition of dynamic programming that it says the similar sub-problem should not be computed more than once. Still, in the above case, the sub-problem is calculated twice.

In the above example, F(18) is calculated two times; similarly, F(17) is also calculated twice. However, this technique is quite useful as it solves the similar subproblems, but we need to be cautious while storing the results because we are not particular about storing the result that we have computed once, then it can lead to a wastage of resources.

In the above example, if we calculate the F(18) in the right subtree, then it leads to the tremendous usage of resources and decreases the overall performance.

The solution to the above problem is to save the computed results in an array.

First, we calculate F(16) and F(17) and save their values in an array. The F(18) is calculated by summing the values of F(17) and F(16), which are already saved in an array. The computed value of F(18) is saved in an array. The value of F(19) is calculated using the sum of F(18), and F(17), and their values are already saved in an array. The computed value of F(19) is stored in an array. The value of F(20) can be calculated by adding the values of F(19) and F(18), and the values of both F(19) and F(18) are stored in an array. The final computed value of F(20) is stored in an array.

How does the dynamic programming approach work?

The following are the steps that the dynamic programming follows:

- It breaks down the complex problem into simpler subproblems.
- It finds the optimal solution to these sub-problems.
- It stores the results of sub-problems (memorization). The process of storing the results of subproblems is known as memorization.
- It reuses them so that same sub-problem is calculated more than once.
- Finally, calculate the result of the complex problem.

Approaches of dynamic programming:

There are two approaches to dynamic programming:

- **Top-down approach**
- **Bottom-up approach**

Top-down approach:

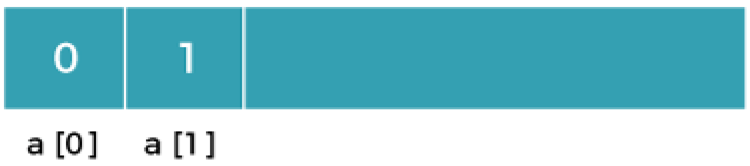
The top-down approach follows the memorization technique, while bottom-up approach follows the tabulation method. Here memorization is equal to the sum of recursion and caching. Recursion means calling the function itself, while caching means storing the intermediate results.

Bottom-Up approach:

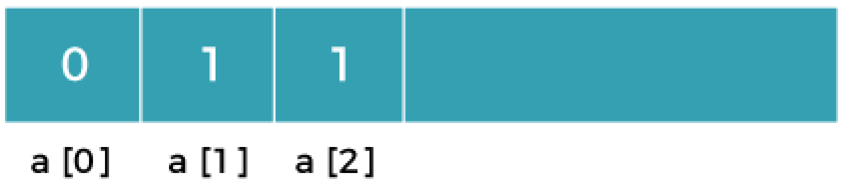
The bottom-up approach is also one of the techniques which can be used to implement the dynamic programming. It uses the tabulation technique to implement the dynamic programming approach. It solves the same kind of problems but it removes the recursion. If we remove the recursion, there is no stack overflow issue and no overhead of the recursive functions. In this tabulation technique, we solve the problems and store the results in a matrix.

Let's understand through an example.

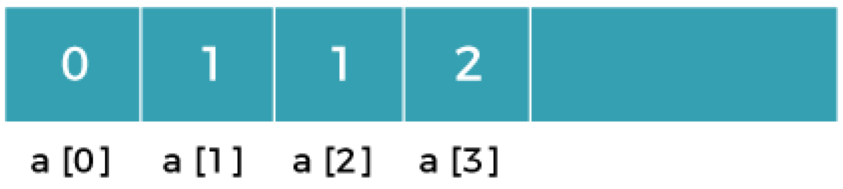
Suppose we have an array that has 0 and 1 values at a[0] and a[1] positions, respectively shown as below:



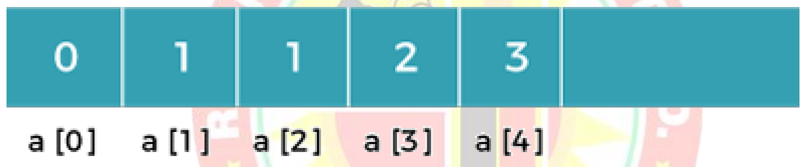
Since the bottom-up approach starts from the lower values, so the values at a[0] and a[1] are added to find the value of a[2] shown as below:



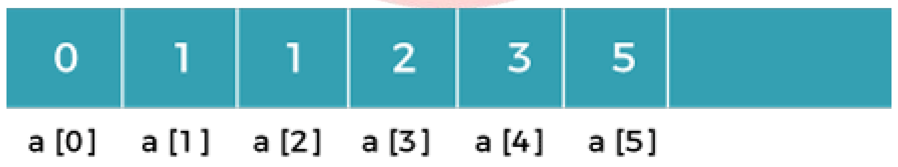
The value of a[3] will be calculated by adding a[1] and a[2], and it becomes 2 shown as below:



The value of a[4] will be calculated by adding a[2] and a[3], and it becomes 3 shown as below:



The value of a[5] will be calculated by adding the values of a[4] and a[3], and it becomes 5 shown as below:



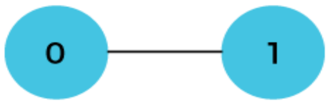
The code for implementing the Fibonacci series using the bottom-up approach is given below:

```
int fib(int n)
{
        int A[];
        A[0] = 0, A[1] = 1;
        for( i=2; i<=n; i++)
        {
                A[i] = A[i-1] + A[i-2]
        }
        return A[n];
}
```

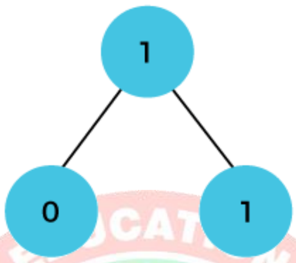
In the above code, base cases are 0 and 1 and then we have used for loop to find other values of Fibonacci series.

Let's understand through the diagrammatic representation.

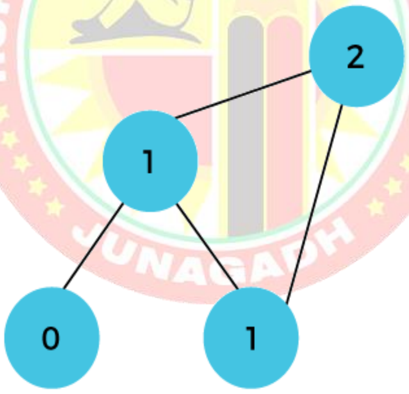
Initially, the first two values, i.e., 0 and 1 can be represented as:



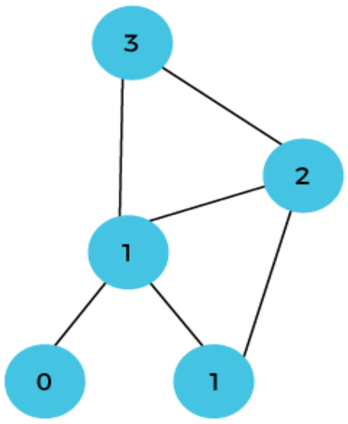
When i=2 then the values 0 and 1 are added shown as below:



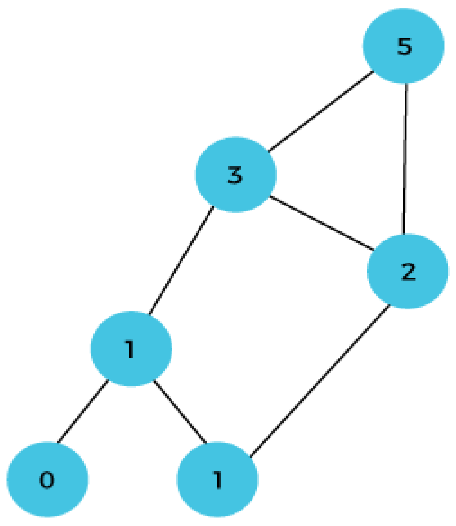
When i=3 then the values 1and 1 are added shown as below:



When i=4 then the values 2 and 1 are added shown as below:



When i=5, then the values 3 and 2 are added shown as below:



In the above case, we are starting from the bottom and reaching to the top.

What is Binomial Coefficient?

Before knowing how to find binomial coefficient. Let's discuss briefly **what is Binomial Coefficient? and why is it even required?**

Because Binomial Coefficient is used heavily to solve combinatory problems.

Let's say you have some **n** different elements and you need to pick **k** elements. So, if you want to solve this problem you can easily write all the cases of choosing k elements out of n elements. But this is a very time-consuming process when n increases.

This problem can be easily solved using binomial coefficient. More than that, this problem of choosing k elements out of n different elements is one of the way to define binomial coefficient **n C k.** Binomial coefficient can be easily calculated using the given formula:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Since now we are good at the basics, we should find ways to calculate this efficiently.

Naive Approach for finding Binomial Coefficient:

This approach isn't **too naive at all.** Consider you are asked to find the number of ways of choosing 3 elements out of 5 elements. So you can easily find **n!, k! and (n-k)!** and put the values in the given formula.

This solution takes only **O(N) time** and **O(1) space**. But sometimes your factorial values may overflow so we need to take care of that. This approach is fine if we want to calculate a single binomial coefficient. But many times we need to calculate many binomial coefficients. So, it's better to have them precomputed. We will find out how to find the binomial coefficients efficiently.

Optimized Approach for finding Binomial Coefficient:

Well, naive approach was not naive if we wanted to find a single binomial coefficient. But when we need to find many binomial coefficients. So the problem becomes difficult to complete in time limit. Because naive approach is still time consuming. So, here we have some queries where we are asked to calculate **nCk** for given n and k. There may be many queries.

To solve this, we should be familiar with **Pascal's Triangle**. Cause that will make us understand much clearly why are we going to do what we are going to do.

```
                        1
                     1     1
                  1     2     1
               1     3     3     1
            1     4     6     4     1
         1     5    10    10     5     1
      1     6    15    20    15     6     1
   1     7    21    35    35    21     7     1
1     8    28    56    70    56    28     8     1
```

Any cell in Pascal's triangle denotes binomial coefficients. We need to know some things regarding the Pascal's triangle.

1. It starts with row 0.
2. Any number in Pascal's triangle denotes binomial coefficient.
3. Any binomial coefficient which is not on the boundaries of the row is made from the summation of elements that are just above it in left and right direction.

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{for all integers } n, k : 1 \le k \le n-1,$$

Java Code to find Binomial Coefficient:

```
import java.util.*;
class Main
{
        static int C [ ][ ];
        static void precomputeBinomialCoefficients()
        {
                for (int i = 0; i <= 50; i++)
                {
                        for (int j = 0; j <= i; j++)
                        {
                                // Base Cases
                                if (j == 0 || j == i)
                                        C[i][j] = 1;
                                else
                                        C[i][j] = C[i - 1][j - 1] + C[i - 1][j];
                        }
                }
        }
        public static void main(String[] args)
        {
                C = new int[51][51];
                precomputeBinomialCoefficients();
                Scanner sc = new Scanner(System.in);
                int noOfQueries;
                noOfQueries = sc.nextInt();
                while(noOfQueries-- > 0)
                {
                        int n = sc.nextInt();
                        int k = sc.nextInt();
                        if(n<=50 && k<=50)
                                System.out.println(C[n][k]);
                        else
                                System.out.println(0);
                }
        }
}
```

Coin Change Problem:

You are given a sequence of coins of various denominations as part of the coin change problem.

For example, consider the following array a collection of coins, with each element representing a different denomination.

{2, 3, 5, 10, 20, 30, 50};

Our goal is to use these coins to accumulate a certain amount of money while using the fewest (or optimal) coins.

Furthermore, you can assume that a given denomination has an infinite number of coins. To put it another way, you can use a specific denomination as many times as you want.

For example, if you want to reach 78 using the above denominations, you will need the four coins listed below.

{3, 5, 20, 50};

Consider the following another set of denominations:

{2, 3, 5, 6};

If you want to make a total of 9, you only need two coins in these denominations, as shown below:

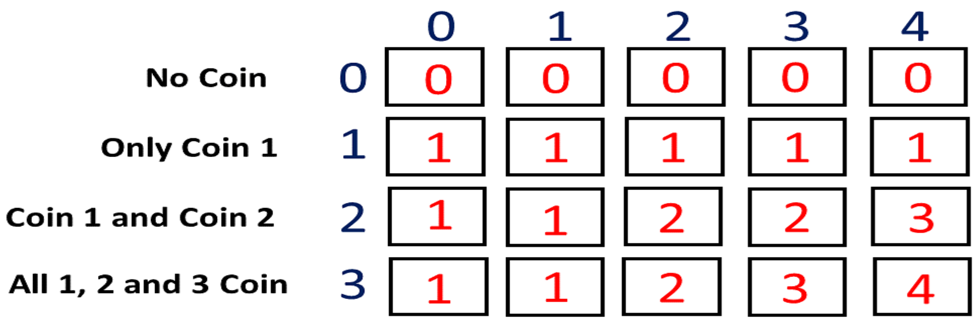
{3, 6};

However, if you recall the greedy algorithm approach, you end up with three coins for the above denominations (5, 2, 2). This is due to the greedy algorithm's preference for local optimization.

After understanding a coin change problem, you will look at the pseudocode of the coin change problem in this tutorial.

Coin Change Problem Solution Using Dynamic Programming:

To store the solution to the subproblems, you must use a 2D array (i.e. table). Then, take a look at the image below.



		0	1	2	3	4
No Coin	0	0	0	0	0	0
Only Coin 1	1	1	1	1	1	1
Coin 1 and Coin 2	2	1	1	2	2	3
All 1, 2 and 3 Coin	3	1	1	2	3	4

- The size of the dynamic progTable is equal to (number of coins +1)*(Sum +1).
- The first column value is one because there is only one way to change if the total amount is 0. (we do not include any coin).
- **Row**: The total number of coins. The fact that the first-row index is 0 indicates that no coin is available. If the value index in the second row is 1, only the first coin is available. Similarly, if the value index in the third row is 2, it means that the first two coins are available to add to the total amount, and so on. The row index represents the index of the coin in the coin's array, not the coin value.
- **Column**: Total amount (sum). Because the first-column index is 0, the sum value is 0. The second column index is 1, so the sum of the coins should be 1. Similarly, the third column value is 2, so a change of 2 is required, and so on.

As a result, each table field stores the solution to a sub problem.

For example, dynamic progTable [2][3]=2 indicates two ways to compute the sum of three using the first two coins 1,2.

The final outcome will be calculated by the values in the last column and row.

In this case, you must loop through all of the indexes in the memo table (except the first row and column) and use previously-stored solutions to the subproblems.

- If the coin value is greater than the dynamic progSum, the coin is ignored, i.e. dynamicprogTable[i][j]=dynamicprogTable[i-1][j].
- If the coin value is less than the dynamic progSum, you can consider it, i.e. dynamicprogTable[i][j]=dynamicprogTable[i1].[dynamicprogSum]+dynamicprogTable[i][j-coins[i-1]].

You will look at the complexity of the coin change problem after figuring out how to solve it.

Code Implementation of Coin Change Problem:

```
#include <stdio.h>
int coins [] = {1,2,3};
int numberofCoins = 3, sum = 4;
int solution (int sol, int i)
{
        if (numberofCoins == 0 || sol > sum || i>=numberofCoins)
        {
                return 0;
        }
        else if (sol == sum)
        {
                return 1;
        }
        return solution (sol + coins[i], i) + solution (sol, i+1);
}
int main ()
{
        printf ("Total solutions: %d", solution (0,0));
        return 0;
}
```

Assembly Line Scheduling:

Assembly line scheduling is a manufacturing problem. In automobile industries assembly lines are used to transfer parts from one station to another station.

Manufacturing of large items like car, trucks etc. generally undergoes through multiple stations, where each station is responsible for assembling particular part only. Entire product be ready after it goes through predefined **n** stations in sequence.

Manufacturing of car may be done through several stages like engine fitting, colouring, light fitting, fixing of controlling system, gates, seats and many other things.

The particular task is carried out at the station dedicated to that task only. Based on the requirement there may be more than one assembly line.

In case of two assembly lines if the load at station j at assembly 1 is very high, then components are transfer to station of assembly line 2 the converse is also true. This technique helps to speed ups the manufacturing process.

The time to transfer partial product from one station to next station on the same assembly line is negligible. During rush factory may transfer partially completed auto from one assembly line to another, complete the manufacturing as quickly as possible.

Assembly line scheduling is a problem in operations management that involves determining the optimal sequence of tasks or operations on an assembly line to minimize production costs or maximize efficiency. This problem can be solved using various data structures and algorithms. One common approach is dynamic programming, which involves breaking the problem down into smaller sub-problems and solving them recursively.

The following is an overview of the steps involved in solving an assembly line scheduling problem using dynamic programming:

- **Define the problem:** The first step is to define the problem, including the number of tasks or operations involved, the time required to perform each task on each assembly line, and the cost or efficiency associated with each task.

- **Define the sub-problems:** Next, we need to define the sub-problems by breaking down the problem into smaller pieces. In assembly line scheduling, this involves determining the optimal sequence of tasks for each station along the assembly line.

- **Define the recurrence relation:** The recurrence relation defines the relationship between the sub-problems and the overall problem. In assembly line scheduling, the recurrence relation involves computing the minimum cost or maximum efficiency of the assembly line by considering the cost or efficiency of the previous station and the time required to transition to the next station.

- **Solve the sub-problems:** To solve the sub-problems, we can use a table or matrix to store the minimum cost or maximum efficiency of each station. We can then use this table to determine the optimal sequence of tasks for the entire assembly line.

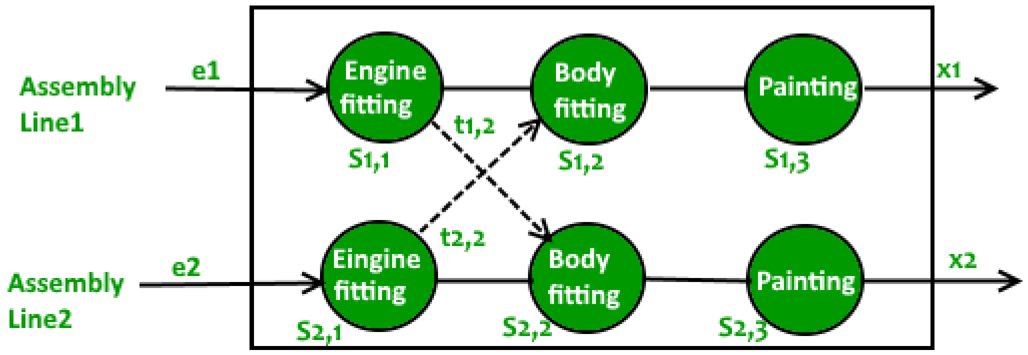
- **Trace the optimal path:** Finally, we can trace the optimal path through the table or matrix to determine the sequence of tasks that minimizes production costs or maximizes efficiency.

A car factory has two assembly lines, each with n stations. A station is denoted by $S_{i,j}$ where i is either 1 or 2 and indicates the assembly line the station is on, and j indicates the number of the station.

The time taken per station is denoted by $a_{i,j}$. Each station is dedicated to some sort of work like engine fitting, body fitting, painting, and so on. So, a car chassis must pass through each of the n stations in order before exiting the factory. The parallel stations of the two assembly lines perform the same task. After it passes through station $S_{i,j}$, it will continue to station $S_{i,j+1}$ unless it decides to transfer to the other line. Continuing on the same line incurs no extra cost, but transferring from line i at station j – 1 to station j on the other line takes time $t_{i,j}$. Each assembly line takes an entry time e_i and exit time x_i which may be different for the two lines.

Give an algorithm for computing the minimum time it will take to build a car chassis.

The below figure presents the problem in a clear picture:



The following information can be extracted from the problem statement to make it simpler:

- Two assembly lines, 1 and 2, each with stations from 1 to n.
- A car chassis must pass through all stations from 1 to n in order(in any of the two assembly lines). i.e. it cannot jump from station i to station j if they are not at one move distance.
- The car chassis can move one station forward in the same line, or one station diagonally in the other line. It incurs an extra cost ti, j to move to station j from line i. No cost is incurred for movement in same line.
- The time taken in station j on line i is $a_{i,j}$.
- $S_{i,j}$ represents a station j on line i.

Example:

```
#include <iostream>
#include <vector>
using namespace std;
int fun(vector<vector<int> > a, vector<vector<int> > t,
        int cl, int cs, int x1, int x2, int n)
{
    // base case
    if (cs == n - 1) {
        if (cl == 0) { // exiting from (current) line =0
            return x1;
        }
        else // exiting from line 2
            return x2;
    }
    // continue on same line
    int same
        = fun(a, t, cl, cs + 1, x1, x2, n) + a[cl][cs + 1];
    // continue on different line
    int diff = fun(a, t, !cl, cs + 1, x1, x2, n)
               + a[!cl][cs + 1] + t[cl][cs + 1];

    return min(same, diff);
}
int main()
{
    int n = 4; // number of statin
    vector<vector<int> > a
        = { { 4, 5, 3, 2 }, { 2, 10, 1, 4 } };
    vector<vector<int> > t
        = { { 0, 7, 4, 5 }, { 0, 9, 2, 8 } };

    int e1 = 10;
    int e2 = 12;
    int x1 = 18;
    int x2 = 7;
    // entry from 1st line
    int x = fun(a, t, 0, 0, x1, x2, n) + e1 + a[0][0];
    // entry from 2nd line
    int y = fun(a, t, 1, 0, x1, x2, n) + e2 + a[1][0];
    cout << min(x, y) << endl;
}
```

Output: 35

Knapsack problem:

Here knapsack is like a container or a bag. Suppose we have given some items which have some weights or profits. We have to put some items in the knapsack in such a way total value produces a maximum profit.

For example, the weight of the container is 20 kg. We have to select the items in such a way that the sum of the weight of items should be either smaller than or equal to the weight of the container, and the profit should be maximum.

There are two types of knapsack problems:

- **0/1 knapsack problem**
- **Fractional knapsack problem**

We will discuss both the problems one by one. First, we will learn about the 0/1 knapsack problem.

What is the 0/1 knapsack problem?

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item, then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming.

Example of 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
The number of items is 4

The above problem can be solved by using the following method:

x_i = {1, 0, 0, 1}
= {0, 0, 0, 1}
= {0, 1, 0, 1}

The above are the possible combinations. 1 denotes that the item is completely picked and 0 means that no item is picked. Since there are 4 items so possible combinations will be: **2^4 = 16;**

So, there are 16 possible combinations that can be made by using the above problem. Once all the combinations are made, we have to select the combination that provides the maximum profit.

Another approach to solve the problem is dynamic programming approach. In dynamic programming approach, the complicated problem is divided into sub-problems, then we find the solution of a sub-problem and the solution of the sub-problem will be used to find the solution of a complex problem.

How this problem can be solved by using the Dynamic programming approach?

First, we create a matrix shown as below:

	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4									

In the above matrix, columns represent the weight, i.e., 8. The rows represent the profits and weights of items. Here we have not taken the weight 8 directly, problem is divided into sub-problems, i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8. The solution of the sub-problems would be saved in the cells and answer to the problem would be stored in the final cell. First, we write the weights in the ascending order and profits according to their weights shown as below:

$w_i = \{3, 4, 5, 6\}$ $p_i = \{2, 3, 4, 1\}$

The first row and the first column would be 0 as there is no item for w=0

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0								
2	0								
3	0								
4	0								

When i=1, W=1

w_1 = 3; Since we have only one item in the set having weight 3, but the capacity of the knapsack is 1. We cannot fill the item of 3kg in the knapsack of capacity 1 kg so add 0 at M[1][1] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0							
2	0								
3	0								
4	0								

When i = 1, W = 2

w_1 = 3; Since we have only one item in the set having weight 3, but the capacity of the knapsack is 2.

We cannot fill the item of 3kg in the knapsack of capacity 2 kg so add 0 at M[1][2] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0						
2	0								
3	0								
4	0								

When i=1, W=3

w_1 = 3; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is also 3; therefore, we can fill the knapsack with an item of weight equal to 3.

We put profit corresponding to the weight 3, i.e., 2 at M[1][3] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2					
2	0								
3	0								
4	0								

When i=1, W = 4

w_1 = 3; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is 4; therefore, we can fill the knapsack with an item of weight equal to 3. We put profit corresponding to the weight 3, i.e., 2 at M[1][4] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2				
2	0								
3	0								
4	0								

When i=1, W = 5

w_1 = 3; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is 5; therefore, we can fill the knapsack with an item of weight equal to 3. We put profit corresponding to the weight 3, i.e., 2 at M[1][5] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2			
2	0								
3	0								
4	0								

When i =1, W=6

w_1 = 3; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is 6; therefore, we can fill the knapsack with an item of weight equal to 3.

We put profit corresponding to the weight 3, i.e., 2 at M[1][6] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2		
2	0								
3	0								
4	0								

When i=1, W = 7

w_1 = 3; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is 7; therefore, we can fill the knapsack with an item of weight equal to 3.

We put profit corresponding to the weight 3, i.e., 2 at M[1][7] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	
2	0								
3	0								
4	0								

When i =1, W =8

w_1 = 3; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is 8; therefore, we can fill the knapsack with an item of weight equal to 3.

We put profit corresponding to the weight 3, i.e., 2 at M[1][8] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0								
3	0								
4	0								

Now the value of 'i' gets incremented, and becomes 2.
When i =2, W = 1

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have only one item in the set having weight equal to 4, and the weight of the knapsack is 1. We cannot put the item of weight 4 in a knapsack, so we add 0 at M[2][1] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0							
3	0								
4	0								

When i =2, W = 2
The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have only one item in the set having weight equal to 4, and the weight of the knapsack is 2. We cannot put the item of weight 4 in a knapsack, so we add 0 at M[2][2] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0						
3	0								
4	0								

When i =2, W = 3

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 3. We can put the item of weight 3 in a knapsack, so we add 2 at M[2][3] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2					
3	0								
4	0								

When i =2, W = 4

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 4. We can put item of weight 4 in a knapsack as the profit corresponding to weight 4 is more than the item having weight 3, so we add 3 at M[2][4] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3				
3	0								
4	0								

When i = 2, W = 5

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 5. We can put item of weight 4 in a knapsack and the profit corresponding to weight is 3, so we add 3 at M[2][5] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3			
3	0								
4	0								

When i = 2, W = 6
The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 6.

We can put item of weight 4 in a knapsack and the profit corresponding to weight is 3, so we add 3 at M[2][6] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3		
3	0								
4	0								

When i = 2, W = 7
The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 7. We can put item of weight 4 and 3 in a knapsack and the profits corresponding to weights are 2 and 3; therefore, the total profit is 5, so we add 5 at M[2][7] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	0	3	3	3	5	
3	0								
4	0								

When i = 2, W = 8

The weight corresponding to the value 2 is 4, i.e., w_2 = 4. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 7. We can put item of weight 4 and 3 in a knapsack and the profits corresponding to weights are 2 and 3; therefore, the total profit is 5, so we add 5 at M[2][7] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0								
4	0								

Now the value of 'i' gets incremented, and becomes 3.
When i = 3, W = 1

The weight corresponding to the value 3 is 5, i.e., w_3 = 5. Since we have three items in the set having weights 3, 4, and 5, and the weight of the knapsack is 1. We cannot put neither of the items in a knapsack, so we add 0 at M[3][1] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0							
4	0								

When i = 3, W = 2

The weight corresponding to the value 3 is 5, i.e., w_3 = 5. Since we have three items in the set having weight 3, 4, and 5, and the weight of the knapsack is 1.

We cannot put neither of the items in a knapsack, so we add 0 at M[3][2] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0						
4	0								

When i = 3, W = 3

The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set of weight 3, 4, and 5 respectively and weight of the knapsack is 3. The item with a weight 3 can be put in the knapsack and the profit corresponding to the item is 2, so we add 2 at M[3][3] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2					
4	0								

When i = 3, W = 4

The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set of weight 3, 4, and 5 respectively, and weight of the knapsack is 4. We can keep the item of either weight 3 or 4; the profit (3) corresponding to the weight 4 is more than the profit corresponding to the weight 3 so we add 3 at M[3][4] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	1	3				
4	0								

When i = 3, W = 5

The weight corresponding to the value 3 is 5, i.e., w_3 = 5. Since we have three items in the set of weight 3, 4, and 5 respectively, and weight of the knapsack is 5. We can keep the item of either weight 3, 4 or 5; the profit (3) corresponding to the weight 4 is more than the profits corresponding to the weight 3 and 5 so we add 3 at M[3][5] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	1	3	3			
4	0								

When i =3, W = 6

The weight corresponding to the value 3 is 5, i.e., w_3 = 5. Since we have three items in the set of weight 3, 4, and 5 respectively, and weight of the knapsack is 6. We can keep the item of either weight 3, 4 or 5; the profit (3) corresponding to the weight 4 is more than the profits corresponding to the weight 3 and 5 so we add 3 at M[3][6] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	1	3	3	3		
4	0								

When i =3, W = 7

The weight corresponding to the value 3 is 5, i.e., w_3 = 5. Since we have three items in the set of weight 3, 4, and 5 respectively, and weight of the knapsack is 7. In this case, we can keep both the items of weight 3 and 4, the sum of the profit would be equal to (2 + 3), i.e., 5, so we add 5 at M[3][7] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	1	3	3	3	5	
4	0								

When i = 3, W = 8
The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set of weight 3, 4, and 5 respectively, and the weight of the knapsack is 8. In this case, we can keep both the items of weight 3 and 4, the sum of the profit would be equal to (2 + 3), i.e., 5, so we add 5 at M[3][8] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	1	3	3	3	5	5
4	0								

Now the value of 'i' gets incremented and becomes 4.
When i = 4, W = 1
The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 1. The weight of all the items is more than the weight of the knapsack, so we cannot add any item in the knapsack; Therefore, we add 0 at M[4][1] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	1	3	3	3	5	5
4	0	0							

When i = 4, W = 2

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 2. The weight of all the items is more than the weight of the knapsack, so we cannot add any item in the knapsack; Therefore, we add 0 at M[4][2] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	1	3	3	3	5	5
4	0	0	0						

When i = 4, W = 3

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 3. The item with a weight 3 can be put in the knapsack and the profit corresponding to the weight 4 is 2, so we will add 2 at M[4][3] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	1	3	3	3	5	5
4	0	0	0	2					

When i = 4, W = 4

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 4. The item with a weight 4 can be put in the knapsack and the profit corresponding to the weight 4 is 3, so we will add 3 at M[4][4] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	1	3	3	3	5	5
4	0	0	0	2	3				

When i = 4, W = 5
The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 5. The item with a weight 4 can be put in the knapsack and the profit corresponding to the weight 4 is 3, so we will add 3 at M[4][5] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	1	3	3	3	5	5
4	0	0	0	2	3	3			

When i = 4, W = 6
The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 6. In this case, we can put the items in the knapsack either of weight 3, 4, 5 or 6 but the profit, i.e., 4 corresponding to the weight 6 is highest among all the items; therefore, we add 4 at M[4][6] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	1	3	3	3	5	5
4	0	0	0	2	3	3	4		

When i = 4, W = 7

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 7. Here, if we add two items of weights 3 and 4 then it will produce the maximum profit, i.e., (2 + 3) equals to 5, so we add 5 at M[4][7] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0	0	0	2	3	3	4	5	

When i = 4, W = 8

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 8. Here, if we add two items of weights 3 and 4 then it will produce the maximum profit, i.e., (2 + 3) equals to 5, so we add 5 at M[4][8] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0	0	0	2	3	3	4	5	5

As we can observe in the above table that 5 is the maximum profit among all the entries. The pointer points to the last row and the last column having 5 value.

Now we will compare 5 value with the previous row; if the previous row, i.e., i = 3 contains the same value 5 then the pointer will shift upwards. Since the previous row contains the value 5 so the pointer will be shifted upwards as shown in the below table:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0	0	0	2	3	3	4	5	5

Again, we will compare the value 5 from the above row, i.e., i = 2. Since the above row contains the value 5 so the pointer will again be shifted upwards as shown in the below table:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0	0	0	2	3	3	4	5	5

Again, we will compare the value 5 from the above row, i.e., i = 1. Since the above row does not contain the same value so we will consider the row i=1, and the weight corresponding to the row is 4. Therefore, we have selected the weight 4 and we have rejected the weights 5 and 6 shown below:

x = { 1, 0, 0}

The profit corresponding to the weight is 3. Therefore, the remaining profit is (5 - 3) equals to 2. Now we will compare this value 2 with the row i = 2. Since the row (i = 1) contains the value 2; therefore, the pointer shifted upwards shown below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0	0	0	2	3	3	4	5	5

Again we compare the value 2 with a above row, i.e., i = 1. Since the row i =0 does not contain the value 2, so row i = 1 will be selected and the weight corresponding to the i = 1 is 3 shown below:

X = {1, 1, 0, 0}

The profit corresponding to the weight is 2. Therefore, the remaining profit is 0. We compare 0 value with the above row. Since the above row contains a 0 value but the profit corresponding to this row is 0. In this problem, two weights are selected, i.e., 3 and 4 to maximize the profit.

Matrix Chain Multiplication:

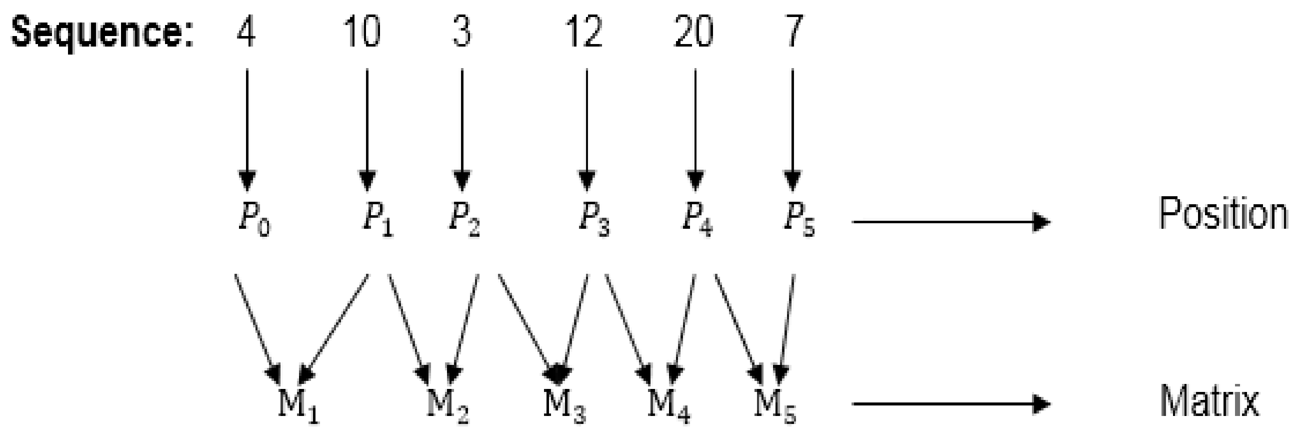
We are given the sequence {4, 10, 3, 12, 20, and 7}.

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.

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

Let us proceed with working away from the diagonal. We compute the optimal solution for the product of 2 matrices.



Here P_0 to P_5 are Position and M_1 to M_5 are matrix of size (p_i to p_{i-1})

On the basis of sequence, we make a formula

$$\text{For } M_i \longrightarrow \begin{array}{l} p\,[i] \text{ as column} \\ p\,[i\text{-}1] \text{ as row} \end{array}$$

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.

Calculation of Multiplication of matrices:

1. **m (1,2)** = m_1 x m_2
   = 4 x 10 x 10 x 3
   = 4 x 10 x 3 = **120**

2. **m (2, 3)** = m_2 x m_3
   = 10 x 3 x 3 x 12
   = 10 x 3 x 12 = **360**

3. **m (3, 4)** = m_3 x m_4
   = 3 x 12 x 12 x 20
   = 3 x 12 x 20 = **720**

4. **m (4,5)** = m_4 x m_5
   = 12 x 20 x 20 x 7
   = 12 x 20 x 7 = **1680**

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

- 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.