UNIT-III

DYNAMIC PROGRAMMING

→ This technique is used to solve optimization problems.
→ In dynamic programming, we obtain the solution to a problem by performing a sequence of decisions. We examine the decision sequence to see, if the optimal decision sequence contains optimal decision subsequences.
→ In dynamic programming, an optimal sequence of decisions is obtained by using the principle of optimality.

PRINCIPLE OF OPTIMALITY:

The principle of optimality states that, "In an optimal sequence of decisions each subsequence must also be optimal".
(OR)
The optimal solution to a problem is composed of optimal solutions to the subproblems.

0/1 KNAPSACK PROBLEM:

→This is similar to fractional knapsack problem, except that x_i's are restricted to have a value either 0 or 1.

→ We are given a set of 'n' items (or, objects), such that each item i has a weight 'w_i' and a profit 'p_i'. We wish to pack a knapsack whose capacity is 'M' with a subset of items such that total profit is maximized.

→ The solution to this problem is expressed as a vector $(x_1, x_2, \ldots, x_n)$,where each x_i is 1 if object i has been placed in knapsack, otherwise x_i is 0.

→ The mathematical formulation of the problem is:

$$\text{maximize } \sum_{i=1}^{n} p_i x_i$$

subject to the constraints:

$$\sum_{i=1}^{n} w_i x_i \leq M$$

and

$$x_i \in \{0,1\},\ 1 \leq i \leq n$$

→ We need to make the decisions on the values of $x_1, x_2, x_3, \ldots, x_n$.

→ When optimal decision sequence contains optimal decision subsequences, we can establish recurrence equations that enable us to solve the problem in an efficient way.

→ We can formulate recurrence equations in two ways:
Forward approach
Backward approach

Recurrence equation for 0/1 knapsack problem using Forward approach:
For the 0/1 knapsack problem, optimal decision sequence is composed of optimal decision subsequences.

Let *f(i, y)* denote the value (or profit), of an optimal solution to the knapsack instance with remaining capacity *y* and remaining objects *i, i+1,..., n* for which decisions have to be taken.

It follows that

$$f(n,y) = \begin{cases} p_n, & if\ w_n \le y \\ 0, & if\ w_n > y \end{cases} \quad \text{-------(1)}$$

and

$$f(i,y) = \begin{cases} max\left(f(i+1,y),\ p_i + f(i+1, y - w_i)\right), & if\ w_i \le y \\ f(i+1,y), & if\ w_i > y \end{cases} \quad \text{--(2)}$$

NOTE: when $w_i > y$, we cannot place the object *i*, and there is no choice to make, but when $w_i \le y$ we have two choices, viz., if object *i* can be placed or not. Here we will consider the choice into account which results in maximum profit.

→ *f(1, M)* is the value (profit) of the optimal solution to the knapsack problem we started with. Equation (2) may be used recursively to determine *f(1, M)*.

Example 1: Let us determine *f(1, M)* recursively for the following 0/1 knapsack instance: $n=3$; $(w_1, w_2, w_3) = (100,14,10)$; $(p_1, p_2, p_3) = (20,18,15)$ and $M=116$.

Solution:

$f(1,116) = max\{f(2, 116),\ 20 + f(2, 116 - 100)\}$, since $w1<116$
$= max\{f(2, 116),\ 20 + f(2, 16)\}$
$f(2,116) = max\{f(3, 116), 18 + f(3, 116 - 14)\}$, since $w2<116$

$= max\{f(3, 116), 18 + f(3, 102)\}$

$f(3,116) = 15$ *(since w3<116)*

$f(3,102) = 15$ *(since w3<102)*

$f(2,116) = max\{15, (18 + 15)\} = max\{15, 33\} = 33$

now,

$f(2, 16) = max\{f(3, 16), 18 + f(3,2)\}$

$f(3,16) = 15$ *(since w3<16)*

$f(3,2) = 0$ *(since w3>2)*

So, $f(2,16) = max(15, 18+0) = 18$

$f(1,116) = max\{33, (20 + 18)\} = 38$

Tracing back the x_i values:

To obtain the values of x_i's, we proceed as follows:

If $f(1, M) = f(2,M)$ then we may set $x_1=0$.

If $f(1, M) \neq f(2,M)$ then we may set $x1=1$. Next, we need to find the optimal solution that uses the remaining capacity $M-w1$. This solution has the value $f(2, M-w1)$. Proceeding in this way, we may determine the values of all the x_i's.

Determining the x_i values for the above example:

$f(1, 116) = 38$

$f(2, 116) = 33$

Since $f(1, 116) \neq f(2, 116) \rightarrow x_1 = 1$.

After placing object *1*, remaining capacity is *116 -100 =16*. This will lead to *f(2, 16)*.

$f(2,16)=18$

$f(3,16)=15$

Since $f(2, 16) \neq f(3, 16) \rightarrow x_2 = 1$.

After placing object *2*, remaining capacity is *16-14=2*, this will lead to *f(3,2)*.

Since $f(3, 2) = 0 \rightarrow x_3 = 0$.

So, optimal solution is: $(x_1, x_2, x_3) = (1,1,0)$, which yields the profit of 38.

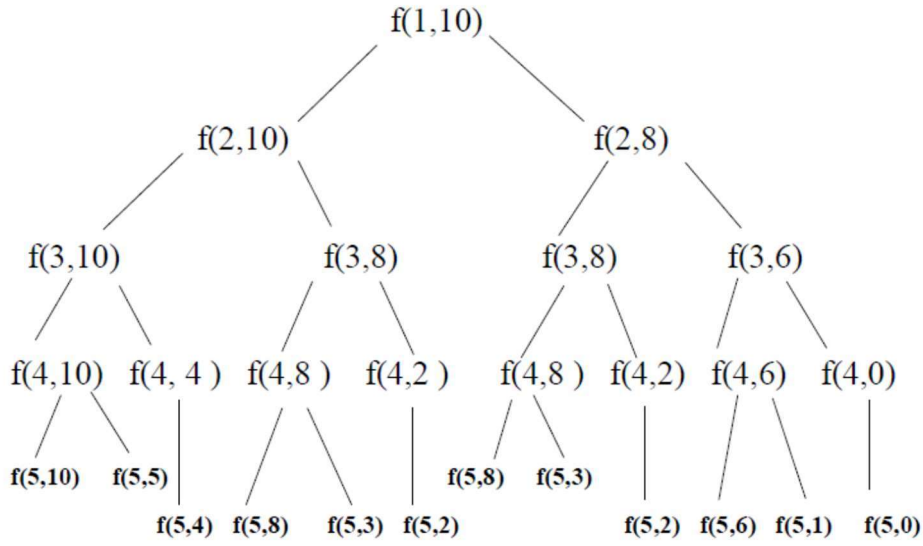
ALOGORITHM FOR 0/1 KNAPSACK PROBLEM USING FORWARD APPROCH:

```
Algorithm f(i, y)
{
    if(i=n) then
    {
        if(w[n]>y) then return 0;
        else return p[n];
    }
```

if(w[i]>y) **then return** f(i+1, y);
else return max [f(i + 1, y), (p_i + f(i + 1, $y - w_i$))];
}
→ INITIALLY THE ABOVE ALGORITHM IS INVOKED AS *f(1, M)*.

TIME COMPLEXITY:
→ let *T(n)* be the time this code takes to solve an instance with *n* objects.
So, $T(n) = \begin{cases} c, & if\ n = 1 \\ 2T(n-1) + c, & if\ n > 1 \end{cases}$
Solving this, we get time complexity equal to $O(2^n)$.
→ In general, if there are *d* choices for each of the *n* decisions to be made, there will be d^n possible decision sequences.

Example: Consider the case *n=5; M=10; (w_1, w_2, w_3, w_4, w_5) =(2,2,6,5,4); (p_1, p_2, p_3, p_4, p_5) = (6,3,5,4,6).*
Solution: To determine *f(1,10)*, function *f* is invoked as *f(1,10)*. The recursive calls made are shown by the following tree of recursive calls.



→ 27 invocations are done on *f*. We notice that several invocations redo the work of previous invocation. For example: *f(3,8), f(4,8), f(4,2), f(5,8), f(5,3), f(5,2)* are computed twice.
If we save the result of previous invocations, we can reduce the number of invocations to 21.

We maintain a table to store the values, as and when the function call computes there.
By using these table values, we can avoid re-computing the function, when it is again invoked.
When the recursive program is designed to avoid the re-computation, the complexity is drastically reduced from exponential to polynomial.

Recurrence equation for 0/1 knapsack problem using backward approach:

Let *f(i, y)* denote the value (or, profit) of an optimal solution to the knapsack instance with remaining capacity *y* and remaining objects *i, i-1, ... ,1* for which decisions have to be taken.
It follows that

$$f(1,y) = \begin{cases} p_1, & if\ w_1 \le y \\ 0, & if\ w_1 > y \end{cases} \quad \text{-------(1)}$$

and

$$f(i,y) = \begin{cases} max\left(f(i-1,y),\ p_i + f(i-1,y-w_i)\right), & if\ w_i \le y \\ f(i-1,y), & if\ w_i > y \end{cases} \quad \text{--(2)}$$

→ *f(n, M)* gives the value (or profit) of the optimal solution by including objects *n, n-1, ..., 1*.

Example: Let us determine *f(n, M)* recursively for the following 0/1 knapsack instance: *n=3; (w_1, w_2, w_3) = (2,3,4); (p_1, p_2, p_3) = (1,2,5) and M=6.*

Solution:

f(3,6) =max{f (2, 6), 5 + f(2, 2)}
f(2,6) =max{f (1, 6), 2 + f(1, 3)}
f(1,6) = p1 = 1
f(1,3) = p1 = 1
f(2,6) = max{1, 2+1} = 3
f(2,2) = f(1,2) = f(1,2) = 1
Now,
f(3,6) = max{3, 5+1} = 6
Tracing back values of x_i:
f(3,6) = 6
f(2,6) = 3
Since *f(3,6) ≠ f(2,6),* x_3*=1.*
After including object *3*, remaining capacity becomes *6-4= 2*, this willlead to the problem *f(2,2).*
f(2,2) = 1
f(1,2) = 1
Since *f(2,2) = f(1,2),* x_2*=0,* and this will lead to the problem *f(1,2).*

Since *f(1,2) = 1, x_1= 1.*
So, optimal solution is: $(x_1, x_2, x_3) = (1,0,1)$.

Solving 0/1 knapsack problem using Sets of Ordered Pairs:

→Let s^i represent the possible state, resulting from the 2^i decision sequences for $x_1, x_2, x_3 \dots \dots x_i$.

→A state refers to a tuple (p_j, w_j), w_j being the total weight of objects included in the knapsack and p_j being the corresponding profit.

NOTE: $s^0 = \{(0,0)\}$

→To obtain s^{i+1} from s^i, we note that the possibilities for x_{i+1} are *1* and *0*.

→When $x_{i+1} = 0$, the resulting states are same as for s^i.

When x_{i+1} =1, the resulting states are obtained by adding (p_{i+1}, w_{i+1}) to each state in s^i.We call the set of these additional states s_1^i.

→Now s^{i+1}can be computed by merging the states in s^i *and* s_1^i together, i.e., $s^{i+1} = s^i \cup s_1^i$. *The states in* s^{i+1} *set should be arranged in the increasing order of profits.*

NOTE:

1. If s^{i+1} contains two pairs (p_j, w_j) and (p_k, w_k) with the property that $p_j \leq p_k$ and $w_j \geq w_k$, we say that (p_k, w_k) dominates (p_j, w_j) and the dominated tuple (p_j, w_j) can be discarded from s^{i+1}. This rule is called purging rule.
2. By this rule all duplicate tuples will also be purged.
3. We can also purge all pairs (p_j, w_j) with $w_j > M$.

Finally profit for the optimal solution is given by *p* value of the last pair in s^n set.

Tracing back values of x_i's:

→Suppose that (p_k, w_k) is the last tuple in s^n, then a set of 0/1 values for the x_i's can be determined by carrying out a search through the s^i sets.

→ if $(p_k, w_k) \in s^{n-1}$, then we will set $x_n = 0$.

If $(p_k, w_k) \notin s^{n-1}$, then $(p_k - p_n, w_k - w_n) \in s^{n-1}$and we will set $x_n = 1$. This process can be done recursively to get remaining x_i values.

Example: Consider the knapsack instance:

n = 3, $(w_1, w_2, w_3) = (2, 3, 4)$ and $(p_1, p_2, p_3) = (1,2,5)$, and *M = 6*. Generate the sets s^i and find the optimal solution.

Solution:

$s^0 = \{(0,0)\}$;

By including object 1,
s_1^0 = {(1,2)};
By merging s^0 *and* s_1^0 , we get
s^1 = {(0,0), (1,2)};
By including object 2,
s_1^1={(2,3), (3, 5)};
By merging s^1 *and* s_1^1, we get
s^2 = {(0, 0), (1, 2), (2, 3), (3, 5)};
By including object 3,
s_1^2= {(5, 4), (6, 6), (7, 7), (8, 9)} = {(5, 4), (6, 6)};
By merging s^2 *and* s_1^2, we get
s^3= {(0,0), (1,2), (2, 3), (3, 5), (5, 4), (6, 6)};
By applying the purging rule, the tuple (3, 5) will get discarded.
s^3={(0,0), (1,2), (2, 3), (5, 4), (6, 6)};
Tracing out the values of x_i :
→The last tuple in s^3 is (6,6) $\notin s^2$. So, x_3=1.
→The last tuple (6,6) of s^3came from a tuple $(6 - p_3, 6 - w_3)$ = (6-5, 6-4) = (1, 2) belonging to s^2.
→The tuple (1, 2) of s^2, is also present in s^1. So, $x_2 = 0$.
→The tuple (1, 2) of s^1, is not present in s^0. So, $x_1 = 1$.
So, the optimal solution is: (x_1, x_2, x_3) = (1,0,1)

Example: Consider the knapsack instance:
n = 5, $(w_1, w_2, w_3, w_4, w_5)$ *= (2, 2, 6, 5, 4)* and $(p_1, p_2, p_3, p_4, p_5)$ *= (6,3,5,4,6)*, and *M = 10*. Generate the sets s^i and find the optimal solution.

Sol:- given $n = 5$, $(w_1, w_2, w_3, w_4, w_5) = (2, 2, 6, 5, 4)$,
$(P_1, P_2, P_3, P_4, P_5) = (6, 3, 5, 4, 6)$, and $M = 10$.

$S^0 = \{(0,0)\}$.

$S_1^0 = \{(6,2)\}$.

$\Rightarrow S^1 = S^0 \cup S_1^0 = \{(0,0), (6,2)\}$.

$S_1^1 = \{(0+3, 0+2), (6+3, 2+2)\} = \{(3,2), (9,4)\}$

$\Rightarrow S^2 = S^1 \cup S_1^1 = \{(0,0), (6,2), (3,2), (9,4)\}$

$= \{(0,0), (3,2), (6,2), (9,4)\}$

$S^2 = \{(0,0), (6,2), (9,4)\}$

$S_1^2 = \{(0+5, 0+6), (6+5, 2+6), (9+5, 4+6)\}$

$= \{(5,6), (11,8), (14,10)\}$

$\Rightarrow S^3 = S^2 \cup S_1^2 = \{(0,0), (6,2), (9,4), (5,6), (11,8), (14,10)\}$

$= \{(0,0), (5,6), (6,2), (9,4), (11,8), (14,10)\}$

$S^3 = \{(0,0), (6,2), (9,4), (11,8), (14,10)\}$

$S_1^3 = \{(6+4, 0+5), (6+4, 2+5), (9+4, 4+5), (11+4, 8+5), (14+4, 10+5)\}$

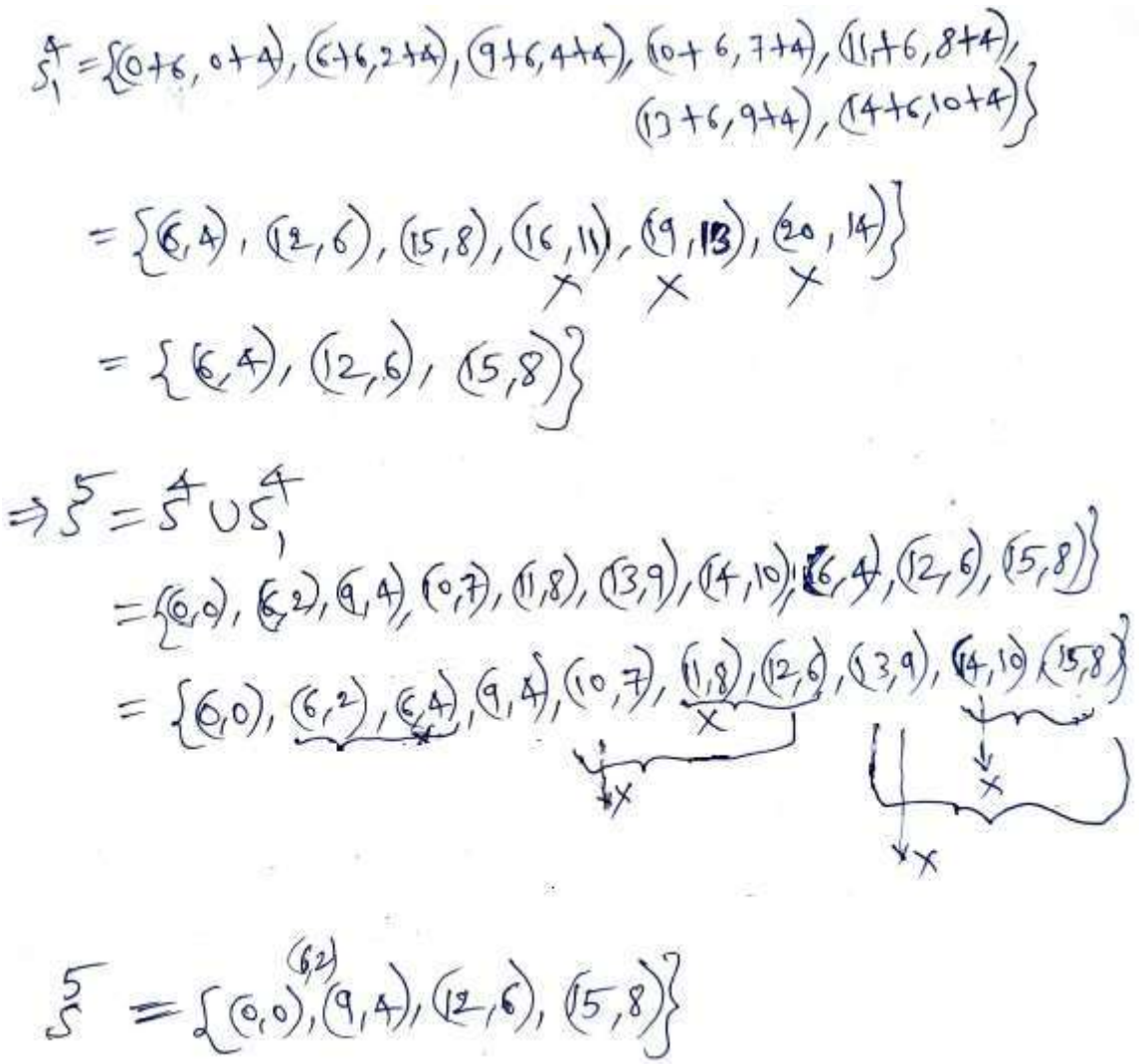
$= \{(4,5), (10,7), (13,9), (15,13), (18,15)\}$

$= \{(4,5), (6,7), (13,9),\}$

$\Rightarrow S^4 = S^3 \cup S_1^3 = \{(0,0), (6,2), (9,4), (11,8), (14,10), (4,5), (10,7), (13,9)\}$

$= \{(0,0), (4,5), (6,2), (9,4), (6,7), (11,8), (13,9), (14,10)\}$

$S^4 = \{(0,0), (6,2), (9,4), (6,7), (11,8), (13,9), (14,10)\}$



Tracing out the value of x_i:

→The last tuple in s^5 is (15,8) $\notin s^4$. So, x_5 = 1.

→The last tuple (15,8) of s^5came from a tuple $(15 - p_5, 8 - w_5)$ = (15-6, 8-4) = (9, 4) belonging to s^4.

→The tuple (9, 4) of s^4, is also present in s^3. So, $x_4 = 0$.

→The tuple (9, 4) of s^3, is also present in s^2. So, $x_3 = 0$.

→The tuple (9, 4) of s^2, is not present in s^1. So, $x_2 = 1$.

→The tuple (9, 4) of s^2 came from a tuple (9-p_2, 4-w_2) = (9-3, 4-2) = (6,2) belonging to s^1.

→ The tuple (6, 2) of s^1, is not present in s^0. So, $x_1 = 1$.

So, the optimal solution is: $(x_1, x_2, x_3, x_4, x_5)$ = (1,1,0,0,1).

How the dynamic-programming method works? (Not required for theory exam)
The dynamic-programming method works as follows.
Having observed that a naive recursive solution is inefficient because it solves the same subproblems repeatedly, we arrange for each subproblem to be solved only *once*, saving its solution. If we need to refer to this subproblem's solution again later, we can just look it up, rather than recompute it. Dynamic programming thus uses additional memory to save computation time; it serves an example of a ***time-memory trade-off***. The savings may be dramatic: an exponential-time solution may be transformed into a polynomial-time solution.

A dynamic-programming approach runs in polynomial time when the number of *distinct* subproblems involved is polynomial in the input size and we can solve each such subproblem in polynomial time.

There are usually two equivalent ways to implement a dynamic-programming approach.

1. Top-down with memoization:

In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table). The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the value in the usual manner and stores the result in the table.

2. Bottom-up method:

This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems. We sort the subproblems by size and solve them in size order, smallest first. When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions. We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

Solution to 0/1 Knapsack problem using Top-down Dynamic Programming approach with Memoization:

→Let us consider the backward recursive equation.

$$f(1, y) = \begin{cases} p_1, & if\ y \geq w_1 \\ 0, & if\ y < w_1 \end{cases}$$

$$f(i, y) = \begin{cases} max\ (f(i-1, y),\ (p_i + , f(i-1, y-w_i))), if\ y \geq w_i \\ f(i-1,\ y) \qquad , if\ \ y < w_i \end{cases}$$

→Let us rewrite the above backward recursive equation as follows:
$f(0, y) = 0$ for $y \geq 0$ and $f(i, 0) = 0$ for $i \geq 0$

$$f(i, y) = \begin{cases} max\ (f(i-1, y),\ (p_i + , f(i-1, y-w_i))), if\ y \geq w_i \\ f(i-1,\ y) \qquad , if\ \ y < w_i \end{cases}$$

→Initially this function is invoked as *f(n, M)*. This problem's size is *n* (i.e., the number of objects on which decisions have to be taken). Next, it calls the subproblem *f(n-1, y)* whose size is *n-1*, and so on. This recursion is repeated until a problem of smallest size i.e., *f(1, y)* is called.

Algorithm f(i, y)
{
// Let T[0:n, 0:M] be a global two dimensional array whose elements are
// initialized with -1 except for row 0 and column 0 which are initialized with 0's.
```
    if (T[i, y] <0) then      // if f(i, y) has not been computed previously
    {
        if(w[i]>y) then
        {
            T[i, y] := f(i-1, y);
        }
        else
        {
            T[i, y] := max(f(i-1, y), p[i] + f(i-1, y-w[i]));
        }
    }
    return T[i, y];
}
```
→Initially this function is invoked as *f(n, M)*.

What is the time and space complexity of the above solution?

Since our memoization array *T[0:n, 0:M]* stores the results for all the subproblems,

we can conclude that we will not have more than *(n+1)*(M+1)* subproblems (where '*n*' is the number of items and '*M*' is the knapsack capacity). This means that the time complexity will be *O(n*M)*.
The above algorithm will be using *O(n*M)* space for the memoization array *T*. Other than that, we will use *O(n)* space for the recursion call-stack. So, the total space complexity will be *O(n*M+n)*, which is asymptotically equivalent to *O(n*M)*.

Tracing back values of x[i]:

```
for i:=n to 1 step -1 do
{
  if T[i, y]=T[i-1, y] then
        x[i]:=0;
  else
  {
       x[i]:=1;
       y:=y-w[i];
   }
}
```

Example: consider the case n=5; M=10; $(w_1, w_2, w_3, w_4, w_5)$ = (2,2,6,5,4) ; $(p_1, p_2, p_3, p_4, p_5)$ = (6,3,5,4,6)
f(5,10) = max(f(4,10), 6+f(4,6))
f(4,10) = max(f(3,10), 4+f(3,5))
f(3,10) = max(f(2,10), 5+f(2,4))
f(2,10) = max(f(1,10), 3+f(1,8))

f(1,10) = max(f(0,10), 6+f(0,8)) = max(0, 6+0) = 6
f(1,8) = max(f(0,8), 6+f(0,6)) = max(0, 6+0) = 6
Now, f(2,10) = max(6, 3+6) = 9
f(2,4) = max(f(1,4), 3+f(1,2)
f(1,4) = max(f(0,4), 6+f(0,2)) = max(0, 6+0) = 6
f(1,2) = max(f(0,2), 6+f(0,0)) = max(0, 6+0) = 6
Now, f(2,4) = max(6,3+6) = 9
Now, f(3,10) = max(6, 5+6) = 11
f(3,5) = f(2,5)
f(2,5) = max(f(1,5), 3+f(1,3))
f(1,5) = max(f(0,5), 6+f(0,3)) = max(0, 6+0) = 6

f(1,3) = max(f(0,3), 6+f(0,1)) = max(0, 6+0) = 6
Now, f(2,5) = max(6, 3+6) = 9
So, f(3,5) = 9
Now, f(4,10) = max(11, 4+9) = 13
f(4,6) = max(f(3,6), 4+f(3,1))
f(3,6) = max(f(2,6), 5+f(2,0))
f(2,6) = max(f(1,6), 3+f(1,4))
f(1,6) = max(f(0,6), 6+f(0,4)) = max(0, 6+0) = 6
f(1,4) = 6 (Obtained from the table)
Now, f(2,6) = max(6, 3+6) = 9.
f(2,0) = 0
Now, f(3,6) = max(9, 5+0) = 9.
f(3,1) = f(2,1) = f(1,1) = 0
f(4,6) = max(9, 4+0) = 9
Now, f(5,10) = max(13, 6+9) =15.

Column indices (y values)

Row indices (i values)	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	-1(0)	-1(6)	-1(6)	-1(6)	-1(6)	-1(6)	-1	-1(6)	-1	-1(6)
2	0	-1(0)	-1	-1	-1(9)	-1(9)	-1(9)	-1	-1	-1	-1(9)
3	0	-1(0)	-1	-1	-1	-1(9)	-1(9)	-1	-1	-1	-1(14)
4	0	-1	-1	-1	-1	-1	-1(9)	-1	-1	-1	-1(14)
5	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1(15)

The optimal solution is: (x1, x2, x3, x4, x5) = (1, 1, 0, 0, 1).

Solution to 0/1 Knapsack problem using Bottom-up Dynamic Programming approach:

→Let us consider backward recursive equation as follows:
f(0, y) = 0 for y≥0 and f(i, 0) = 0 for i≥0

$$f(i, y) = \begin{cases} max\ (f(i-1, y),\ (p_i + ,f(i-1, y-w_i))), if\ y \geq w_i \\ f(i-1,\ y) \qquad\quad , if\ \ y < w_i \end{cases}$$

Step-01:

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

Column indices (y values)

Row indices (i values)

	0	1	2	3	…	M
0	0	0	0	0	…	0
1	0					
2	0					
3	0					
…	…					
N	0					

Step-02:
Start filling the table row wise top to bottom from left to right.
Use the following formula:

$T[i, y] = max\ \{ T[i-1, y], p_i + T[i-1, y-w_i] \}, if\ y \geq w_i$

$= T[i-1, y], if\ \ y < w_i$

Here, *T[i , y]* = maximum profit earned by taking decisions on items *1* to *i* with remaining capacity *y*.

- This step leads to completely filling the table.
- Then, value of the last cell (i.e., intersection of last row and last column) represents the maximum possible profit that can be earned.

Step-03:

To identify the items that must be put into the knapsack to obtain that maximum profit (that means to trace back the values of *xi*),

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

Algorithm:

```
Algorithm KnapSack(n, M)
{
// Let T[0:n, 0:M] be a global two dimensional array whose elements in row 0 and
// column 0 are initialized with 0's.
        for i:=1 to n do
        {
                for y:=1 to M do
                {

                        if(w[i]>y) then
                        {
                                T[i, y] := T[i-1, y];
                        }
                        else
                        {
                                T[i, y] := max(T[i-1, y], p[i] + T[i-1, y-w[i]]);
                        }
                }
        }

        return T[n, M];
}
```

→This function is invoked as KnapSack(n, M).

Time and Space Complexity:

Each entry of the table requires constant time $\theta(1)$ for its computation.

It takes $\theta(nM)$ time to fill $(n+1)(M+1)$ table entries. This means that the time complexity will be $O(n*M)$. Even though it appears to be polynomial time but actually it is called *pseudo polynomial time* because when $M \geq 2^n$, the time complexity is actually exponential but not polynomial.

The space complexity is $\theta(nM)$.

Tracing back values of x[i]:

```
for i:=n to 1 step -1 do
{
  if T[i, y]=T[i-1, y] then
        x[i]:=0;
  else
   {
       x[i]:=1;
       y:=y-w[i];
    }
}
```

Example: consider the case n=5; M=10; ; $(w_1, w_2, w_3, w_4, w_5)$ = (2,2,6,5,4) ; $(p_1, p_2, p_3, p_4, p_5)$ = (6,3,5,4,6)

Column indices (y values)

Row indices (i values)		0	1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0	0	0
w1=2, p1=6	1	0	0	6	6	6	6	6	6	6	6	6
w2=2, p2=3	2	0	0	6	6	9	9	9	9	9	9	9
w3=6, p3=5	3	0	0	6	6	9	9	9	9	11	11	14
w4=5, p4=4	4	0	0	6	6	6	9	9	10	11	13	14
W5=4, p5=6	5	0	0	6	6	6	9	12	12	12	15	15

The optimal solution is: (x1, x2, x3, x4, x5) = (1, 1, 0, 0, 1).

MATRIX CHAIN MULTIPLICATION:

→ Given a sequence of matrices we need to find the most efficient way to multiply those matrices together.

→ Since the matrix multiplication is associative, we can multiply a chain of matrices in several ways.

→ Here, we are actually not interested in performing the multiplication, but in determining in which order the multiplication has to be performed.

→ Let A be an $m \times n$ matrix and B be an $n \times p$ matrix.

→ The number of scalar multiplications needed to perform A*B is m*n*p. We will use this number as a measure of the time (or, cost) needed to multiply two matrices.

→ Suppose we have to compute the matrix product $M_1*M_2*\ldots*M_n$, where M_i has dimensions $r_i \times r_{i+1}$, $1 \le i \le n$.

$M_1 : r_1 \times r_2$

$M_2 : r_2 \times r_3$

.

.

.

$M_n : r_n \times r_{n+1}$

→ We have to determine the order of multiplication that minimizes the total number of scalar multiplications required.

→ Consider the case n=4. The matrix product $M_1*M_2*M_3*M_4$ may be computed in any of the following 5 ways.

1. $M_1*((M_2*M_3)*M_4)$
2. $M_1*(M_2*(M_3*M_4))$
3. $(M_1*M_2)*(M_3*M_4)$
4. $((M_1*M_2)*M_3)*M_4$
5. $(M_1*(M_2*M_3))*M_4$

Example:
Consider three matrices A_{2x3}, B_{3x4}, C_{4x5}.
The product A*B*C can be computed in two ways: (AB)C and A(BC).
The cost of performing (AB)C is: 2*3*4+2*4*5 = 64.
The cost of performing A(BC) is: 3*4*5+2*3*5 = 90.
So, the optimal (i.e., best) order of multiplication is (AB)C.

(OR)

The best way of parenthesizing the given matrix chain multiplication ABC is, (AB)C.

→The number of different ways in which the product of n matrices may be computed, increases exponentially with n, that is $\Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$. As a result, the brute force method of evaluating all the multiplication schemes and select the best one is not practical for large n.

DYNAMIC PROGRAMING FORMULATION:

→ We can use dynamic programming to determine an optimal sequence of pair wise matrix multiplications. The resulting algorithm runs in only $O(n^3)$ time.

→ Let $M_{i,j}$ denote the result of the product chain $M_i * M_{i+1} * \ldots * M_j$, i<j.

Ex: $M_1*M_2*M_3 = M_{1,3}$.

Thus $M_{i,i} = M_i$, $1 \leq i \leq n$.

Clearly $M_{i,j}$ has dimensions: $r_i \times r_{j+1}$.

→ Let c(i, j) be the cost of computing $M_{i,j}$ in an optimal way.Thus c(i, i) = 0, $1 \leq i \leq n$.

→Now in order to determine how to perform the multiplication $M_{i,j}$ optimally, we need to make decisions. What we want to do is to break the problem into sub problems of similar structure.

→ In parenthesizing the matrix multiplication, we can consider the highest level (i.e., last level) of parenthesization. At this level, we simply multiply two matrices together. That is, for any k, (i≤k<j),

$M_{i,j} = M_{i,k} * M_{k+1,j}$

→ Thus, the problem of determining the optimal sequence of multiplications is

broken up into two questions:
1. How do we decide where to split the chain (i.e., what is k)?
2. How do we parenthesize the sub chains $(M_i * M_{i+1} * \ldots * M_k)$ and $(M_{k+1} * \ldots * M_j)$?

→ In order to compute $M_{i,j}$ optimally, $M_{i,k}$ and $M_{k+1,j}$ should also becomputed optimally. Hence, the principle of optimality holds.

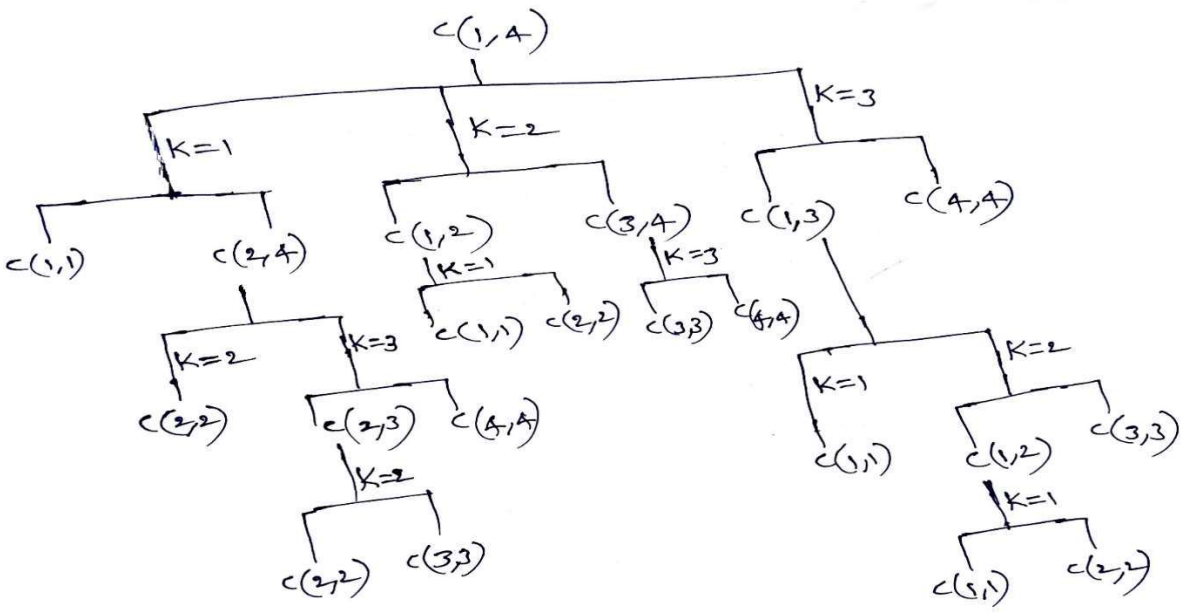
→ The cost of computing $M_{i,k}$ (i.e., $M_i * M_{i+1} * \ldots * M_k$) optimally is $c(i, k)$. The cost of computing $M_{k+1,j}$ (i.e., $M_{k+1} * \ldots * M_j$) optimally is $c(k+1, j)$.
→Since $M_{i,k}$ has dimensions $r_i \times r_{k+1}$ and $M_{k+1,j}$ has dimensions $r_{k+1} \times r_{j+1}$, the cost of multiplying the two matrices $M_{i,k}$ and $M_{k+1,j}$ is, $r_i * r_{k+1} * r_{j+1}$.
→ So, the cost of computing $M_{i,j}$ optimally is,
$c(i, j) = c(i, k) + c(k+1, j) + r_i * r_{k+1} * r_{j+1}$.
→ But, in the above equation, k (which is the splitting point of matrix product chain) can take different values based on the inequality condition $i \leq k < j$.

→ This suggests the following recurrence equation for computing $c(i, j)$:

$$c(i, j) = \begin{cases} 0, & if\ i = j \\ min_{i \leq k < j}\{c(i,k) + c(k+1,j) + r_i * r_{k+1} * r_{j+1}\}, & if\ i < j \end{cases}$$

→ The above recurrence equation for c may be solved recursively. The k value which results in $c(i, j)$ is denoted by $kay(i, j)$.
→ $c(1, n)$ is the cost of the optimal way to compute the matrix product chain $M_{1,n}$. And $kay(1, n)$ defines the last product to be done or where the splitting is done.
→ The remaining products can be determined by using kay values.

The tree of recursive calls of $c(\ )$ function for the matrix product chain $M_1 * M_2 * M_3 * M_4$:



SOLUTION FOR MATRIX CHAIN MULTIPLICATION:

→ The dynamic programing recurrence equation for *c* may be solved by computing each *c(i, j)* and *kay(i, j)* values exactly once in the order *j-i =1, 2, 3, .., n-1.*

Example: Apply dynamic programing technique for finding an optimal order of multiplying the five matrices with *r =(10, 5,1,10,2,10).*

Solution:

Initially, we have $c_{ii}=0$ and $kay_{ii}=0,\ 1\le i \le 5.$

Using the equation:

$$c(i,j)=\begin{cases}0, & if\ i=j\\ min_{i\le k<j}\{c(i,k)+c(k+1,j)+r_i * r_{k+1} * r_{j+1}\}, & if\ i<j\end{cases}$$

For $j-i=1$:

$c(1,2)=min_{1\le k<2}\{c(1,1)+c(2,2)+r_1 * r_2 * r_3\}$

$=0+0+10*5*1=50$

$kay(1,2)=1$

$c(2,3)=min_{2\le k<3}\{c(2,2)+c(3,3)+r_2 * r_3 * r_4\}$

$=0+0+5*1*10=50$

$kay(2,3)=1$

$c(3,4)=min_{3\le k<4}\{c(3,3)+c(4,4)+r_3 * r_4 * r_5\}$

$=0+0+1*10*2=20$
$kay(3,4)=3$

$c(4,5)=min_{4\leq k<5}\{c(4,4)+c(5,5)+r_4*r_5*r_6\}$
$=0+0+10*2*10=200$
$kay(4,5)=4$

For j – i = 2:

$c(1,3)=min_{1\leq k<3}\{c(1,1)+c(2,3)+r_1*r_2*r_4;\ c(1,2)+c(3,3)+r_1*r_3*r_4\}$
$=min\{0+50+10*5*10,\ 50+0+10*1*10\}$
$=min\{550,150\}$
$= 150$
$kay(1,3)=2$
$c(2,4)=min_{2\leq k<4}\{c(2,2)+c(3,4)+r_2*r_3*r_5;\ c(2,3)+c(4,4)+r_2*r_4*r_5\}$

$=min\{0+20+5*1*2,\ 50+0+5*10*2\}$
$=min\{30,\ 150\}$
$= 30\ kay(2,4)=2$

$c(3,5)=min_{3\leq k<5}\{c(3,3)+c(4,5)+r_3*r_4*r_6;\ c(3,4)+c(5,5)+r_3*r_5*r_6\}$
$=min\{0+200+1*10*10,\ 20+0+1*2*10\}$
$=min\{300,\ 40\}$
$= 40\ kay(3,5)=4$

For j – i = 3:

$c(1,4)=min_{1\leq k<4}\{c(1,1)+c(2,4)+r_1*r_2*r_5;\ c(1,2)+c(3,4)+r_1*r_3*r_5;\ c(1,3)+c(4,4)+r_1*r_4*r_5\}$
$=min\{0+30+10*5*2,\ 50+20+10*1*2,\ 150+0+10*2*2\}$
$=min\{130,\ 90,\ 190\}$
$= 90\ kay(1,4)=2$

$c(2,5)=min_{2\leq k<5}\{c(2,2)+c(3,5)+r_2*r_3*r_6;\ c(2,3)+c(4,5)+$

$r_2 * r_4 * r_6$; $c(2,4) + c(5,5) + r_2 * r_5 * r_6$}
=min{0+40+5*1*10, 50+200+5*10*10, 30+0+5*2*10}
=min{90,750,130}
= 90 kay(2,5)=2

$c(1,5)=min_{1 \le k < 5}$\{$c(1,1) + c(2,5) + r_1 * r_2 * r_6$; $c(1,2) + c(3,5) + r_1 * r_3 * r_6$; $c(1,3) + c(4,5) + r_1 * r_4 * r_6$; $c(1,4) + c(5,5) + r_1 * r_5 * r_6$\}
=min{0+90+10*5*10, 50+40+10*1*10, 150+200+10*10*10, 90+0+10*2*10}
=min{590, 190, 1350, 290}
= 190

kay(1,5)=2

j-i ⇩ \ i ⇨	1	2	3	4	5
0	$c_{11} = 0$ $kay_{11} = 0$	$c_{22} = 0$ $kay_{22} = 0$	$c_{33} = 0$ $kay_{33} = 0$	$c_{44} = 0$ $kay_{44} = 0$	$c_{55} = 0$ $kay_{55} = 0$
1	$c_{12} = 50$ $kay_{12} = 1$	$c_{23} = 50$ $kay_{23} = 2$	$c_{34} = 20$ $kay_{34} = 3$	$c_{45} = 200$ $kay_{45} = 4$	
2	$c_{13} = 150$ $kay_{13} = 2$	$c_{24} = 30$ $kay_{24} = 2$	$c_{35} = 40$ $kay_{35} = 4$		
3	$c_{14} = 90$ $kay_{14} = 2$	$c_{25} = 90$ $kay_{25} = 2$			
4	$c_{15} = 190$ $kay_{15} = 2$				

→ If k is the splitting point, $M_{i,j} = M_{i,k} * M_{k+1,j}$.
→ From the above table, the optimal multiplication sequence has cost 190. The sequence can be determined by examining *kay(1,5)*, which is equal to *2*.
→ So, $M_{1,5} = M_{1,2} * M_{3,5} = (M_1 * M_2) * (M_3 * M_4 * M_5)$
Since $kay(3,5) = 4$; $M_{3,5} = M_{3,4} * M_{5,5} = (M_3 * M_4) * M_5$
So, the optimal order of matrix multiplication is:
$M_{1,5} = M_1 * M_2 * M_3 * M_4 * M_5 = (M_1 * M_2) * ((M_3 * M_4) * M_5)$

Algorithm:

```
Algorithm MATRIX-CHAIN-ORDER(r)
{
    n := length(r) -1;  // n denotes number of matrices
    for i:= 1 to n do
        c[i, i]:= 0;
    for l := 2 to n   do      // l is the chain length
    {
        for i := 1 to n-l+1 do  // n-l+1 gives number of cells in the current row
        {
            j := i+l-1;
            c[i, j] := ∞;
            for k := i to j-1 do
            {
                q := c[i, k] + c[k+1, j] + r_i*r_{k+1}*r_{j+1};
                if q < c[i, j] then
                {
                    c[i, j]:= q;
                    kay[i, j]:= k;
                }
            }
        }
    }
    return c and kay;
}
```

Constructing an optimal solution

Although MATRIX-CHAIN-ORDER determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it does not directly show how to multiply the matrices.

The table *kay[1:n, 2:n]* gives us the information we need to do so. Each entry *kay[i,j]* records a value of k such that an optimal parenthesization of $M_i * M_{i+1} * \ldots * M_j$ splits the product between M_k and M_{k+1}.

Thus, we know that the final matrix multiplication in computing $M_{1,n}$ optimally is $M_{1,k} * M_{k+1,n} = M_{1,kay(i,j)} * M_{kay(i,j)+1,n}$ We can determine the earlier matrix multiplications recursively, since *kay[1, kay[1,n]]* determines the last matrix multiplication when computing $M_{1,kay(i,j)}$ and *kay[kay[1,n]+1, n]* determines the last matrix multiplication when computing $M_{kay(i,j)+1,n}$. The following recursive

procedure prints an optimal parenthesization of $M_i*M_{i+1}*...*M_j$, given the *kay* table computed by MATRIX-CHAIN-ORDER and the indices *i* and *j*. The initial call PRINT-OPTIMAL-PARENS*(kay, i, n)* prints an optimal parenthesization of $M_1*M_{i+1}*...*M_n$.

```
Algorithm PRINT-OPTIMAL-PARENS(kay, i, j)
{
        if (i = j) then
                print "M"i;
        else
        {
                print "(";
                PRINT-OPTIMAL-PARENS(kay, i, kay[i, j]);
                PRINT-OPTIMAL-PARENS(kay, kay[i, j]+1, j ];
                print ")"
        }
}
```

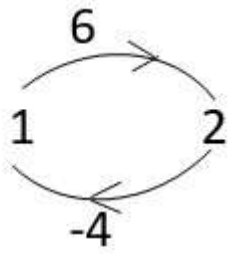
All-Pairs Shortest Paths problem: -

Given a graph *G*, we have to find a shortest path between every pair of vertices. That is, for every pair of vertices *(i, j)*, we have to find a shortest from *i* to *j*.

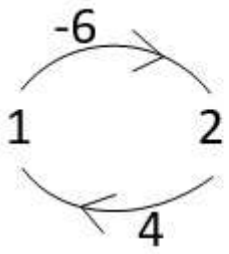
Let *G = (V, E)* be a weighted graph with *n* vertices. Let *c* be the cost adjacency matrix for G such that $c[i, i]=0,\ 1\le i\le n$ and $c[i, j]=\infty$ if $i \ne j$ and $<i, j> \notin E(G)$.

When no edge has a negative length, the All-Pairs Shortest Paths problem may be solved by applying Dijkstra's greedy Single-Source Shortest Paths algorithm *n* times, once with each of the *n* vertices as the source vertex. This process results in $O(n^3)$ solution to the All-Pairs problem.

We will develop a dynamic programming solution called Floyd's algorithm which also runs in $O(n^3)$ time but works even when the graph has negative length edges (provided there are no negative length cycles).



Graph with positive cycle
Cycle length = 2
Floyd's algorithm works



Graph with negative cycle
Cycle length = -2
Floyd's algorithm doesn't work

We need to determine a matrix $A[1:n,1:n]$ such that $A[i, j]$ is the length of the shortest path from vertex i to vertex j.

The matrix A is initialized as $A^0[i, j] = c[i, j],\ 1 \le i \le n,\ 1 \le j \le n$. The algorithm makes n passes over A. In each pass, A is transformed. Let $A^1, A^2, A^3, \ldots, A^n$ be the transformations of A on the n passes.

Let $A^k[i, j]$ denote the length of the shortest path from i to j that has no intermediate vertex larger than k. That means, the path possibly passes through the vertices $\{1, 2, 3, \ldots, k\}$, but not through the vertices $\{k+1, k+2, \ldots, n\}$.

So, $A^n[i, j]$ gives the length of the shortest path from i to j because all intermediate vertices $\{1, 2, 3, \ldots, n\}$ are considered.

How to determine $A^k[i, j]$ for any $k \ge 1$?

A shortest path from i to j going through no vertex higher than k may or mayn't go through k.

If the path goes through k, then $A^k[i, j]=A^{k-1}[i, k]+A^{k-1}[k, j]$, following the principle of optimality.

If the path doesn't go through k, then no intermediate vertex has index greater than $(k-1)$.

Hence $A^k[i, j]=A^{k-1}[i, j]$.

By combining both cases, we get

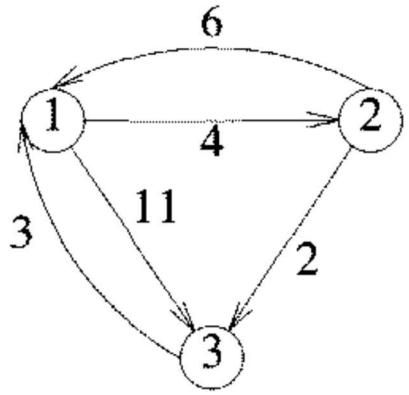
$A^k[i, j]=min\{ A^{k-1}[i, j], A^{k-1}[i, k]+A^{k-1}[k, j]\}, k\geq 1.$

This recurrence can be solved for A^n by first computing A^1, then A^2, then A^3, and so on. In the algorithm, the same matrix A is transformed over n passes and so the superscript on A is not needed. The k^{th} pass explores whether the vertex k lies on an optimal path from i to j, for all i and j. We assume that the shortest path *(i, j)* contains no cycles.

```
Algorithm Allpaths(c, A, n)
{
        //c[1:n,1:n] is the cost adjacency matrix of a graph with n vertices.
        //A[i, j] is length of a shortest path from vertex i to vertex j.
        //c[i, i]=0,  for 1≤i≤n.
        for i:=1 to n do
        for j:=1 to n do
                        A[i, j]=c[i, j];  //copy c into A
        for k:=1 to n do
        for i=1 to n do
                        for j=1 to n do
                                A[i, j]:=min(A[i, j], A[i, k]+A[k, j]);
}
```

Time complexity, $T(n)=O(n^3)$.

EXAMPLE: - Find out shortest paths between all pairs of vertices in the following digraph.



Step-1: $A^0 \leftarrow c$

0	4	11
6	0	2
3	∞	0

Step-2: Using, $A^1[i, j] = \min\{A^0[i, j], A^0[i, 1]+A^0[1, j]\}$

A^1

0	4	11
6	0	2
3	7	0

Step-3: Using, $A^2[i, j] = \min\{A^1[i, j], A^1[i, 2]+A^1[2, j]\}$

A^2

0	4	6
6	0	2
3	7	0

Step-4: Using, $A^3[i, j] = \min\{A^2[i, j], A^2[i, 3]+A^2[3, j]\}$

A^3

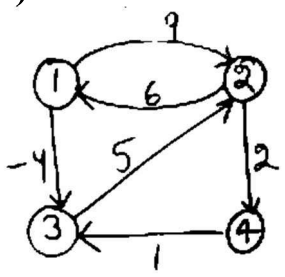
0	4	6
5	0	2
3	7	0

All-Pairs Shortest Paths:

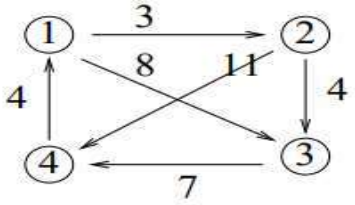
$1 \rightarrow 2$	$2 \rightarrow 3$	$3 \rightarrow 1$
$1 \rightarrow 2 \rightarrow 3$	$2 \rightarrow 3 \rightarrow 1$	$3 \rightarrow 1 \rightarrow 2$

Exercises:

(1)



(2)



The Traveling Salesperson problem: -

→Given a set of cities and distances between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

→Let the cities be represented by vertices in a graph and the distances between them be represented by weights on edges in the graph.

→Let $G(V, E)$ be the directed graph with n vertices.
→Let graph G be represented by cost adjacency matrix $C[1:n, 1:n]$.
→For simplicity, $C[i, j]$ is denoted by C_{ij}. If $<i, j> \notin E(G)$, $C_{ij}=\infty$; otherwise $C_{ij} \geq 0$.

→A tour of G is a directed cycle that includes every vertex in V. The cost of a tour is the sum of the costs of the edges on the tour.
→The travelling salesperson problem aims at finding an optimal tour (i.e., a tour of minimum cost).

→Let the given set of vertices be $\{1, 2, ..., n\}$. In our discussion, we shall consider a tour be a simple path that starts at vertex 1 and terminates at 1.

→Every possible tour can be viewed as consisting of an edge $<1, k>$ for some $k \in V-\{1\}$ and a path from vertex k to vertex 1.
The path from vertex k to vertex 1 goes through each vertex in the set $V-\{1, k\}$ exactly once.

→Suppose the tour consisting of the edge $<1, k>$ (for some $k \in V-\{1\}$) followed by a path from k to 1 is an optimal tour. Then the path from k to 1 should also be optimal. Thus, the principle of optimality holds.

→Let $g(i, S)$ denote the length of the shortest path starting at vertex i, going through all vertices in set S, and terminating at vertex 1. Thus $g(1, V-\{1\})$ gives the length of an optimal tour.
→If the optimal tour consists of the edge $<1, k>$, then from the principle

of optimality, it follows that $g(1, V-\{1\}) = C_{1k}+g(k, V-\{1,k\})$.
→But we don't know for which value of k the tour will be optimal. We know that k take can any value from the set $\{2, 3, \ldots, n\}$.
→The RHS of the above recurrence can be evaluated for each value of k and the minimum of those results should be assigned to $g(1, V-\{1\})$ resulting in the following recurrence equation:

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

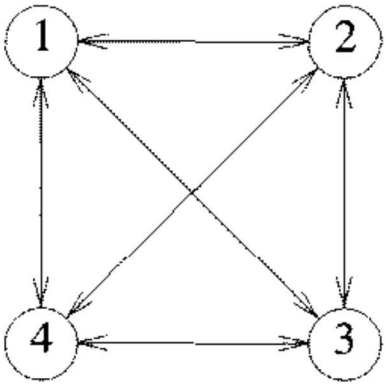
→Generalizing equation *(1)* for i not in S, we obtain

$$g(i, S)=\min_{\forall j\in S} \{C_{ij}+g(j, S-\{j\})\} \text{ ------ (2)}$$

→Equation *(1)* can be solved for $g(1, V-\{1\})$ if we know $g(k, V-\{1,k\})$ for all choices of k which can be obtained by using equation *(2)*.
→Clearly, $g(i, \emptyset)=C_{i1}, 1\le i\le n$.

→The optimal tour can be found by noting the vertex which resulted in minimum cost at each stage.
An algorithm that proceeds to find an optimal tour by making use of *(1)* and *(2)* will require
$O(n^2 2^n)$ time.

<u>EXAMPLE</u>: - Consider the directed graph below and its edge lengths given by cost adjacency matrix.



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

Find optimal sales person tour from vertex1.

Answer:-

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

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

$g(3,\{4\})=C_{34}+g(4,\emptyset)=C_{34}+C_{41}$

$=12+8=20.$

$g(4,\{3\})=C_{43}+g(3,\emptyset)=C_{43}+C_{31}$

$=9+6=15.$

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

$=25.$

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

$g(2,\{4\})=C_{24}+g(4,\emptyset)=C_{24}+C_{41}$

$=10+8=18.$

$g(4,\{2\})=C_{42}+g(2,\emptyset)=C_{42}+C_{21}$

$=8+5=13.$

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

$=\min\{31, 25\}$

$=25.$

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

$g(2,\{3\})=C_{23}+g(3,\emptyset)=C_{23}+C_{31}$

$=9+6=15.$

$g(3,\{2\})=C_{32}+g(2,\emptyset)=C_{32}+C_{21}$

$=13+5=28.$

$g(4,\{2,3\})=\min\{8+15,9+28\}$

$=\min\{23,37\}$

$=23.$

$g(1,\{2,3,4\})=\min\{10+25,15+25,20+23\}$

$=\min\{35,40,43\}$

$=35.$

$g(1,\{2,3,4\})=35.$

Construction of the optimal tour step by step:-

$g(1,\{2,3,4\})=C_{12}+g(2,\{3,4\})$ ➔ $1\rightarrow2$

$g(2,\{3,4\})=C_{24}+g(4,\{3\})$ ➔ $1\rightarrow2\rightarrow4$

$g(4,\{3\})=C_{43}+g(\{3,\emptyset\})$ ➔ $1\rightarrow2\rightarrow4\rightarrow3\rightarrow1$

So, the optimal tour is:- $1\rightarrow2\rightarrow4\rightarrow3\rightarrow1$.