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:

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

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 $x_1, x_2, x_3, \ldots, x$.

→ 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 $i, i+1, \ldots, n$ for which decisions have to be taken.

It follows that

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

and

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

<u>NOTE:</u> 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 take into account that choice 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 $f(1, M)$.

<u>Example 1:</u> 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 $M=116$.

<u>Solution:</u>

$f(1,116) = max\{f(2, 116),\ 20 + f(2, 116 - 100)\}$, 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,

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

$f(3,16) = 15$

$f(3,2) = 0$

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

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

<u>Tracing back the x_i values:</u>

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 $f(1, M) \ne f(2,M)$ 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.

<u>Determining the x_i values for the above example:</u>

$f(1, 116) = 38$

$f(2, 116) = 33$

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.

<u>ALOGORITHM FOR 0/1 KNAPSACK PROBLEM USING FORWARD APPROCH:</u>

```
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 $f(1, M)$.

<u>TIME COMPLEXITY:</u>

→ let $T(n)$ be the time this code takes to solve an instance with n objects.

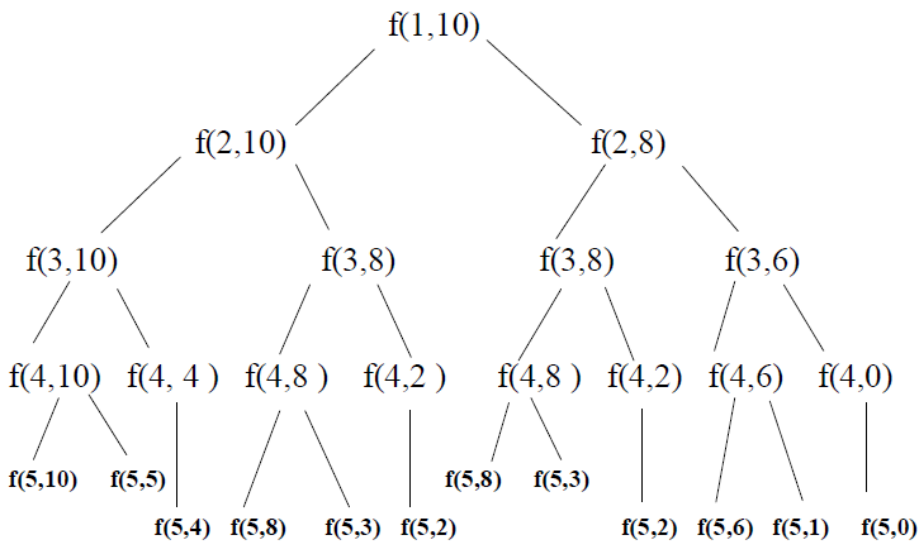
So, $T(n) = \begin{cases} c, & if\ n = 1 \\ 2T(n-1) + c, & if\ n > 1 \end{cases}$

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.

<u>Example:</u> 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.

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

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*.

<u>Example</u>: 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 $M=6$.

<u>Solution:</u>

$f(3,6) = max\{f(2, 6), 5 + f(2, 2)\}$
$f(2,6) = max\{f(1, 6), 2 + f(1, 3)\}$
$f(1,6) = p1 = 1$
$f(1,3) = p1 = 1$
$f(2,6) = max\{1, 2+1\} = 3$
$f(2,2) = f(1,2) = f(1,2) = 1$
Now,
$f(3,6) = max\{3, 5+1\} = 6$
<u>Tracing back values of x_i:</u>
$f(3,6) = 6$
$f(2,6) = 3$
Since $f(3,6) \neq f(2,6)$, $x_3=1$.
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: $(x_1, x_2, x_3) = (1,0,1)$.

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

→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 value 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)

<u>Example:</u> 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 $M = 10$.

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

$S^0_1 = \{(6,2)\}$.

$\Rightarrow S^1 = S^0 \cup S^0_1 = \{(0,0), (6,2)\}$.

$S^1_1 = \{(6+3, 0+2), (6+3, 2+2)\} = \{(9,2), (9,4)\}$

$\Rightarrow S^2 = S^1 \cup S^1_1 = \{(0,0), (6,2), (3,2), (9,4)\}$

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

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

$S^2_1 = \{(6+5, 0+6), (6+5, 2+6), (9+5, 4+6)\}$

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

$\Rightarrow S^3 = S^2 \cup S^2_1 = \{(0,0), (6,2), (9,4), (5,6), (11,8), (14,10)\}$

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

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

$S^3_1 = \{(6+4, 0+5), (6+4, 2+5), (9+4, 4+5), (11+4, 8+5), (14+4, 10+5)\}$

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

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

$\Rightarrow S^4 = S^3 \cup S^3_1 = \{(0,0), (6,2), (9,4), (11,8), (14,10), (4,5), (10,7), (13,9)\}$

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

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

$$S_1^4 = \{(0+6, 0+4), (6+6, 2+4), (9+6, 4+4), (10+6, 7+4), (11+6, 8+4), (13+6, 9+4), (14+6, 10+4)\}$$

$$= \{(6,4), (12,6), (15,8), (16,11), (19,13), (20,14)\}$$

$$= \{(6,4), (12,6), (15,8)\}$$

$$\Rightarrow S^5 = S^4 \cup S_1^4$$

$$= \{(0,0), (6,2), (9,4), (10,7), (11,8), (13,9), (14,10), (6,4), (12,6), (15,8)\}$$

$$= \{(0,0), (6,2), (6,4), (9,4), (10,7), (11,8), (12,6), (13,9), (14,10), (15,8)\}$$

$$S^5 = \{(0,0), (6,2), (9,4), (12,6), (15,8)\}$$

<u>Tracing out the value of x_i:</u>

→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.

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

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

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

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

→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).*

<u>What is the time and space complexity of the above solution?</u>

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)*.

<u>Tracing back values of x[i]:</u>

```
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

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

Step-01:

- Draw a table say '*T*' with *(n+1)* number of rows and *(M+1)* number of columns.
- Fill all the boxes of 0th row and 0th 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:

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

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

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).

<u>Time and Space Complexity:</u>
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 $\theta(nM)$.

<u>Tracing back values of x[i]:</u>

```
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)

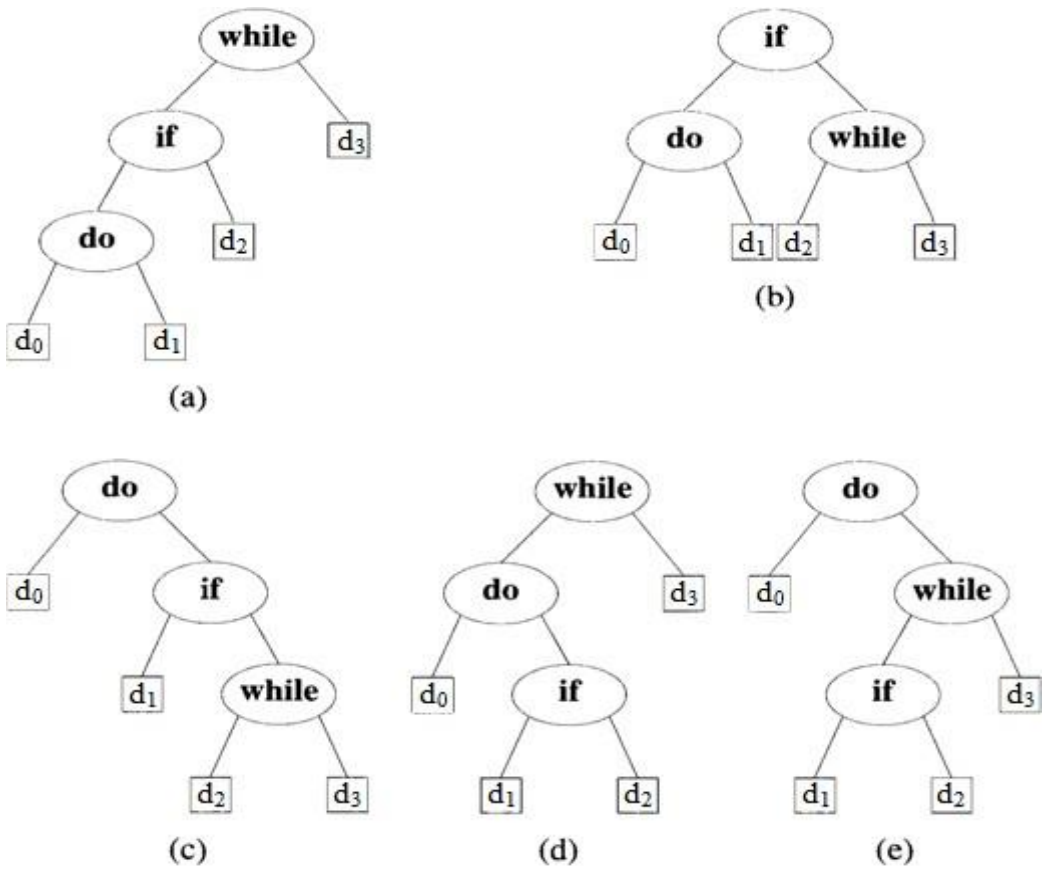
Row indices (i values)		0	1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0	0	0
w1=2, p1=6	1	0	0	6	6	6	6	6	6	6	6	6
w2=2, p2=3	2	0	0	6	6	9	9	9	9	9	9	9
w3=6, p3=5	3	0	0	6	6	9	9	9	9	11	11	14
w4=5, p4=4	4	0	0	6	6	6	9	9	10	11	13	14
W5=4, p5=6	5	0	0	6	6	6	9	12	12	12	15	15

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

OPTIMAL BINARY SEARCH TREE:(OBST)

→ We are given a set of sorted identifiers (or, keys) $\{a_1, a_2, \ldots, a_n\}$ such that $a_1 < a_2 < a_3 < \ldots < a_n$, and probabilities $p_1, p_2, \ldots, p_n$ with which the keys are searched for respectively.
→ The problem is to arrange the keys in a Binary Search Tree in a way that minimizes the expected total search time. Such a BST is called optimal BST (OBST).
→ We assume that there are $n+1$ dummy keys named $d_0, d_1, d_2, \ldots, d_n$ such that d_0 represents all values less than a_1 and d_n represents all values greater than a_n and for $i=1, 2, 3, \ldots, n-1$ the dummy key d_i represents all values between a_i and a_{i+1}.
→ The dummy keys are leaves (or, external nodes) and the data keys are internal nodes in the BST.
→ For each dummy key d_i, we have a search probability q_i.
Since every search is either successful or unsuccessful, the probabilities sum to *1*.
Therefore: $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$.
→ The actual search cost for any key = Number of items examined (or Number of comparisons made).
→ So, for any key a_i, the actual search cost = $level(a_i)$.
The expected search cost of $a_i = level(a_i) * p_i$.
→ For dummy key d_i, the actual search cost = $level(d_i)-1$.
The expected search cost of $d_i = (level(d_i)-1) * q_i$.
→ Now, the expected search cost of a Binary Search Tree *'T'* is,
$\sum_{i=1}^{n}(\text{level}(a_i) * p_i) + \sum_{i=0}^{n}\big((level(d_i) - 1) * q_i\big) \rightarrow$ (1)
=Expected search cost of successful searches + Expected search cost of unsuccessful searches.

Example:
Draw the possible BST's along with dummy keys for the identifier set (a_1, a_2, a_3) = ***(do, if, while)***, with the probabilities $p1= 0.5$, $p2=0.1$, $p3= 0.05$, $q0= 0.15$, $q1= 0.1$, $q2= 0.05$ and $q3=0.05$ and then find out expected search cost of each tree.



EXPECTED SEARCH COST FOR TREE (a):

$$\sum_{i=1}^{3}(\text{level}(a_i) * p_i) + \sum_{i=0}^{3}\big((level(d_i) - 1) * q_i\big)$$

= (3*0.5 + 2*0.1 + 1*0.05) + ((4-1)*0.15 + (4-1)*0.1 + (3-1)*0.05 + (2-1)*0.05)
= 2.65

EXPECTED SEARCH COST FOR TREE (b):

= (2*0.5 + 1*0.1 + 2*0.05) + ((3-1)*0.15 + (3-1)*0.1 + (3-1)*0.05 + (3-1)*0.005)
= 1.9

EXPECTED SEARCH COST FOR TREE (c):
= (1*0.5 + 2*0.1 + 3*0.05) + ((2-1)*0.15 + (3-1)*0.1 + (4-1)*0.05 + (4-1)*0.05)
=1.5

EXPECTED SEARCH COST FOR TREE (d):
= (2*0.5 + 3*0.1 + 1*0.05) + ((3-1)*0.15 + (4-1)*0.1 + (4-1)*0.05 +(2-1)*0.05)

= 2.15

EXPECTED SEARCH COST FOR TREE (e):

= (1*0.5 + 3*0.1 + 2*0.05) + ((2-1)*0.15 + (4-1)*0.1 + (4-1)*0.05 + (3-1)*0.05)
= 1.65

The tree (c) has the minimum cost. So, it is OBST.
<u>NOTE:</u>
OBST may not have smallest height.
OBST may not have highest probability key at root.

<u>APPLYING DYNAMIC PROGRAMING TO SOLVE OBST PROBLEM:</u>

→ To apply dynamic programing to the problem of obtaining an OBST, we need to view the construction of such a tree as the result of a sequence of decisions.

→ The 1st decision to make is, which of the a_i*'s* should be assigned to the rootnode of the BST.

→ If <u>we choose a_k as the root node</u> (for $1 \le k \le n$), then the internal nodes $a_1, a_2, \ldots, a_{k-1}$ and the external nodes $d_0, d_1, d_2, d_3, \ldots, d_{k-1}$, will lie in the left subtree of the root. The remaining nodes, (i.e., $a_{k+1}, \ldots, a_n$ and $d_k, d_{k+1}, \ldots, d_n$), will lie in the right subtree.

→ Denote BST by *'T'*, its left subtree by *'L'*, and its right subtree by *'R'*.

→ Then for any a_i, $i<k$, we have:
$level_T(a_i) = level_L(a_i) + 1$ (*i.e.*, Level of a_i in T = (Level of a_i in L) +1)
For any d_i, i<k, we have:
$level_T(d_i) = level_L(d_i) + 1$
→ For any a_i, $i>k$, we have:
$level_T(a_i) = level_R(a_i) + 1$
For any d_i, i≥k, we have:
$level_T(d_i) = level_R(d_i) + 1$
→ Let *Cost(T)* denote the estimated cost of BST *'T'*.
Let *Cost(L)* denote the estimated cost of left subtree *'L'*. Let *Cost(R)* denote the estimated cost of right subtree *'R'*.

→ We know that,

$$Cost(T) = \sum_{i=1}^{n} (level_T(a_i) * p_i) + \sum_{i=0}^{n} \big((level_T(d_i) - 1) * q_i\big)$$

→ If a_k is the root, then

$$Cost(L) = \sum_{i=1}^{k-1} (level_L(a_i) * p_i) + \sum_{i=0}^{k-1} \big((level_L(d_i) - 1) * q_i\big)$$

$$Cost(R) = \sum_{i=k+1}^{n} (level_R(a_i) * p_i) + \sum_{i=k}^{n} \big((level_R(d_i) - 1) * q_i\big)$$

→ Now,

$$Cost(T)=level_T(a_k)*p_k+\sum_{i=1}^{k-1}(level_T(a_i) * p_i)+\sum_{i=k+1}^{n}(level_T(a_i) * p_i)+ \sum_{i=0}^{k-1}\big((level_T(d_i) - 1) * q_i\big)+\sum_{i=k}^{n}\big((level_T(d_i) - 1) * q_i\big)$$

$$=1*p_k+\sum_{i=1}^{k-1}((level_L(a_i) + 1) * p_i)+\sum_{i=k+1}^{n}((level_R(a_i) + 1) * p_i)+\sum_{i=0}^{k-1}(level_L(d_i) * q_i)+\sum_{i=k}^{n}(level_R(d_i) * q_i)$$

$$=p_k+\sum_{i=1}^{k-1}(level_L(a_i) * p_i)+\sum_{i=k+1}^{n}((level_R(a_i)) * p_i)+\sum_{i=0}^{k-1}((level_L(d_i) - 1) * q_i)+\sum_{i=k}^{n}((level_R(d_i) - 1) * q_i)+\sum_{i=1}^{k-1} p_i+\sum_{i=k+1}^{n} p_i+\sum_{i=0}^{k-1} q_i+\sum_{i=k}^{n} q_i$$

$$= p_k+[\sum_{i=1}^{k-1}(level_L(a_i) * p_i)+\sum_{i=0}^{k-1}((level_T(d_i) - 1) * q_i)]+ [\sum_{i=k+1}^{n}((level_R(a_i) + 1) * p_i)+\sum_{i=k}^{n}((level_R(d_i) - 1) * q_i)] +\sum_{i=1}^{k-1} p_i+\sum_{i=0}^{k-1} q_i+\sum_{i=k+1}^{n} p_i + \sum_{i=k}^{n} q_i$$

$Cost(T)= p_k+ Cost(L)+ Cost(R)+ w(0, k-1) + w(k, n)$ → *(2)*

where, $w(i, j)= \sum_{l=i+1}^{j} p_l+\sum_{l=i}^{j} q_l=(p_{i+1}+\ldots+p_j)+(q_i+q_{i+1}+\ldots+q_j)$

→ In equation (2), k can take any value from the set $\{1, 2, \ldots, n\}$. If the tree *'T'* is optimal, then the *Cost(T)* in equation (2) must be minimum over all BST's containing keys $a_1, a_2, \ldots, a_n$ and dummy keys $d_0, d_1, \ldots, d_n$.Hence *Cost(L)* must be minimum over all BST's containing keys $a_1, a_2, \ldots, a_{k-1}$ and dummy keys $d_0, d_1, \ldots, d_{k-1}$. Similarly, Cost(R) must be minimum over all BST's containing keys $a_{k+1}, \ldots, a_n$ and dummy keys $d_k, d_{k+1}, \ldots, d_n$.

→ Let $c(i, j)$ denote the cost of OBST $T_{i,j}$ containing keys $a_{i+1}, a_{i+2}, \ldots, a_j$ and

dummy keys $d_i, d_{i+1}, \ldots, d_j$.

Then for the tree $T_{0,n}$ to be optimal,
We must have
$Cost(L) = c(0, k-1)$ and
$Cost(R) = c(k, n)$
→ So, cost of OBST $T_{0,n}$ is,
$c(0, n) = min_{1 \le k \le n}\{p_k + c(0, k-1) + c(k, n) + w(0, k-1) + w(k, n)\} \rightarrow (3)$

→ We can generalize equation (3) to obtain cost for any OBST $T_{i,j}$,
$c(i, j) = min_{i+1 \le k \le j}\{p_k + c(i, k-1) + c(k, j) + w(i, k-1) + w(k, j)\}$

→ Since $p_k + w(i, k-1) + w(k, j) = w(i, j)$,

$$\boxed{c(i, j) = w(i, j) + min_{i+1 \le k \le j}\{c(i, k-1) + c(k, j)\}} \rightarrow (4)$$

<u>NOTE:</u>
(1) $c(i, i) = 0,\ 0 \le i \le n,$ because $T_{i,i}$ is empty.
(2) $w(i, i) = q_i,\ 0 \le i \le n$, since $w(i,j)=(p_{i+1} + \cdots + p_j) + (q_i + q_{i+1} + \cdots + q_j)$
(3)

$$\boxed{w(i, j) = p_j + q_j + w(i, j-1)}$$

→ Equation (4), can be solved for $c(0, n)$ by first computing all $c(i, j)$ values such that $j-i = 1$. Next we can compute all $c(i, j)$ values such that $j-i = 2$, then all $c(i, j)$ values with $j-i=3$, and so on till $j-i=n$.

→ If during this computation, we record the root $r(i, j)$ of each tree $T_{i,j}$, then an OBST can be constructed from those $r(i, j)$ values.

<u>Note</u>:
$r(i, j)$ is the value of k that minimizes equation (4).
$r(i, i) = 0,\ 0 \le i \le n.$

<u>Example:</u>

Let $n = 4$ and $(a1, a2, a3, a4) =$ ***(do, if, int, while)***. Let $p(1 : 4) = (3, 3, 1, 1)$ and $q(0 : 4) = (2, 3, 1, 1, 1)$. The $p's$ and $q's$ have been multiplied by 16 for convenience.

Solution:

Initially, we have $w(i, i) = q(i)$, $c(i, i) = 0$ and $r(i, i) = 0$, $0 \le i \le 4$.

So,

$w(0,0) = q_0 = 2$
$w(1,1) = q_1 = 3$
$w(2,2) = q_2 = 1$
$w(3,3) = q_3 = 1$
$w(4,4) = q_4 = 1$

Using equations

$c(i, j)= w(i, j) + min_{i+1\le k\le j}\{c(i,k-1) + c(k,j)\}$ and
$w(i, j) = p(j) + q(j) + w(i, j-1)$,

we get,

for j-i =1:

$w(0, 1) = p(1) +q(1) +w(0,0) = 3+3+2=8$
$c(0, 1)=w(0,l) +min_{k=1}\{c(0,k-1) +c(k, 1)\}=w(0,1) + c(0,0) +c(1, 1)\}=8+0+0=8$
$r(0, 1) = 1$

$w(1, 2) = p(2)+q(2)+w(1, 1) = 7$
$c(1, 2) = w(1, 2) +c(1,1) +c(2, 2) = 7$
$r(1, 2) = 2$

$w(2, 3) = p(3)+q(3)+w(2,2) = 3$
$c(2, 3) = w(2,3)+c(2, 2) +c(3, 3) = 3$
$r(2, 3) = 3$

$w(3, 4) = p(4)+q(4)+w(3,3) = 3$
$c(3, 4) = w(3, 4) +c(3,3) +c(4,4) = 3$
$r(3, 4) = 4$

for j-i =2:

$w(0, 2) = p(2) +q(2) +w(0,1) = 3+1+8 =12$
$c(0, 2) = w(0,2) +min\{c(0,0) +c(1, 2), c(0,1)+c(2,2)\} = 12+min\{0+7,8+0\}= 12+7=19$

r(0, 2) = 1

w(1, 3) = p(3) +q(3) +w(1,2) = 1+1+7 = 9
c(1,3)=w(1,3)+min{c(1,1)+c(2,3),c(1,2)+c(3,3)}=9+min{0+3,7+0}=9+3=12
r(1, 3) = 2

w(2, 4) = p(4) +q(4) +w(2,3) = 1+1+3 = 5
c(2, 4) = w(2,4) +min{c(2,2) +c(3, 4), c(2,3)+c(4,4)} = 5+min{0+3,3+0}=8
r(2, 4) = 3 ***(or)*** *4*

for j-i =3:
w(0, 3) = p(3) +q(3) +w(0,2) = 14
c(0, 3) = w(0,3) +min{c(0,0) +c(1,3); c(0,1)+c(2,3); c(0,2)+c(3,3)}

=14+min{0+12; 8+3;19+0}=14+11
=25
r(0, 3) = 2

w(1, 4) = p(4)+q(4)+w(1,3) =1+1+9=11
c(1, 4) = w(1, 4) +min {c(1,1) +c(2, 4); c(1,2)+c(3,4); c(1,3)+c(4,4)}

=11+min{0+8;7+3; 12+0}
=11+8=19
r(1, 4) = 2

for j-i =4:
w(0, 4) = p(4) +q(4) +w(0,3) =1+1++14= 16
c(0,4)=w(0,4)+min{c(0,0)+c(1,4);c(0,1)+c(2,4);c(0,2)+c(3,4);c(0,3)+c(4,4)}

=16+min{0+19; 8+8; 19+3; 25+0}
=16+16=32
r(0, 4) = 2

j-i ⇩ \ i ⇨	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

<u>Construction of OBST:</u>

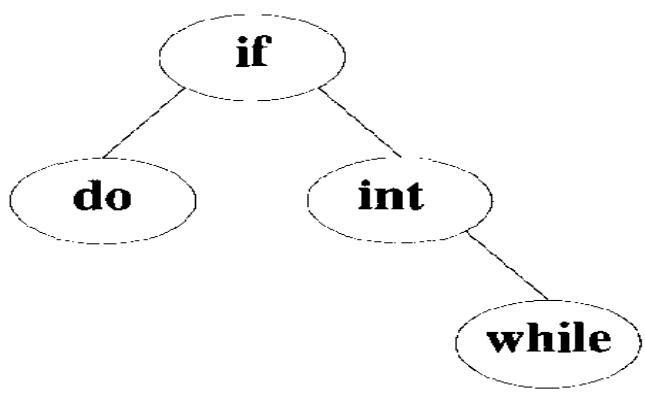
→ From the above table, we see that $c(0, 4)=32$, is the cost of OBST for the given identifiers. And, the root of the OBST $T_{0,4}$ is a_2.

Hence the left subtree is $T_{0,1}$ and the right subtree is $T_{2,4}$.

→ Tree $T_{0,1}$ has root a_1 and subtrees $T_{0,0}$ and $T_{1,1}$.

→ Tree $T_{2,4}$ has root a_3 and its left and right subtrees are $T_{2,2}$ and $T_{3,4}$.

→ Thus, with the above data it is possible to construct OBST as shown below:



Example:

Use function OBST to compute w(i,j), r(i,j) and c(i,j), 0 ≤ i<j ≤ 4 for the identifier set (a1,a2,a3,a4)=(cout, float, if, while) with p(1)= 1/20, p(2)=1/5, p(3)= 1/10, and p(4)=1/20, q(0)=1/5, q(1)= 1/10, q(2)=1/5, q(3)= 1/20, and q(4)=1/20. Using the r(i,j)'s construct the optimal binary search tree.

Solution:

Let
p(1 : 4) = (1, 4, 2, 1) and q(0 : 4) = (4, 2, 4, 1, 1). The p's and q's have been multiplied by 20 for convenience.
Solution:
Initially, we have w(i,i) = q(i), c(i,i) = 0 and r(i,i) = 0, 0 ≤i≤4.
So,

w(0,0) =4
w(1,1)=2
w(2,2)=4
w(3,3)=1
w(4,4)=1

Using Equations
c(i, j) = w(i, j) + $min_{i+1\leq k\leq j}\{c(i,k-1) + c(k,j)\}$
w(i,j) = p(j) + q(j) + w(i, j - 1), we get

for j - i =1:
w(0, 1) = p(l) +q(1) +w(0,0) = 1+2+4 = 7
c(0, 1) = w(0,l) +min{c(0,0) +c(l,l)} = 7
r(0,l) = 1

w(l,2) = p(2)+q(2)+w(l,l) = 4+4+2 = 10
c(1,2) = w(l, 2) +min {c(l,1) +c(2, 2)} = 10
r(1,2) = 2

w(2,3) = p(3)+q(3)+w(2,2) = 2+1+4 = 7
c(2, 3) = w(2,3)+min {c(2, 2) +c(3, 3)} = 7
r(2,3) = 3

w(3,4) = p(4)+q(4)+w(3,3) = 1+1+1 = 3
c(3, 4) = w(3, 4) +min {c(3,3) +c(4,4)} = 3
r(3,4) = 4

for j - i =2:
w(0, 2) = p(2) +q(2) +w(0,1) = 4+4+7 =15
c(0, 2) = w(0,2) +min{c(0,0) +c(l, 2), c(0,1)+c(2,2)} = 15+ min{0+10,7+0}= 15+7=22
r(0, 2) = 2

w(1, 3) = p(3) +q(3) +w(1,2) = 2+1+10 = 13

c(1, 3) = w(1,3) +min{c(1,1) +c(2, 3), c(1,2)+c(3,3)}=13+min{0+7,10+0}= 13+7=20
r(1, 3) = 2

w(2, 4) = p(4) +q(4) +w(2,3) = 1+1+7 = 9
c(2, 4) = w(2,4) +min{c(2,2) +c(3, 4), c(2,3)+c(4,4)} =9+ min{0+3,7+0}= 9+3=12
r(2, 4) = 3

<u>for j - i =3:</u>
w(0, 3) = p(3) +q(3) +w(0,2) = 2+1+15=18
c(0, 3) = w(0,3) +min{c(0,0) +c(l,3); c(0,1)+c(2,3); c(0,2)+c(3,3)}
=18+min{0+20, 7+7, 22+0}=18+14=32
r(0,3) = 2

w(l,4) = p(4)+q(4)+w(l,3) = 1+1+13=15
c(1,4) = w(l, 4) +min {c(l,1) +c(2, 4); c(1,2)+c(3,4); c(1,3)+c(4,4)}
=15+min{0+12, 10+3, 20+0}=15+12=27
r(1,4) = 2

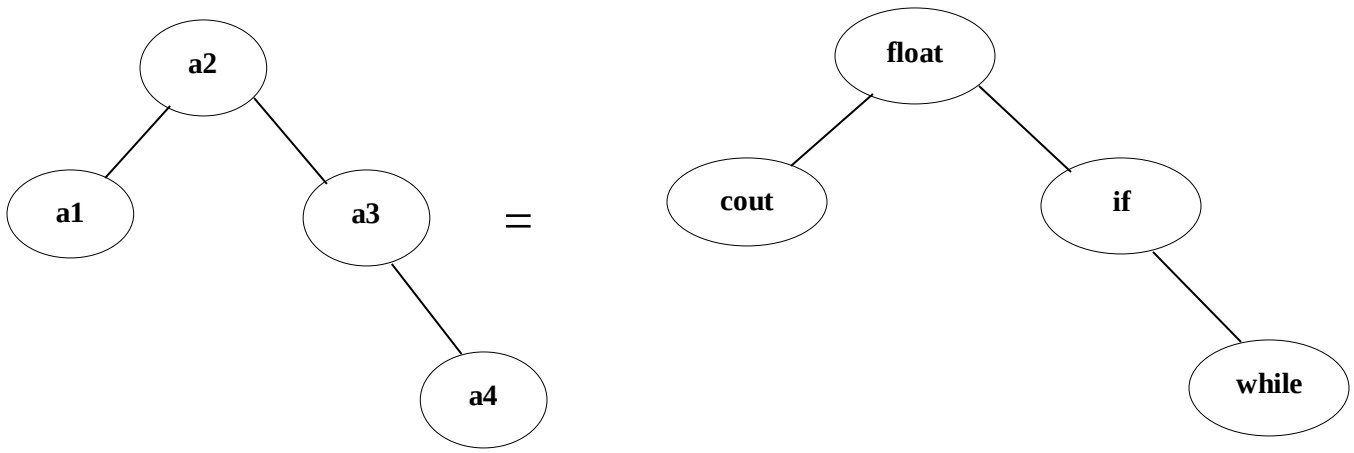
<u>for j - i =4:</u>
w(0, 4) = p(4) +q(4) +w(0,3) = 1+1+18=20
c(0, 4) = w(0,4) +min{c(0,0) +c(l,4); c(0,1)+c(2,4); c(0,2)+c(3,4); c(0,3)+c(4,4)}
= 20+min{0+27, 7+12. 22+3, 32+0}=20+19=39
r(0,4) = 2

j-i ↓ \ i→	**0**	**1**	**2**	**3**	**4**
0	w00=4 c00=0 r00=0	w11=2 c11=0 r11=0	w22=4 c22=0 r22=0	w33=1 c33=0 r33=0	w44=1 c44=0 r44=0
1	w01=7 c01=7 r01=1	w12=10 c12=10 r12=2	w23=7 c23=7 r23=3	w34=3 c34=3 r34=4	
2	w02=15 c02=22 r02=2	w13=13 c13=20 r13=2	w24=9 c24=12 r24=3		
3	w03=18 c03=32 r03=2	w14=15 c14=27 r14=2			
4	w04=20 c04=39 r04=2				

Construction for OBST:

From the above table, we see that c(0, 4)=39, is the minimum cost of OBST for the given identifiers. And the root of the OBST $T_{0,4}$ is a_2.
Hence the left subtree is $T_{0,1}$ and the right subtree is $T_{2,4}$.
Tree $T_{0,1}$ has root a_1 (since r01=1) and subtrees $T_{0,0}$ and $T_{1,1}$.
Tree $T_{2,4}$ has root a_3 (since r24=3) and subtrees are $T_{2,2}$ and $T_{3,4}$.
Tree $T_{3,4}$ has root a_4 (since r34=4) and subtrees $T_{3,3}$ and $T_{4,4}$.

Thus, with the above data, it is possible to construct OBST as shown below:



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. $(M_1*(M_2*M_3))*M_4$

<u>Example:</u>

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*.

<u>DYNAMIC PROGRAMING FORMULATION:</u>

→ 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.
<u>Ex:</u> $M_1 * M_2 * M_3 = M_{1,3}$.

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

→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, ($i \le k < j$),
$M_{i,j} = M_{i,k} * M_{k+1,j}$

→ 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.

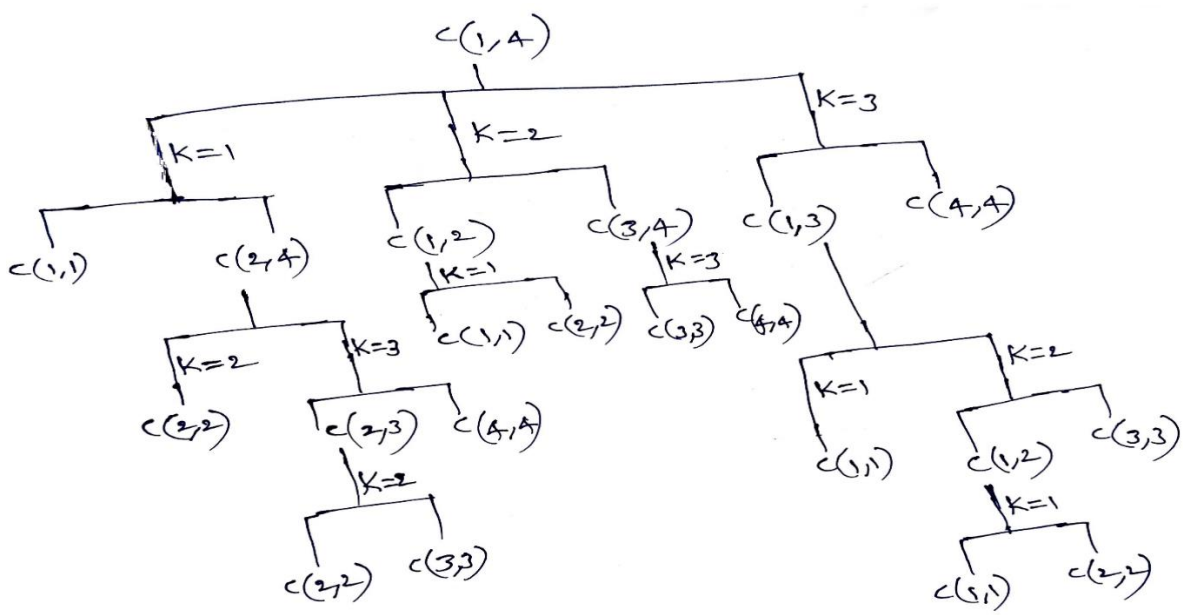
→ The cost of computing $M_{i,k}$ (i.e., $M_i * M_{i+1} * \ldots * M_k$) optimally is $c(i, k)$. The cost of computing $M_{k+1,j}$ (i.e., $M_{k+1} * \ldots * M_j$) optimally is $c(k+1, j)$.
→Since $M_{i,k}$ has dimensions $r_i \times r_{k+1}$ and $M_{k+1,j}$ has dimensions $r_{k+1} \times r_{j+1}$, the 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 $i \le k < j$.

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

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

→ 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.

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



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 $j - i = 1$:

$c(1,2) = min_{1 \le k < 2}\{c(1,1) + c(2,2) + r_1 * r_2 * r_3\}$
$= 0+0+10*5*1 = 50$
$kay(1,2) = 1$

$c(2,3) = min_{2 \le k < 3}\{c(2,2) + c(3,3) + r_2 * r_3 * r_4\}$
$= 0+0+5*1*10 = 50$
$kay(2,3) = 1$

$c(3,4) = min_{3 \le k < 4}\{c(3,3) + c(4,4) + r_3 * r_4 * r_5\}$
$= 0+0+1*10*2 = 20$
$kay(3,4) = 3$

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

For $j - i = 2$:

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

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

$c(3,5) = min_{3 \le k < 5}\{c(3,3) + c(4,5) + r_3 * r_4 * r_6;\; c(3,4) + c(5,5) +$

$r_3 * r_5 * r_6\}$
$=\min\{0+200+1*10*10,\ 20+0+1*2*10\}$
$=\min\{300, 40\}$
$= 40$ kay(3,5)=4

For $j - i = 3$:

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

$c(2,5)=\min_{2\le k<5}\{c(2,2) + c(3,5) + r_2 * r_3 * r_6;\ c(2,3) + c(4,5) + r_2 * r_4 * r_6;\ c(2,4) + c(5,5) + r_2 * r_5 * r_6\}$
$=\min\{0+40+5*1*10,\ 50+200+5*10*10,\ 30+0+5*2*10\}$
$=\min\{90,750,130\}$
$= 90$ 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;

}
```