DESIGN AND ANALYSIS OF ALGORITHMS (DAA)

Unit-4

Dr. G. S. Naveen Kumar,

Dean, Quality

Associate Professor, CSE

Syllabus

Dynamic Programming- General Method, applications- Chained matrix multiplication, All pairs shortest path problem, 0/1 knapsack problem, Traveling sales person problem.

Dynamic Programming

Definition

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.

- ✓ Dynamic programming is a technique.
- ✓ It is 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 sub problems are optimized to optimize the overall solution is known as optimal substructure property.
- ✓ The main use of dynamic programming is to solve optimization problems.
- ✓ we are trying to find out the minimum or the maximum solution of a problem.

PRINCIPLE OF OPTIMALITY

- ✓ The principle of optimality is the basic principle of dynamic programming, which was developed by Richard Bellman: that an optimal path has the property that whatever the initial conditions and control variables (choices) over some initial period, the control (or decision variables) chosen over the remaining period must be optimal for the remaining problem, with the state resulting from the early decisions taken to be the initial condition.

Consider an example of the Fibonacci series. The following series is the Fibonacci series:

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

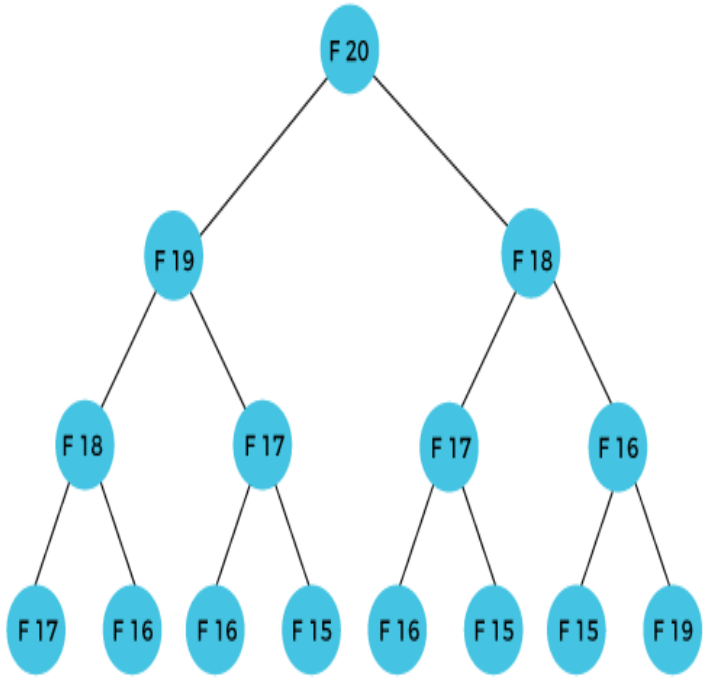
The numbers in the above series are not randomly calculated. Mathematically, we could write each of the terms using the below formula:

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.



The solution,

- 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 subproblems (memoization). 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:

1. Top-down approach
2. 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.

Advantages

- ✓ It is very easy to understand and implement.
- ✓ It solves the sub problems only when it is required.
- ✓ It is easy to debug.

Disadvantages

- ❑ It uses the recursion technique that occupies more memory in the call stack.
- ❑ Sometimes when the recursion is too deep, the stack overflow condition will occur.
- ❑ It occupies more memory that degrades the overall performance.

Let's understand dynamic programming through an example.

```
int fib(int n)
{
   if(n<0)
   error;
 if(n==0)
 return 0;
 if(n==1)
return 1;
sum = fib(n-1) + fib(n-2);
}
```

- In this code, we have used the recursive approach to find out the Fibonacci series.
- When the value of 'n' increases, the function calls will also increase, and computations will also increase.
- In this case, the time complexity increases exponentially, and it becomes 2n.
- One solution to this problem is to use the dynamic programming approach.
- Rather than generating the recursive tree again and again,
- we can reuse the previously calculated value.
- If we use the dynamic programming approach, then the time complexity would be O(n).

Bottom-Up approach

- 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.

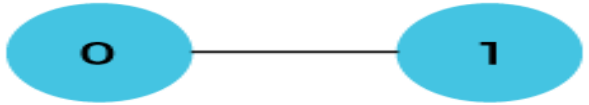
Key points

- We solve all the smaller sub-problems that will be needed to solve the larger sub-problems then move to the larger problems using smaller sub-problems.
- We use for loop to iterate over the sub-problems.
- The bottom-up approach is also known as the tabulation or table filling method.

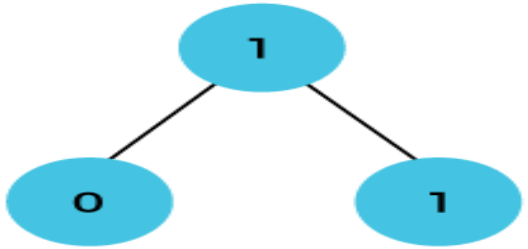
Bottom-Up approach

Let's understand through the diagrammatic representation.

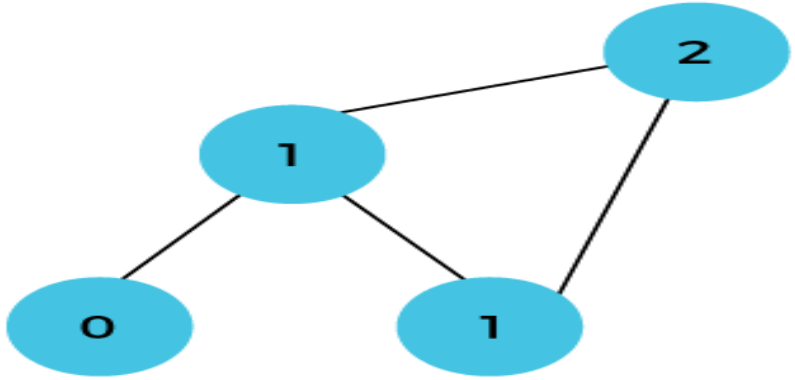
Initially, the first two values, i.e., 0 and 1 can be represented as:



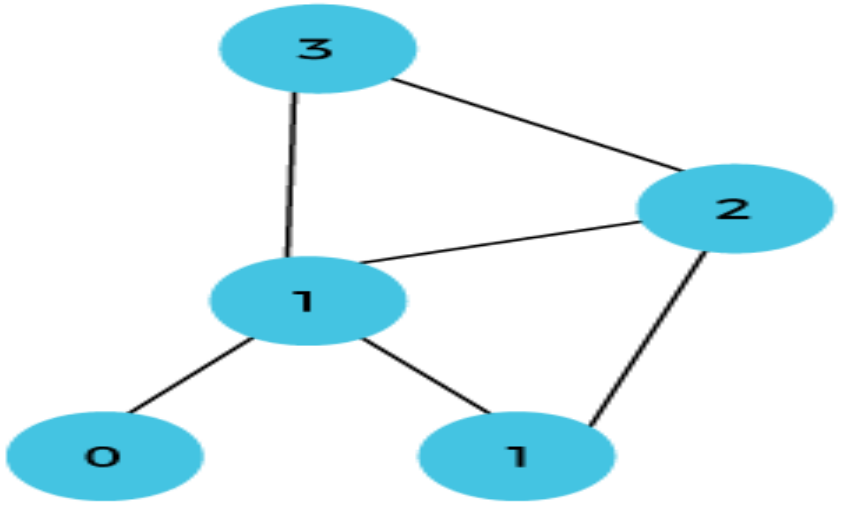
When i=2 then the values 0 and 1 are added shown as below:



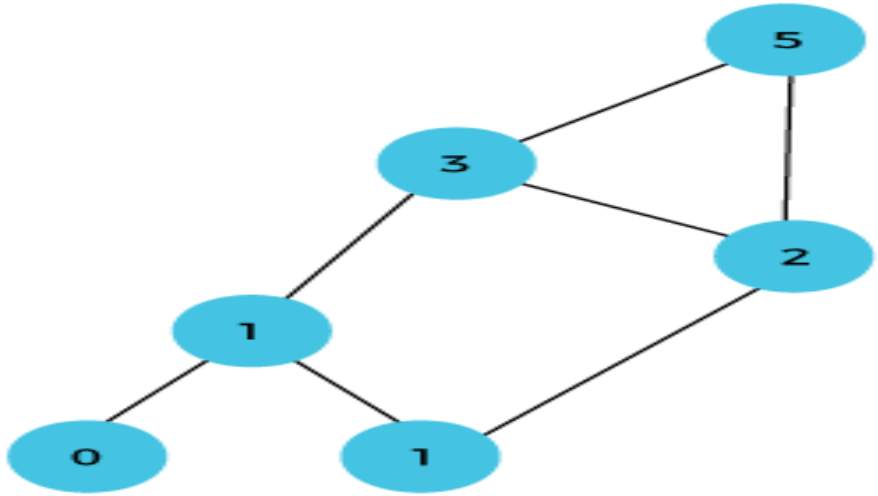
When i=3 then the values 1and 1 are added shown as below:



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.

DIFFERENCES BETWEEN GREEDY METHOD AND DYNAMIC PROGRAMMING

Feature	Greedy method	Dynamic programming
Feasibility	In a greedy Algorithm, we make whatever choice seems best at the moment in the hope that it will lead to global optimal solution.	In Dynamic Programming we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution .
Optimality	In Greedy Method, sometimes there is no such guarantee of getting Optimal Solution.	It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then choose the best.
Recursion	A greedy method follows the problem solving heuristic of making the locally optimal choice at each stage.	A Dynamic programming is an algorithmic technique which is usually based on a recurrent formula that uses some previously calculated states.
Memoization	It is more efficient in terms of memory as it never look back or revise previous choices	It requires dp table for memoization and it increases it's memory complexity.
Time complexity	Greedy methods are generally faster. For example, Dijkstra's shortest path algorithm takes O(ELogV + VLogV) time.	Dynamic Programming is generally slower. For example, Bellman Ford algorithm takes O(VE) time.
Fashion	The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.	Dynamic programming computes its solution bottom up or top down by synthesizing them from smaller optimal sub solutions.
Example	Fractional knapsack .	0/1 knapsack problem

The various applications of Dynamic Programming are :

1. Longest Common Subsequence.
2. Finding Shortest Path.
3. Finding Maximum Profit with other Fixed Constraints.
4. Job Scheduling in Processor.
5. Bio Informatics.
6. Optimal search solutions.

Dynamic Programming : Matrix Chain Multiplication

It is a Method under Dynamic Programming in which previous output is taken as input for next.

Here, Chain means one matrix's column is equal to the second matrix's row [always].

In general:

```
If A = [a_ij] is a p x q matrix
   B = [b_ij] is a q x r matrix
   C = [c_ij] is a p x r matrix
```

$AB = C$ if $c_{ij} = \sum_{k=1}^{q} a_{ik}\, b_{kj}$

Given following matrices $\{A_1, A_2, A_3, \ldots A_n\}$ and we have to perform the matrix multiplication, which can be accomplished by a series of matrix multiplications

$A_1 \times A_2 \times A_3 \times \ldots\ldots \times A_n$

Dynamic Programming : Matrix Chain Multiplication

Matrix Multiplication operation is **associative** in nature rather commutative. By this, we mean that we have to follow the above matrix order for multiplication but we are free to **parenthesize** the above multiplication depending upon our need.

In general, for $1 \le i \le p$ and $1 \le j \le r$

$$C[i, j] = \sum_{k=1}^{q} A[i, k]B[k, j]$$

Example1: Let us have 3 matrices, A_1, A_2, A_3 of order (10 x 100), (100 x 5) and (5 x 50) respectively.

Three Matrices can be multiplied in two ways:

1. **$A_1, (A_2, A_3)$:** First multiplying(A_2 and A_3) then multiplying and resultant withA_1.
2. **$(A_1, A_2), A_3$:** First multiplying(A_1 and A_2) then multiplying and resultant withA_3.

Dynamic Programming : Matrix Chain Multiplication

No of Scalar multiplication in Case 1 will be:

> (100 x 5 x 50) + (10 x 100 x 50) = 25000 + 50000 = 75000

No of Scalar multiplication in Case 2 will be:

> (100 x 10 x 5) + (10 x 5 x 50) = 5000 + 2500 = 7500

Number of ways for parenthesizing the matrices:

There are very large numbers of ways of parenthesizing these matrices. If there are n items, there are (n-1) ways in which the outer most pair of parenthesis can place.

Dynamic Programming : Matrix Chain Multiplication

$(A_1)\ (A_2, A_3, A_4, \ldots\ldots\ldots\ldots\ldots A_n)$
Or $(A_1, A_2)\ \ (A_3, A_4\ \ldots\ldots\ldots\ldots\ldots A_n)$
Or $(A_1, A_2, A_3)\ \ (A_4\ \ldots\ldots\ldots\ldots\ldots A_n)$
$\ldots\ldots\ldots\ldots\ldots\ldots\ldots$

Or$(A_1, A_2, A_3 \ldots\ldots\ldots\ldots A_{n-1})\ (A_n)$

It can be observed that after splitting the kth matrices, we are left with two parenthesized sequence of matrices: one consist 'k' matrices and another consist 'n-k' matrices.

Now there are 'L' ways of parenthesizing the left sublist and 'R' ways of parenthesizing the right sublist then the Total will be L.R:

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

Also p (n) = c (n-1) where c (n) is the nth **Catalon number**

$$c(n) = \frac{1}{n+1}\binom{2n}{n}$$

On applying Stirling's formula we have

$$c(n) = \Omega\left(\frac{4^n}{n^{1.5}}\right)$$

Which shows that 4^n grows faster, as it is an exponential function, then $n^{1.5}$.

Development of Dynamic Programming Algorithm

1. Characterize the structure of an optimal solution.
2. Define the value of an optimal solution recursively.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct the optimal solution from the computed information.

Dynamic Programming Approach

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 subproblems whose solution can be combined to solve the global problem.

At the greatest level of parenthesization, we multiply two matrices

> $A_{1.....n}=A_{1....k}$ x $A_{k+1....n})$

Thus we are left with two questions:

- How to split the sequence of matrices?
- How to parenthesize the subsequence $A_{1.....k}$ and$A_{k+1.....n}$?

Dynamic Programming : Matrix Chain Multiplication

Step1: Structure of an optimal parenthesization: Our first step in the dynamic paradigm is to find the optimal substructure and then use it to construct an optimal solution to the problem from an optimal solution to subproblems.

Let $A_{i....j}$ where $i \leq j$ denotes the matrix that results from evaluating the product

$A_i A_{i+1}....A_j$.

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

Step 2: A Recursive Solution: Let m [i, j] be the minimum number of scalar multiplication needed to compute the matrix$A_{i....j}$.

If i=j the chain consist of just one matrix $A_{i....i}=A_i$ so no scalar multiplication are necessary to compute the product. Thus m [i, j] = 0 for i= 1, 2, 3....n.

If i<j we assume that to optimally parenthesize the product we split it between A_k and A_{k+1} where i≤ k ≤j. Then m [i,j] equals the minimum cost for computing the subproducts $A_{i....k}$ and $A_{k+1....j}$+ cost of multiplying them together. We know A_i has dimension p_{i-1} x p_i, so computing the product $A_{i....k}$ and $A_{k+1....j}$takes p_{i-1} p_k p_j scalar multiplication, we obtain

```
m [i,j] = m [i, k] + m [k + 1, j] + p_{i-1}  p_k p_j
```

There are only (j-1) possible values for 'k' namely k = i, i+1.....j-1. Since the optimal parenthesization must use one of these values for 'k' we need only check them all to find the best.

So the minimum cost of parenthesizing the product A_i A_{i+1}.....A_j becomes

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

To construct an optimal solution, let us define s [i,j] to be the value of 'k' at which we can split the product A_i A_{i+1} .....A_j To obtain an optimal parenthesization i.e. s [i, j] = k such that

```
m [i,j] = m [i, k] + m [k + 1, j] + p_{i-1}  p_k p_j
```

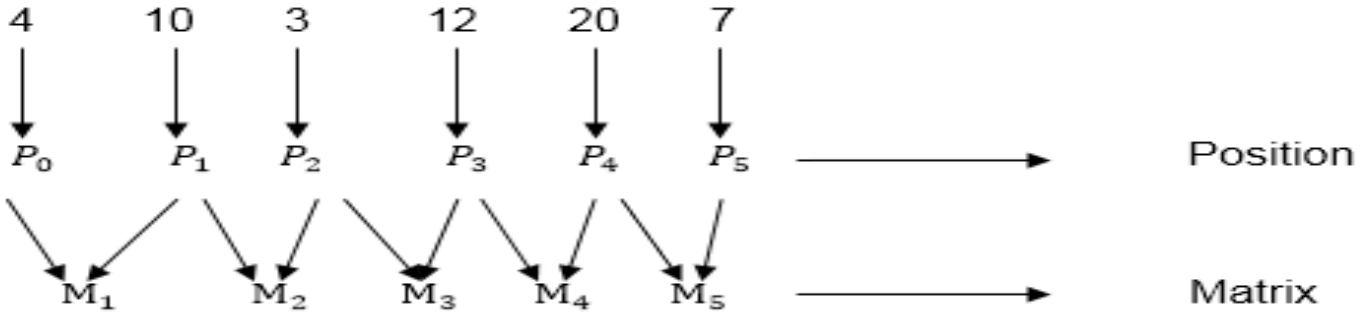
Dynamic Programming : Example of Matrix Chain Multiplication

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.

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.

Sequence: 4 10 3 12 20 7



For M_i → p [i] as column

p [i-1] as row

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

Example of Matrix Chain Multiplication

Calculation of Product of 2 matrices:

1. $m(1,2) = m_1 \times m_2$

   $= 4 \times 10 \times 10 \times 3$

   $= 4 \times 10 \times 3 = 120$

2. $m(2,3) = m_2 \times m_3$

   $= 10 \times 3 \times 3 \times 12$

   $= 10 \times 3 \times 12 = 360$

3. $m(3,4) = m_3 \times m_4$

   $= 3 \times 12 \times 12 \times 20$

   $= 3 \times 12 \times 20 = 720$

4. $m(4,5) = m_4 \times m_5$

   $= 12 \times 20 \times 20 \times 7$

   $= 12 \times 20 \times 7 = 1680$

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

Now product of 3 matrices:

```
M [1, 3] = M1 M2 M3
```

1. There are two cases by which we can solve this multiplication: (M_1 x M_2) + M_3, M_1+ (M_2x M_3)
2. After solving both cases we choose the case in which minimum output is there.

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

M [1, 3] =264

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] = M2 M3 M4
```

1. There are two cases by which we can solve this multiplication: (M_2x M_3)+M_4, M_2+(M_3 x M_4)
2. After solving both cases we choose the case in which minimum output is there.

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

M [2, 4] = 1320

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_4xM_5)
2. After solving both cases we choose the case in which minimum output is there.

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

M [3, 5] = 1140

As Comparing both output **1140** is minimum in both cases so we insert **1140** in table and (M_3 x M_4) + M_5this 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

Example of Matrix Chain Multiplication

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

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

Dynamic Programming : Example of Matrix Chain Multiplication

MALLA REDDY UNIVERSITY

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

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

$$M[2,5] = \min \begin{cases} M[2,4] + M[5,5] + p_1 p_4 p_5 = 1320 + 0 + 10.20.7 = 2720 \\ M[2,3] + M[4,5] + p_1 p_3 p_5 = 360 + 1680 + 10.12.7 = 2880 \\ M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 1140 + 10.3.7 = 1350 \end{cases}$$

$M\ [2,\ 5] = 1350$

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

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. $(M_1 \times M_2 \times M_3 \times M_4) \times M_5$
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. $M_1 \times M_2 \times (M_3 \times M_4 \times M_5)$

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

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
```

Algorithm of Matrix Chain Multiplication

MATRIX-CHAIN-ORDER (p)

```
1. n   length[p]-1
2. for i ← 1 to n
3. do m [i, i] ← 0
4. for l ← 2 to n    // l is the chain length
5. do for i ← 1 to n-l + 1
6. do j ← i+ l -1
7. m[i,j] ←  ∞
8. for k  ← i to j-1
9. do q  ← m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j
10. If q < m [i,j]
11. then m [i,j] ← q
12. s [i,j] ← k
13. return m and s.
```

Analysis: There are three nested loops. Each loop executes a maximum n times.

1. l, length, O (n) iterations.
2. i, start, O (n) iterations.
3. k, split point, O (n) iterations

Body of loop constant complexity

Total Complexity is: O (n^3)

What is All Pairs Shortest Path Problem?

The all-pairs shortest path problem is the determination of the shortest graph distances between every pair of vertices in a given graph. We have to calculate the minimum cost to find the shortest path.

All-Pairs Shortest Paths

Procedure to find the all pairs shortest path:

- First we consider "G" as a directed graph
- The cost of the graph is the length or cost of each edges and cost(i,i)=0
- If there is an edge between i and J then cost of (i,j)=length/cost of the edge from i to j and id there is no edge then (i, j)= ∞
- Need to calculate the shortest path/ cost between ant two nodes using intermediary nodes.
- The following equation is used to calculate the minimum cost

$$A^k(i,j)=\min\{A^{k-1}(i,j), A^{k-1}(i,k), A^{k-1}(k,j)\}$$

All-Pairs Shortest Paths

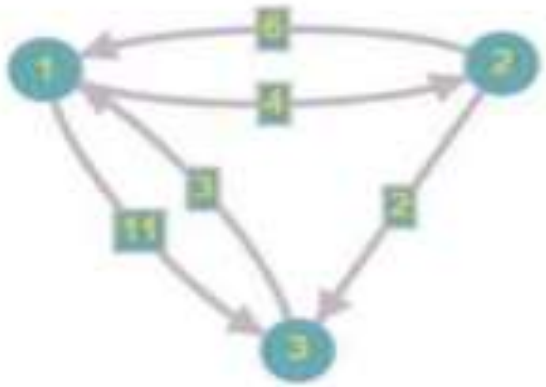
Algorithm of All Pairs Shortest Path

```
Algorithm AllPaths(cost, A, n){
        for i :=1to n do
                for j :=1to n do
                A[i, j]:=cost[i,j];
        for k :=1to n do
                for i :=1to n do
                        for j :=1to n do
                        A[i,j]:=min(A[i,j],A[i,k]+A[k,j];
}
```

All-Pairs Shortest Paths

Example:

Here, $A^0 = Cost = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$



When we calculate A^1 will omit column 1 and row 1 and calculate cost for rest of the 4 element.

$A^1(2,3) = \min\{A^{1-1}(2,3), A^{1-1}(2,1) + A^{1-1}(1,3)\}$

$= \min\{2,17\} = 2$

$A^1(3,2) = \min\{A^{1-1}(3,2), A^{1-1}(3,1) + A^{1-1}(1,2)\}$

$= \min\{\infty,7\} = 7$

$A^1 = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$

All-Pairs Shortest Paths

Now we will calculate A^2

$A^2(1,3)=\min\{A^{2-1}(1,3),\ A^{2-1}(1,2)+A^{2-1}(2,3)\}$

$=\min\{11,6\}$

$=6$

$A^2(3,1)=\min\{A^{2-1}(3,1),\ A^{2-1}(3,2)+A^{2-1}(2,1)\}$

$=\min\{3,13\}$

$=3$

$$A^2=\begin{bmatrix}0 & 4 & 6\\ 6 & 0 & 2\\ 3 & 7 & 0\end{bmatrix}$$

All-Pairs Shortest Paths

Now we will calculate A^2

$A^3(1,2)=\min\{A^{3-1}(1,2), A^{3-1}(1,3)+ A^{3-1}(3,2)\}$

$=\min\{4,13\}$

$=4$

$A^3(2,1)=\min\{A^{3-1}(2,1), A^{3-1}(2,3)+ A^{3-1}(3,1)\}$

$=\min\{6,5\}$

$=5$

$$A^3=\begin{bmatrix}0 & 4 & 6\\ 5 & 0 & 2\\ 3 & 7 & 0\end{bmatrix}$$

Applications of All Pairs Shortest Path

- Road Networking
- Network Routing
- Flight Reservations
- Driving Directions

TRAVELLING SALES PERSON PROBLEM

Algorithm: Traveling-Salesman-Problem

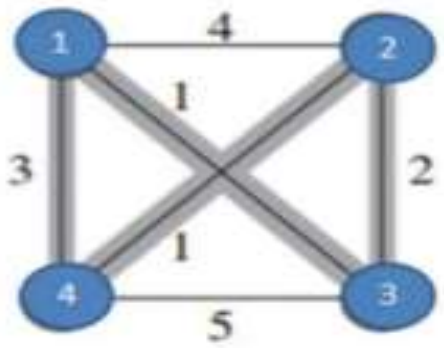
1. $C(\{1\}, 1) = 0$
2. for $s = 2$ to n do
3. for all subsets $S \in \{1, 2, 3, \ldots, n\}$ of size s and containing 1
4. $C(S, 1) = \infty$
5. for all $j \in S$ and $j \neq 1$
6. $C(S, j) = min\{C(S - \{j\}, i) + d(i, j)\ for\ i \in S\ and\ i \neq j\}$
7. Return $min_j\ C(\{1, 2, 3, \ldots, n\}, j) + d(j, i)$

Time Complexity

- There are at the most $2\ n$ .n sub-problems and each one takes linear time to solve. Therefore, the total running time is $O(2\ n\ .n\ 2\ )$.

Simple Example

IS there a route that takes you through every city and back to the starting point 1 for less than 7.



The solution is : 13241

Applications

- A real-world application that calculates the route of the Travelling Salesman Problem using the current traffic intensity information from Google Maps is prepared.
- A user-friendly interface, displaying the shortest route in distance or duration on Google Maps, has been developed by adding different features.

TRAVELLING SALESPERSON PROBLEM:

Let G = (V, E) be a directed graph with edge costs c_{ij}. The variable c_{ij} is defined such that $c_{ij} > 0$ for all I and j and $c_{ij} = \alpha$ if $< i, j> \notin E$. Let $|V| = n$ and assume $n > 1$. A tour of G is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The traveling sales person problem is to find a tour of minimum cost. The tour is to be a simple path that starts and ends at vertex 1.

Let g (i, S) be the length of shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1. The function g (1, V – {1}) is the length of an optimal salesperson tour. From the principal of optimality it follows that:

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, k\})\} \quad -- \quad 1$$

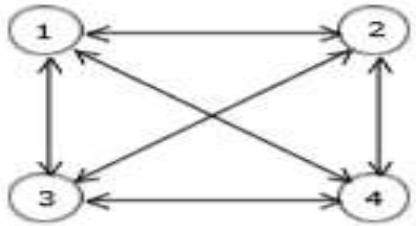
Generalizing equation 1, we obtain (for $i \notin S$)

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\} \quad -- \quad 2$$

The Equation can be solved for g (1, V – 1}) if we know g (k, V – {1, k}) for all choices of k.

Example :

For the following graph find minimum cost tour for the traveling salesperson problem:



The cost adjacency matrix =

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

Let us start the tour from vertex 1:

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, K\})\} \quad - \quad (1)$$

More generally writing:

$$g(i, s) = \min \{c_{ij} + g(J, s - \{J\})\} \quad - \quad (2)$$

Clearly, $g(i, \Phi) = c_{i1}$, $1 \le i \le n$. So,

$g(2, \Phi) = C_{21} = 5$

$g(3, \Phi) = C_{31} = 6$

$g(4, \Phi) = C_{41} = 8$

Using equation – (2) we obtain:

$g(1, \{2, 3, 4\}) = \min \{c_{12} + g(2, \{3, 4\}, c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\}$

$g(2, \{3, 4\}) = \min \{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\}$
$= \min \{9 + g(3, \{4\}), 10 + g(4, \{3\})\}$

$g(3, \{4\}) = \min \{c_{34} + g(4, \Phi)\} = 12 + 8 = 20$

$g(4, \{3\}) = \min \{c_{43} + g(3, \Phi)\} = 9 + 6 = 15$

Therefore, $g(2, \{3, 4\}) = \min \{9 + 20, 10 + 15\} = \min \{29, 25\} = 25$

$g(3, \{2, 4\}) = \min \{(c_{32} + g(2, \{4\}), (c_{34} + g(4, \{2\})\}$

$g(2, \{4\}) = \min \{c_{24} + g(4, \Phi)\} = 10 + 8 = 18$

$g(4, \{2\}) = \min \{c_{42} + g(2, \Phi)\} = 8 + 5 = 13$

Therefore, $g(3, \{2, 4\}) = \min \{13 + 18, 12 + 13\} = \min \{41, 25\} = 25$

$g(4, \{2, 3\}) = \min \{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\}$

$g(2, \{3\}) = \min \{c_{23} + g(3, \Phi\} = 9 + 6 = 15$

$g(3, \{2\}) = \min \{c_{32} + g(2, \Phi\} = 13 + 5 = 18$

Therefore, $g(4, \{2, 3\}) = \min \{8 + 15, 9 + 18\} = \min \{23, 27\} = 23$

$g(1, \{2, 3, 4\}) = \min \{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\}$
$= \min \{10 + 25, 15 + 25, 20 + 23\} = \min \{35, 40, 43\} = 35$

The optimal tour for the graph has length = 35

The optimal tour is: 1, 2, 4, 3, 1.

Conclusion

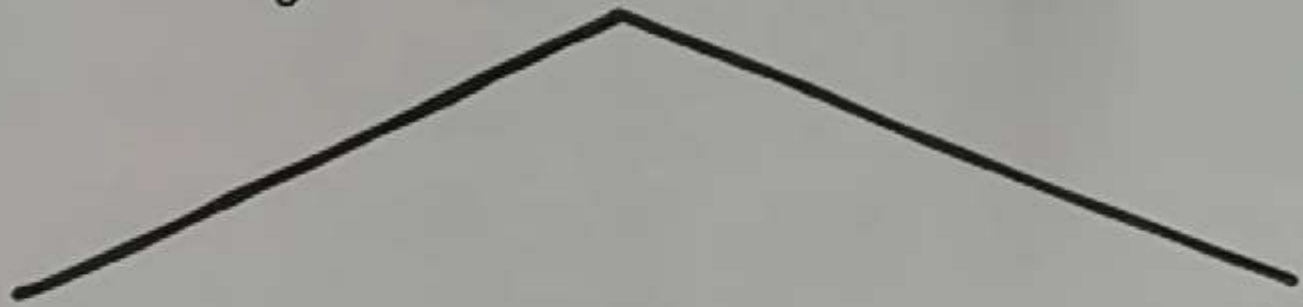
- It can be concluded that on application of TSP algorithm with dynamic programming is able to produce optimal route tour to serve a customers including the shortest or minimum length of travel time or optimal travel time.

Problem :-

You are given a knapsack with a limited weight capacity and so items each of which have a weight and a value.

The problem is - " which items to place in the knapsack such that the weight limit is not exceeded and the total value of the items is as large as possible ? "

Knapsack Problem Variants



0/1 Knapsack Problem

- Items are indivisible i.e. you cannot break an item, you either take an item or not.
- Solved with a dynamic programming approach.

Fractional Knapsack Problem

- Items are divisible i.e. you can take any fraction of an item.
- Solved with a greedy approach.

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

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									

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 = 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.