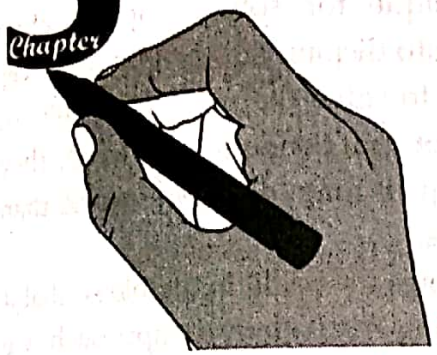
5 Chapter



DYNAMIC PROGRAMMING

CHAPTER OUTLINE

After studying this chapter, the reader will be able to understand the

- Greedy Algorithms vs Dynamic Programming, Recursion vs Dynamic Programming, Elements of DP Strategy
- DP Algorithms: Matrix Chain Multiplication, String Editing, Zero-One Knapsack Problem, Floyd Warshwall Algorithm, Travelling Salesman Problem and their Analysis.
- Memoization Strategy, Dynamic Programming vs Memoization

Introduction

Dynamic Programming is the most powerful design technique for solving optimization problems. Divide & Conquer algorithm partition the problem into disjoint sub-problems solves the sub-problems recursively and then combine their solution to solve the original problems. Dynamic Programming is used when the sub-problems are not independent, e.g. when they share the same sub-problems. In this case, divide and conquer may do more work than necessary, because it solves the same sub problem multiple times.

Dynamic Programming solves each sub-problem just once and stores the result in a table so that it can be repeatedly retrieved if needed again. Dynamic Programming is a Bottom-up approach- we solve all possible small problems and then combine to obtain solutions for bigger problems. Dynamic Programming is a paradigm of algorithm design in which an optimization problem is solved by a combination of achieving sub-problem solutions and appearing to the "principle of optimality".

Characteristics of Dynamic Programming

Dynamic Programming works when a problem has the following features:-

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

If a problem has optimal substructure, then we can recursively define an optimal solution. If a problem has overlapping sub-problems, then we can improve on a recursive implementation by computing each sub-problem only once

Elements of DP Strategy

There are basically three elements that characterize a dynamic programming algorithm:

1. **Substructure:** Decompose the given problem into smaller sub-problems. Express the solution of the original problem in terms of the solution for smaller problems.
2. **Table Structure:** After solving the sub-problems, store the results to the sub problems in a table. This is done because sub-problem solutions are reused many times, and we do not want to repeatedly solve the same problem over and over again.
3. **Bottom-up Computation:** Using table, combine the solution of smaller sub-problems to solve larger sub-problems and eventually arrives at a solution to complete problem.

Development of Dynamic Programming Algorithm

It can be broken into four steps:

- Characterize the structure of an optimal solution.
- Recursively defined the value of the optimal solution. Like Divide and Conquer, divide the problem into two or more optimal parts recursively. This helps to determine what the solution will look like.
- Compute the value of the optimal solution from the bottom up (starting with the smallest sub-problems)
- Construct the optimal solution for the entire problem form the computed values of smaller sub-problems.

Greedy Algorithms vs. Dynamic Programming

Dynamic Programming	Greedy Method
1. Dynamic Programming is used to obtain the optimal solution.	1. Greedy Method is also used to get the optimal solution.
2. A Dynamic programming is an algorithmic technique which is usually based on a recurrent formula that uses some previously calculated states.	2. A greedy method follows the problem solving heuristic of making the locally optimal choice at each stage.
3. In Dynamic Programming, we choose at each step, but the choice may depend on the solution to sub-problems.	3. In a greedy Algorithm, we make whatever choice seems best at the moment and then solve the sub-problems arising after the choice is made.
4. It requires DP table for memorization and it increases its memory complexity.	4. It is more efficient in terms of memory as it never look back or revise previous choices
5. Less efficient as compared to a greedy approach	5. More efficient as compared to a greedy approach
6. It is guaranteed that Dynamic Programming will generate an optimal solution using Principle of Optimality.	6. In Greedy Method, there is no such guarantee of getting Optimal Solution.
7. Example: 0/1 Knapsack	7. Example: Fractional Knapsack

Divide and Conquer Algorithm vs. Dynamic Programming

Divide & Conquer Method	Dynamic Programming
1. It deals (involves) three steps at each level of recursion: • Divide the problem into a number of sub-problems. • Conquer the sub-problems by solving them recursively. • Combine the solution to the sub-problems into the solution for original sub-problems.	1. It involves the sequence of four steps: • Characterize the structure of optimal solutions. • Recursively defines the values of optimal solutions. • Compute the value of optimal solutions in a Bottom-up minimum. • Construct an Optimal Solution from computed information.
2. It is Recursive.	2. It is non Recursive.
3. It does more work on sub-problems and hence has more time consumption.	3. It solves sub-problems only once and then stores in the table.
4. It is a top-down approach.	4. It is a Bottom-up approach.
5. In this sub-problems are independent of each other.	5. In this sub-problems are interdependent.
6. For example: Merge Sort & Binary Search etc.	6. For example: Matrix Multiplication.

table ma store gareko value chai use garne ho aafae lai call gardae garne haenw

less efficient in terms of memory

more efficient in terms of memory

DP Algorithms: Matrix Chain Multiplication

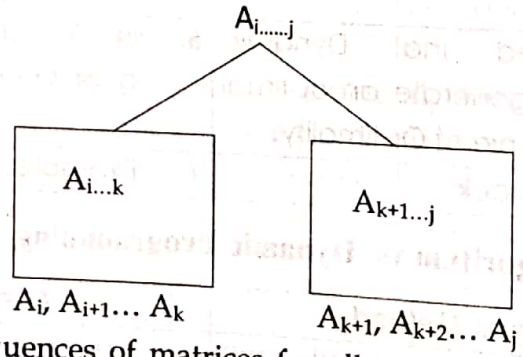
It is a Method under Dynamic Programming in which previous output is taken as input for next. Here, Chain means one matrix's column is equal to the second matrix's row. This algorithm does not perform the multiplications; it just determines the best order in which to perform the multiplications. it just performs the multiplication in which the number of multiplicaiton is less as possible

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

(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations

A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

Even for this small example, considerable savings can be achieved by reordering the evaluation sequence. Given a sequence of matrices $A_1, A_2 \ldots A_n$, and dimensions $P_0, P_1 \ldots P_n$, where A_i is of dimension $P_{i-1} \times P_i$, determine the order of multiplication that minimizes the number of operations. Let Ai....j denote the result of multiplying matrices i through j. It is easy to see that Ai...j is a $P_{i-1} \times P_j$ matrix. So for some k total cost is sum of cost of computing $A_{i...k}$, cost of computing $A_{k+1...j}$, and cost of multiplying $A_{i...k}$ and $A_{k+1...j}$.



Here check all the possible sequences of matrices for all possible choices of k and take best sequence among them.

Recursive definition of optimal solution: let M[i, j] denotes minimum number of scalar multiplications needed to compute $A_{i...j}$.

$$M[i, j] = \begin{cases} 0 & \text{if } i = j \text{ [if sequence contain only one matrix]} \\ \min_{i \le k < j} \{m[i, k] + m[k + 1, j] + P_{i-1} P_k P_j\} & \text{if } i < j \end{cases}$$

Example: Consider matrices A1, A2, A3 and A4 of order 3x4, 4x5, 5x2 and 2x3. Then find the optimal sequence for the computation of multiplication operation.

M Table (Cost of multiplication)

i \ j	1	2	3	4
1	0	60	64	82
2		0	40	64
3			0	30
4				0

S Table (points of parenthesis)

i \ j	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

For m[1,1]= m[2,2]= m[3,3]= m[4,4]=0

m[1,2] = min {m[1, 1]+m[2,2]+p0*p1*p2}=min{0+0+3*4*5}=60

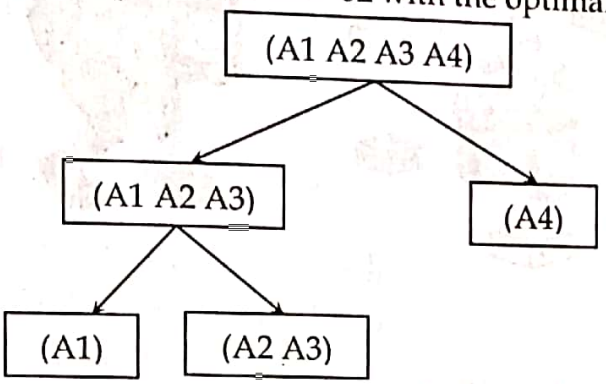
m[2,3] = min {m[2, 2]+m[3,3]+p1*p2*p3}= (0+0+4*5*2)=40

m[1,3] = min {{m[1, 1]+m[2,3]+p0*p1*p3}, { m[1, 2]+m[3,3]+p0*p2*p3}}

= min {(0+40+3*4*2),(60+0+3*5*2)}= min{64,90}= 64

And so on……..

Now the optimal multiplication cost = 82 with the optimal sequence is



(A1A2A3A4) => ((A1A2A3)(A4)) => (((A1)(A2A3))(A4))

This means at first multiply matrix A2 and A3 then multiply their result with matrix A1 and finally multiply their result with A4.

Algorithm

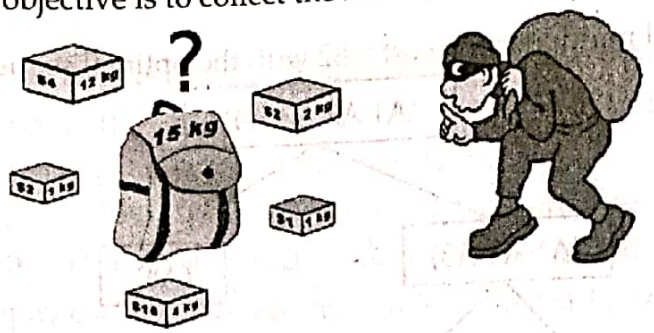
```
Matrix-Chain-Multiplication(p)
{
    n =length[p]
    for( i= 1 i<=n i++)
        m[i, i]= 0
    for(l=2; l<= n; l++)
    {
        for( i= 1; i<=n-l+1; i++)
        {
            j = i + l − 1
            m[i, j] = ∞
            for(k= i; k<= j-1; k++)
            {
                c= m[i, k] + m[k + 1, j] + p[i−1] * p[k] * p[j]
                if c < m[i, j]
                {
                    m[i, j] = c
                    s[i, j] = k
                }
            }
        }
    }
    Return m and s
}
```

Analysis

The above algorithm can be easily analyzed for running time as $O(n^3)$, due to three nested loops. The space complexity is $O(n^2)$.

Zero-One (0/1) Knapsack Problem

A thief has a bag or knapsack that can contain maximum weight W of his loot. There are n items and the weight of i^{th} item is w_i and it worth v_i. An amount of item can be put into the bag is 0 or 1 i.e. x_i is 0 or 1. Here the objective is to collect the items that maximize the total profit earned.



Let W=Capacity of Knapsack

n=No. of items

w = {w1, w2... wn} = weights of items

V= {v1, v2, v3... vn} = value of items

C [i, w] = maximum profit earned with item i and with knapsack of capacity w

Then the recurrence relation for 0/1 knapsack problem is given as,

$$C[i, w] = \begin{cases} 0 & \text{if } = 0 \text{ or } w = 0 \\ C[i-1, w] & \text{if } wi > w \\ \text{Max } \{vi + C[i-1, w-wi], c[i-1, w] & \text{if } i > 0 \text{ and } wi \leq w \end{cases}$$

Example 1: Let there are three item of weight and values are listed below,

W=4

Items = {i1, i2, i3, i4}

wi = {1, 2, 3, 2}

vi = {2, 3, 4, 1}

Where W be the capacity of knapsack, then find maximum profit earned by using 0/1 knapsack problem.

Solution:

Since there are 4 items and knapsack capacity W=4 so we need to construct a table of size 5 x 5 as below,

i \ W	0	1	2	3	4
0	0	0	0	0	0
1	0	2	2	2	2
2	0	2	3	5	5
3	0	2	3	5	6
4	0	2	3	5	6

Since, in first row i=0 and in first column w=0 so C[i, w]=0

For C[1, 1]

i = 1, w = 1, wi = w1 = 1
Since wi ≤ w
So C[i, w]= Max {vi + C[i-1,w-wi], c[i-1,w]}
Or, C[1, 1] = Max {v1 + C[1-1,w-w1], c[1-1,1]}
= Max {2 + C[0, 0], c[0, 1]}
= Max {2+0, 0}
= 2

For C[1, 2]
i=1, w=2, wi=w1=1
Since wi≤ w
So C[i, w]= Max {vi + C[i-1,w-wi], c[i-1,w]}
Or, C[1, 2] = Max {v1 + C[1-1, 2-1], c[1-1,2]}
= Max {2 + C[0, 1], c[0, 2]}
= Max {2+0, 0}
= 2

For C[1, 3]
i=1, w=3, wi=w1=1
Since wi≤ w
So C[i, w]= Max {vi + C[i-1,w-wi], c[i-1,w]}
Or, C[1, 3] = Max {v1 + C[1-1, 3-1], c[1-1,3]}
= Max {2 + C[0, 2], c[0, 3]}
= Max {2+0, 0}
= 2

For C[1, 4]
i=1, w=4, wi=w1=1
Since wi≤ w
So C[i, w]= Max {vi + C[i-1,w-wi], c[i-1,w]}
Or, C[1, 4] = Max {v1 + C[1-1, 4-1], c[1-1,4]}
= Max {2 + C[0, 3], c[0, 4]}
= Max {2+0, 0}
= 2

For C[2, 1]
i=2, w=1, wi=w2=2
Since wi ≥ w
So C[i, w]= C[i-1,w]
Or, C[2, 1] = C[2-1, 1]
= C[1, 1]
= 2

For C[2, 2]
i=2, w=2, wi=w2=2

Since wi ≤ w
So C[i, w]= Max {vi + C[i-1,w-wi], c[i-1,w]}
Or, C[2, 2] = Max {v2 + C[2-1, 2-2], c[2-1, 2]}
= Max {3 + C[1, 0], c[1, 2]}
= Max {3+0, 2}
= 3
For C[2, 3]
i=2, w=3, wi=w2=2
Since wi ≤ w
So C[i, w]= Max {vi + C[i-1,w-wi], c[i-1,w]}
Or, C[2, 3] = Max {v2 + C[2-1, 3-2], c[2-1, 3]}
= Max {3 + C[1, 1], c[1, 3]}
= Max {3+2, 2}
= 5
For C[2, 4]
i=2, w=4, wi=w2=2
Since wi ≤ w
So C[i, w] = Max {vi + C[i-1,w-wi], c[i-1,w]}
Or, C[2, 4] = Max {v2 + C[2-1, 4-2], c[2-1, 4]}
= Max {3 + C[1, 2], c[1, 4]}
= Max {3+2, 2}
= 5
For C[3, 1]
i=3, w=1, wi=w3=3
Since wi ≥ w
So C[i, w]= C[i-1,w]
Or, C[3, 1] = C[3-1, 1]
= C[2, 1]
= 2
For C[3, 2]
i=3, w=2, wi=w3=3
Since wi ≥ w
So C[i, w]= C[i-1,w]
Or, C[3, 2] = C[3-1, 2]
= C[2, 2]
= 3
For C[3, 3]
i=3, w=3, wi=w3=3
Since wi ≤ w
So C[i, w]= Max {vi + C[i-1,w-wi], c[i-1,w]}
Or, C[3, 3] = Max {v3 + C[3-1, 3-3], c[3-1, 3]}
= Max {4 + C[2, 0], c[2, 3]}
= Max {4+0, 5}
= 5
For C[3, 4]
i=3, w=4, wi=w3=3
Since wi ≤ w

So C[i, w]= Max {vi + C[i-1,w-wi], c[i-1,w]}
Or, C[3, 4] = Max {v3 + C[3-1, 4-3], c[3-1, 4]}
= Max {4 + C[2, 1], c[2, 4]}
= Max {4+2, 5}
= 6

For C[4, 1]
i=4, w=1, wi=w4=2
Since wi ≥ w
So C[i, w]= C[i-1,w]
Or, C[4, 1] = C[4-1, 1]
= C[3, 1]
= 2

For C[4, 2]
i=4, w=2, wi=w4=2
Since wi ≤ w
So C[i, w]= Max {vi + C[i-1,w-wi], c[i-1,w]}
Or, C[4, 2] = Max {v4 + C[4-1, 2-2], c[4-1, 2]}
= Max {v4 + C[4-1, 2-2], c[4-1, 2]}
= Max {1 + C[3, 0], c[3, 2]}
= Max {1 + 0, 3}
= 3

For C[4, 3]
i=4, w=3, wi=w4=2
Since wi ≤ w
So C[i, w]= Max {vi + C[i-1,w-wi], c[i-1,w]}
Or, C[4, 3] = Max {v4 + C[4-1, 3-2], c[4-1, 3]}
= Max {1 + C[3, 1], c[3, 3]}
= Max {1+2, 5}
= 5

For C[4, 4]
i=4, w=4, wi=w4=2
Since wi ≤ w
So C[i, w]= Max {vi + C[i-1,w-wi], c[i-1,w]}
Or, C[4, 4] = Max {v4 + C[4-1, 4-2], c[4-1, 4]}
= Max {1 + C[3, 2], C[3, 4]}
= Max {1+3, 6}
= 6

Thus maximum profit = 6 Ans.

Example 2: Let the problem instance be with 7 items where

v[ ] = {2,3,3,4,4,5,7}
w[ ] = {3,5,7,4,3,9,2} and
W = 9

Then find maximum profit earned by using 0/1 knapsack problem.

Solution:

For i=0 or w=0
C[i, w]=0

i.e. C[0,1]=C[0,1]=C[0,2]=C[0,3]=C[0,4]=C[0,5]=C[0,6]=C[0,7]=C[0,8]=C[0,9]=C[1, 0]= .....=0
C[1,1]=C[0,1]=0 since w_1>W i.e. 3>1 so it satisfied second case of the recurrence relation
C[1,3]= Max{V1+C[0,3-3], C[0,3]}=Max{2+0,0}=2 since w >= w1 i.e. 3>=3 so it satisfied the third case.

Continue this process to calculate value of each cell and finally we get following table,

i \ w	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2	2
2	0	0	0	2	2	3	3	3	5	5
3	0	0	0	2	2	3	3	3	5	5
4	0	0	0	2	4	4	4	6	6	7
5	0	0	0	4	4	4	6	8	8	8
6	0	0	0	4	4	4	6	8	8	8
7	0	0	7	7	7	11	11	11	13	15

Profit= C[7][9]=15

Algorithm

```
DynaKnapsack(W, n, v, w)
{
        for(w=0; w<=W; w++)
                C[0,w] = 0;
        for(i=1; i<=n; i++)
                C[i,0] = 0;
        for(i=1; i<=n; i++)
        {
                for(w=1; w<=W;w++)
                {
                        if(w[i]<w)
                        {
                                if v[i] +C[i-1,w-w[i]] > C[i-1,w]
                                        C[i,w] = v[i] +C[i-1,w-w[i]];
                                else
                                        C[i,w] = C[i-1,w];
                        }
                        else
                                C[i,w] = C[i-1,w];
                }
        }
}
```

Analysis

For run time analysis examining the above algorithm the overall run time of the algorithm is O(nW).

Longest Common Subsequence Problem (LCS)

This method is used to test the matching or similarities between the two strings. A subsequence of a given sequence is just the given sequence with some elements left out.

Given two sequences X and Y, we say that the sequence Z is a common sequence of X and Y if Z is a subsequence of both X and Y.

In the longest common subsequence problem, we are given two sequences X = (x1 x2....xm) and Y = (y1 y2 yn) and wish to find a maximum length common subsequence of X and Y. LCS Problem can be solved using dynamic programming.

For example, let X = (ABRACADABRA) and Y = (YABBADABBAD). Then the longest common subsequence is Z = (ABADABA)

Recurrence relation for LCS

Let xi and yj represent any two sequences of characters. L[i, j] represents the LCS of xi and yj then its recurrence relation is,

$$L[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \text{ \{If either of the sequence is empty\}} \\ L[i-1, j-1] + 1 & \text{if } x_i = y_j \text{ \{if last character of both sequence match\}} \\ \text{Max } \{L[i-1], L[i,j-1]\} & \text{if } i > 0, j > 0 \text{ and } xi \neq Y j \text{ \{if last character of both sequence does not match\}} \end{cases}$$

Algorithm

```
LCS(X, Y)
{
        m = length[X];
        n = length[Y];
        for(i=1;i<=m;i++)
                c[i,0] = 0;
        for(j=0;j<=n;j++)
                c[0,j] = 0;
        for(i = 1;i<=m;i++)
        {
                for(j=1;j<=n;j++)
                        if(X[i]==Y[j])
                                c[i][j] = c[i-1][j-1]+1; b[i][j] = "upleft";
                        else if(c[i-1][j]>= c[i][j-1])
                                c[i][j] = c[i-1][j]; b[i][j] = "up";
                        else
                                c[i][j] = c[i][j-1]; b[i][j] = "left";
        }
        Return b and c;
}
```

Analysis

The above algorithm can be easily analyzed for running time as O(mn), due to two nested loops.

The space complexity is O(mn).

Example 1: **Given two sequences X [1...m] and Y [1.....n]. Find the longest common subsequences to both.**
X= {ABCBDAB}
Y= {BDCABA}

Solution:

Here X = (A,B,C,B,D,A,B) and Y = (B,D,C,A,B,A)
m = length [X] and n = length [Y]
m = 7 and n = 6
Here x1= x [1] = A y1= y [1] = B
x2= B y2= D
x3= C y3= C
x4= B y4= A
x5= D y5= B
x6= A y6= A
x7= B
Now fill the values of c [i, j] in m x n table
Initially, for i=1 to 7 c [i, 0] = 0
For j = 0 to 6 c [0, j] = 0
Now for i=1 and j = 1
x1 and y1 we get x1 ≠ y1 i.e. A ≠ B
And c [i-1,j] = c [0, 1] = 0
c [i, j-1] = c [1,0] = 0
That is, c [i-1,j]= c [i, j-1] so c [1, 1] = 0 and b [1, 1] = ' ↑ '
Now for i=1 and j = 2
x1 and y2 we get x1 ≠ y2 i.e. A ≠ D
c [i-1,j] = c [0, 2] = 0
c [i, j-1] = c [1,1] = 0
That is, c [i-1,j]= c [i, j-1] and c [1, 2] = 0 b [1, 2] = ' ↑ '
Now for i=1 and j = 3
x1 and y3 we get x1 ≠ y3 i.e. A ≠ C
c [i-1,j] = c [0, 3] = 0
c [i, j-1] = c [1,2] = 0
So c [1,3] = 0 b [1,3] = ' ↑ '
Now for i=1 and j = 4
x1 and y4 we get. x1=y4 i.e A = A
c [1,4] = c [1-1,4-1] + 1
= c [0, 3] + 1
= 0 + 1 = 1
c [1,4] = 1
b [1,4] = ' ↖ '

Now for i=1 and j = 5

x1 and y5 we get x1 ≠ y5

c [i-1,j] = c [0, 5] = 0

c [i, j-1] = c [1,4] = 1

Thus c [i, j-1] > c [i-1,j] i.e. c [1, 5] = c [i, j-1] = 1. So b [1, 5] = '←'

Now for i=1 and j = 6

x1 and y6 we get x1=y6

c [1, 6] = c [1-1,6-1] + 1

= c [0, 5] + 1 = 0 + 1 = 1

c [1,6] = 1

b [1,6] = ' ↖ '

Now for i=2 and j = 1

We get x_2 and y_1 B = B i.e. $x_2 = y_1$

c [2,1] = c [2-1,1-1] + 1

= c [1, 0] + 1

= 0 + 1 = 1

c [2, 1] = 1 and b [2, 1] = ' ↖ '

Similarly, we fill the all values of c [i, j] and we get

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	X_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ (1)	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ (2)	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ (3)	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ (4)
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Thus Final LCS = {BCBA} of length 4

Example 2: **Consider the character Sequences X=abbabba Y=aaabba find LCS**

X \ Y	Φ	A	A	A	b	b	a
Φ	0	0	0	0	0	0	0
a	0	1	1 upleft	1	1	1	1
b	0	1	1 up	1	2	2	2
b	0	1	1 up	1	2	3	3
a	0	1	2	2 upleft	2	3	4
b	0	1	2	2	3 upleft	3	4
b	0	1	2	2	3	4 upleft	4
a	0	1	2	3	3	4	5 upleft

LCS = a a b b a

Floyd Warshwall Algorithm

The algorithm being discussed uses dynamic programming approach. The algorithm being presented here works even if some of the edges have negative weights. It is mainly used to find all pair shortest path of given weighted graph.

Consider a weighted graph G = (V, E) and denote the weight of edge connecting vertices i and j by w_{ij}. Let W be the adjacency matrix for the given graph G. Let D^k denote an n X n matrix such that $D^k(i, j)$ is defined as the weight of the shortest path from the vertex i to vertex j using only vertices from 1,2,...,k as intermediate vertices in the a^{th}. If we consider shortest path with intermediate vertices as above, then computing the path contains two cases. $D^k(i, j)$ does not contain k as intermediate vertex and $D^k(i, j)$ contains k as intermediate vertex. Then we have the following relations

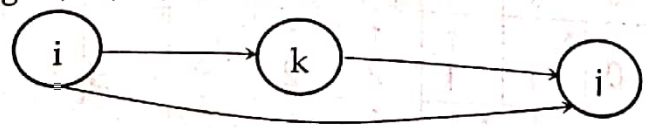
$D^k(i, j) = D^{k-1}(i, j)$ when k is not an intermediate vertex, and

$D^k(i, j) = D^{k-1}(i, k) + D^{k-1}(k, j)$ when k is an intermediate vertex

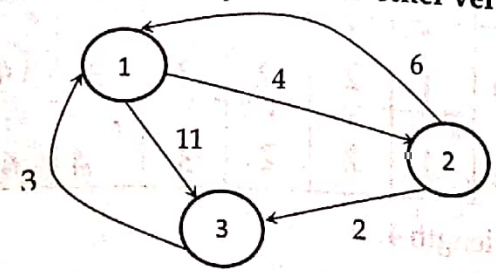
So from above relations we obtain;

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

The above relation is used by Floyd's algorithm to compute all pairs shortest path in bottom up manner for finding $D^1, D^2, D^3, \ldots\ldots D^n$.



Example: **Find shortest path from every vertex to other vertices.**



Solution:

Adjacency matrix of given graph is;

W or D^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

Case 1: When vertex (1) as intermediate vertex:

$D^1(1, 1) = 0$

$D^1(1, 2) = \min\{D^0(1, 2), D^0(1, 1) + D^0(1, 2)\}$ = unchanged= min {4, 0+4} = 4

$D^1(1, 3) = \min\{D^0(1, 3), D^0(1, 1) + D^0(1, 3)\}$ = unchanged=min {11, 0+11} = 11

$D^1(2, 1) = \min\{D^0(2, 1), D^0(2, 1) + D^0(1, 1)\}$ =unchanged= min {6, 6+0} = 6

$D^1(2, 2) = 0$

$D^1(2, 3) = \min\{D^0(2, 3), D^0(2, 1) + D^0(1, 3)\}$ = may change=min {2, 6+11} = 2

$D^1(3, 1) = \min\{D^0(3, 1), D^0(3, 1) + D^0(1, 1)\}$ = unchanged= min {3, 3+0} = 3

$D^1(3, 2) = \min\{D^0(3, 2), D^0(3, 1) + D^0(1, 2)\}$ =may change= min {∞, 3+4} = 7

$D^1(3, 3) = 0$

Thus adjacency matrix can be modified as

D^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

Case 2: When vertex (2) as intermediate vertex:

$D^2(1, 1) = 0$

$D^2(1, 2)$ = unchanged= 4

$D^2(1, 3) = \min\{D^1(1, 3), D^1(1, 2) + D^1(2, 3)\}$ = may change=min {11, 4+2} = 6

$D^2(2, 1)$ = unchanged= 6

$D^2(2, 2) = 0$

$D^2(2, 3)$ = unchanged = 2

$D^2(3, 1) = \min\{D^1(3, 1), D^1(3, 2) + D^1(2, 1)\}$ = may change= min {3, 7+6} = 3

$D^2(3, 2)$ = unchanged = 7

$D^2(3, 3) = 0$

Thus adjacency matrix can be modified as

D^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

Case 3: When vertex (3) as intermediate vertex:

$D^3(1, 1) = 0$

$D^3(1, 2) = \min\{D^2(1, 2), D^2(1, 3) + D^2(3, 2)\}$ = may change= min {4, 6+7} = 4

$D^3(1, 3)$ = unchanged = 6

$D^3(2, 1) = \min\{D^2(2, 1), D^2(2, 3) + D^2(3, 1)\}$ = may change= min {6, 2+3} = 5

$D^3(2, 2) = 0$

$D^3(2, 3)$ = unchanged = 2
$D^3(3, 1)$ = unchanged = 3
$D^3(3, 2)$ = unchanged = 7
$D^3(3, 3) = 0$
Thus adjacency matrix can be modified as

D^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

Thus, the shortest path form vertex 1 to 1=0
The shortest path form vertex 1 to 2=4
The shortest path form vertex 1 to 3=6
The shortest path form vertex 2 to 1=5
The shortest path form vertex 2 to 2=0
The shortest path form vertex 2 to 3=2
The shortest path form vertex 3 to 1=3
The shortest path form vertex 3 to 2=7
The shortest path form vertex 3 to 3=0

Algorithm

let W be a matrix that contain weight of each edges of given graph G, n be number of nodes and D be the adjacency matrix of given graph.

```
FloydWarsal_ASP(W, D, n)
 {
 for(i=1; i<=n; i++)
 {
        for(j=1; j<=n; j++)
        {
              D[i][j]=W[i][j];   //Original D0 matrix
        }
 }
 for(k=1; k<=n; k++)
 {
        for(i=1; i<=n; i++)
        {
              for(j=1; j<=n; j++)
              {
                    if(D[i][j]>D[i][k]+D[k][j]) then
                          Set, D[i][j]=D[i][k]+D[k][j]
              }
        }
 }
}
```

Analysis

The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops. Each loop executes at most O(n) time. The algorithm thus runs in time O(n3).

Travelling Salesman Problem

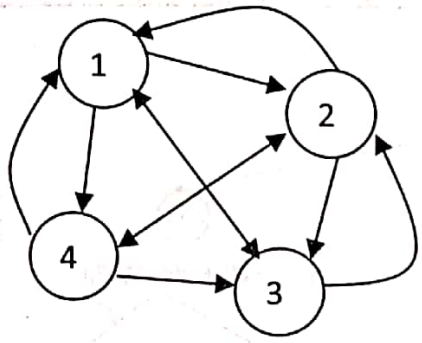
In the traveling salesman Problem, a salesman must visits n cities. We can say that salesman wishes to make a tour or Hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is a non-negative cost C(i, j) to travel from the city i to city j. The goal is to find a tour of minimum cost. We assume that every two cities are connected. Such problems are called Traveling-salesman problem (TSP).

The travelling salesman problem can be solved either by dynamic program paradigm or non-dynamic programming ways. But here we use dynamic programming ways as below,

Let i be the starting vertex, S be the set of remaining vertices except vertex i, k be the any one vertex of S and g(i, S) be the minimum travel cost of TSP. Then their recurrence relation can be defined as below,

$$g(i, S) = \text{Min } k \varepsilon S \{ C_{ik} + g(k, S-(k)\}$$

Example: Let's take a weighted directed graph as below,



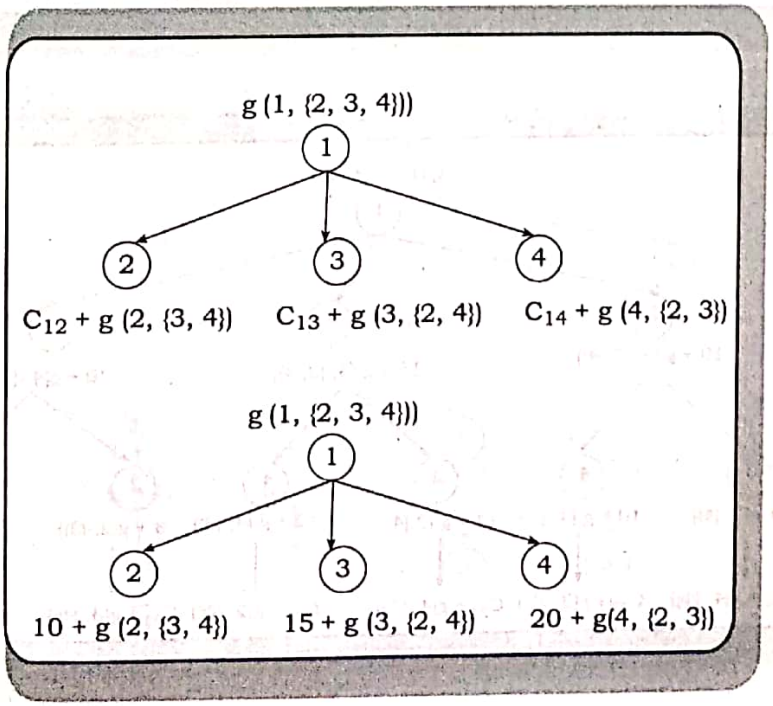
	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

Solution:

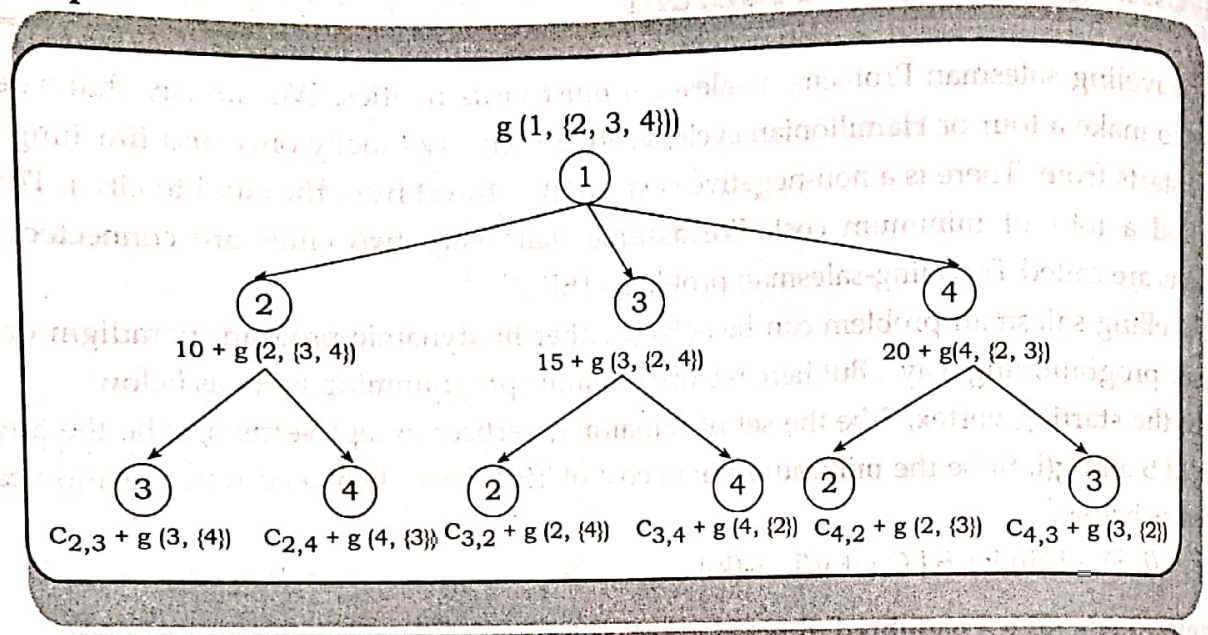
The given formula can be modified for above problem is,

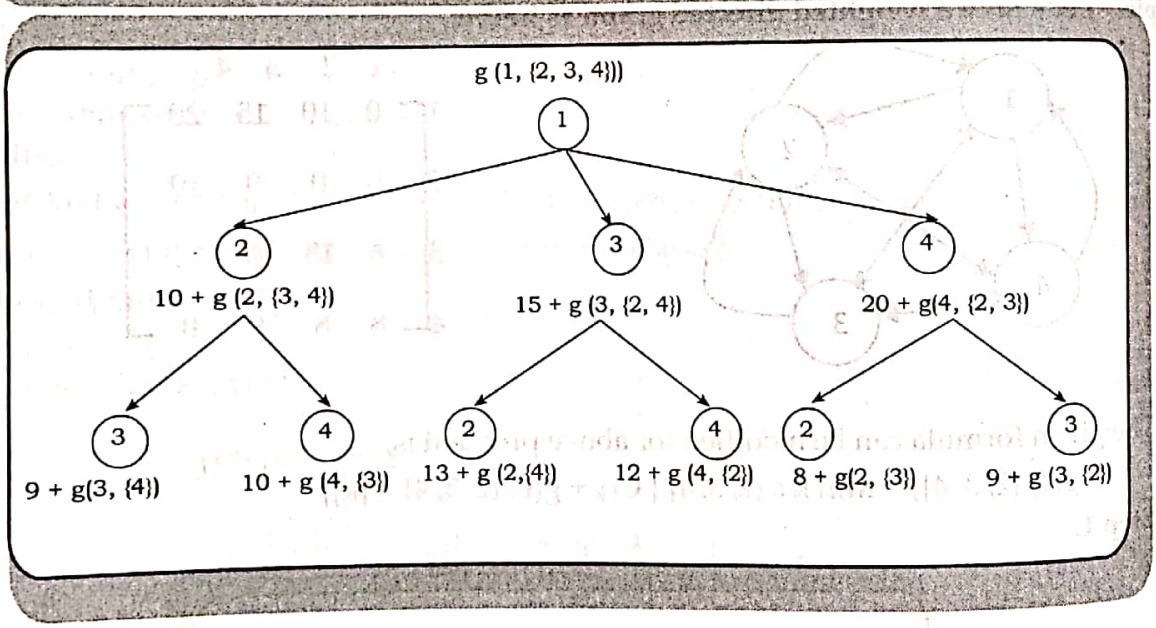
$$g(1, \{2, 3, 4\}) = \text{Min } k \varepsilon \{2, 3, 4\} \{ C_{1k} + g(k, \{2, 3, 4\} - \{k\}\}$$

Step 1:

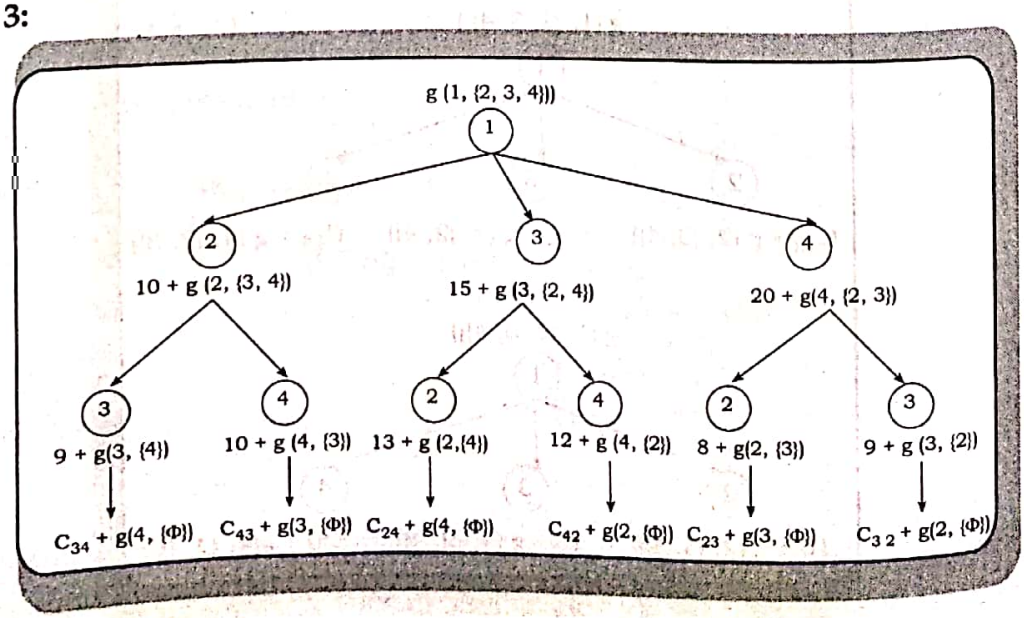


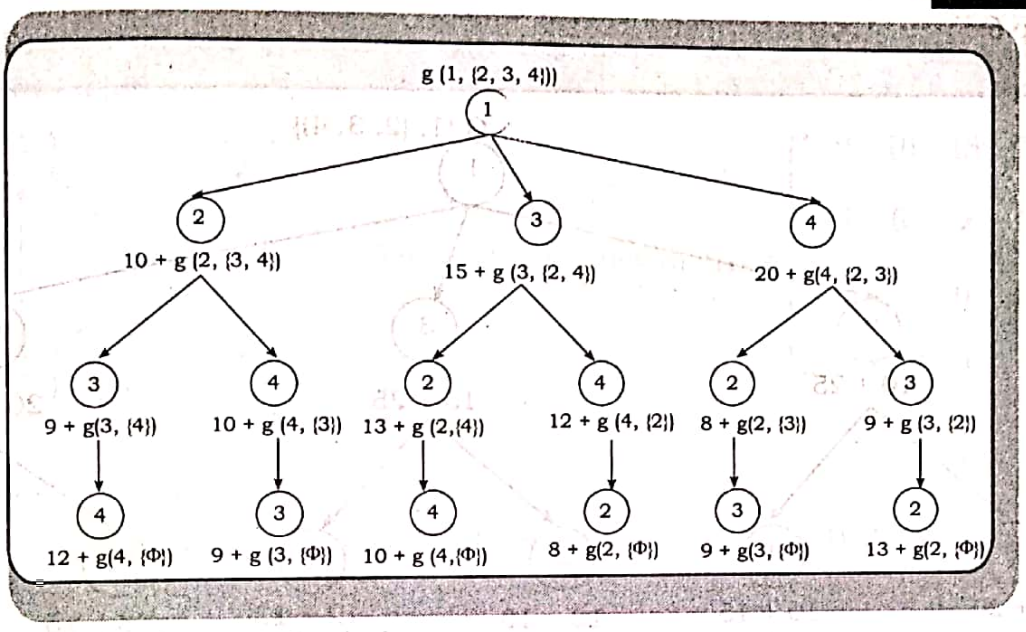
Step 2:



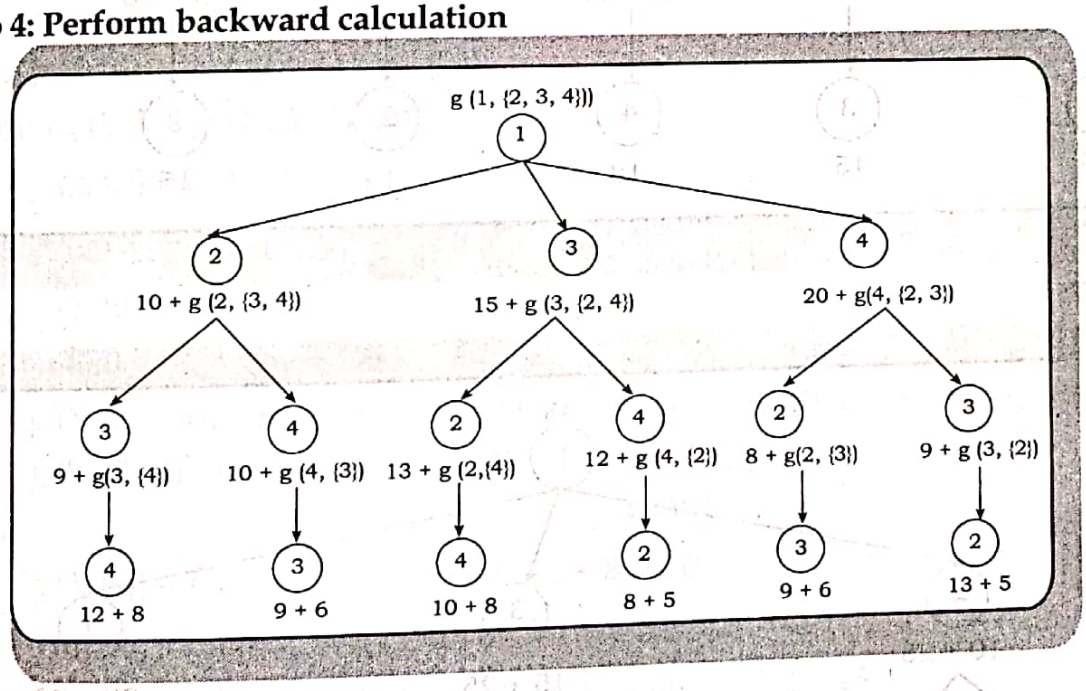


Step 3:

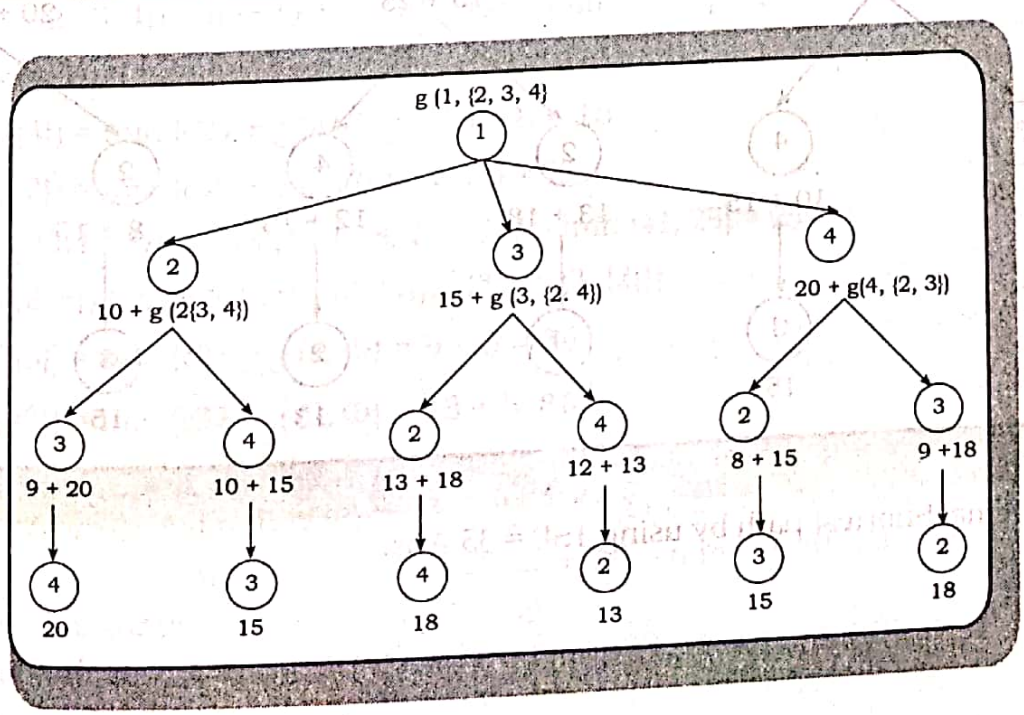




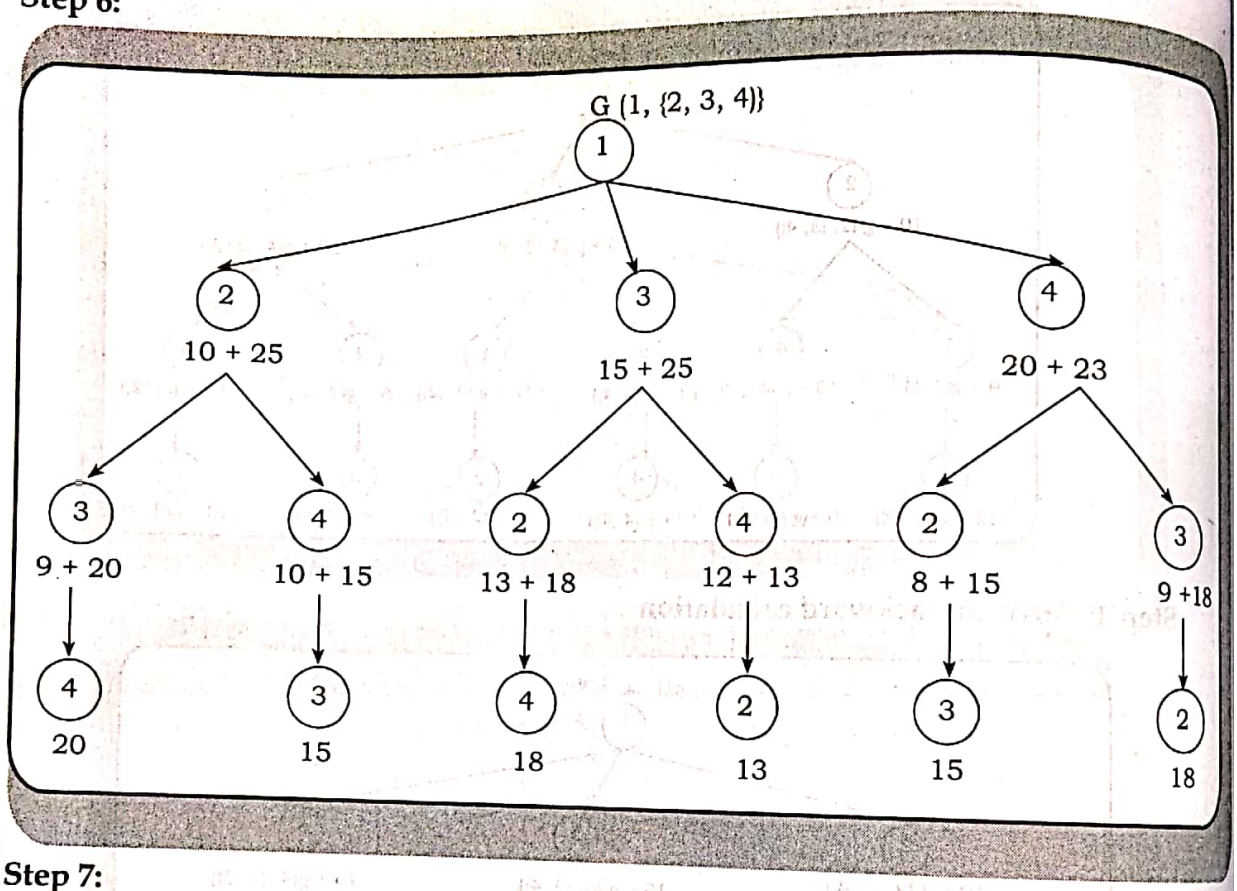
Step 4: Perform backward calculation



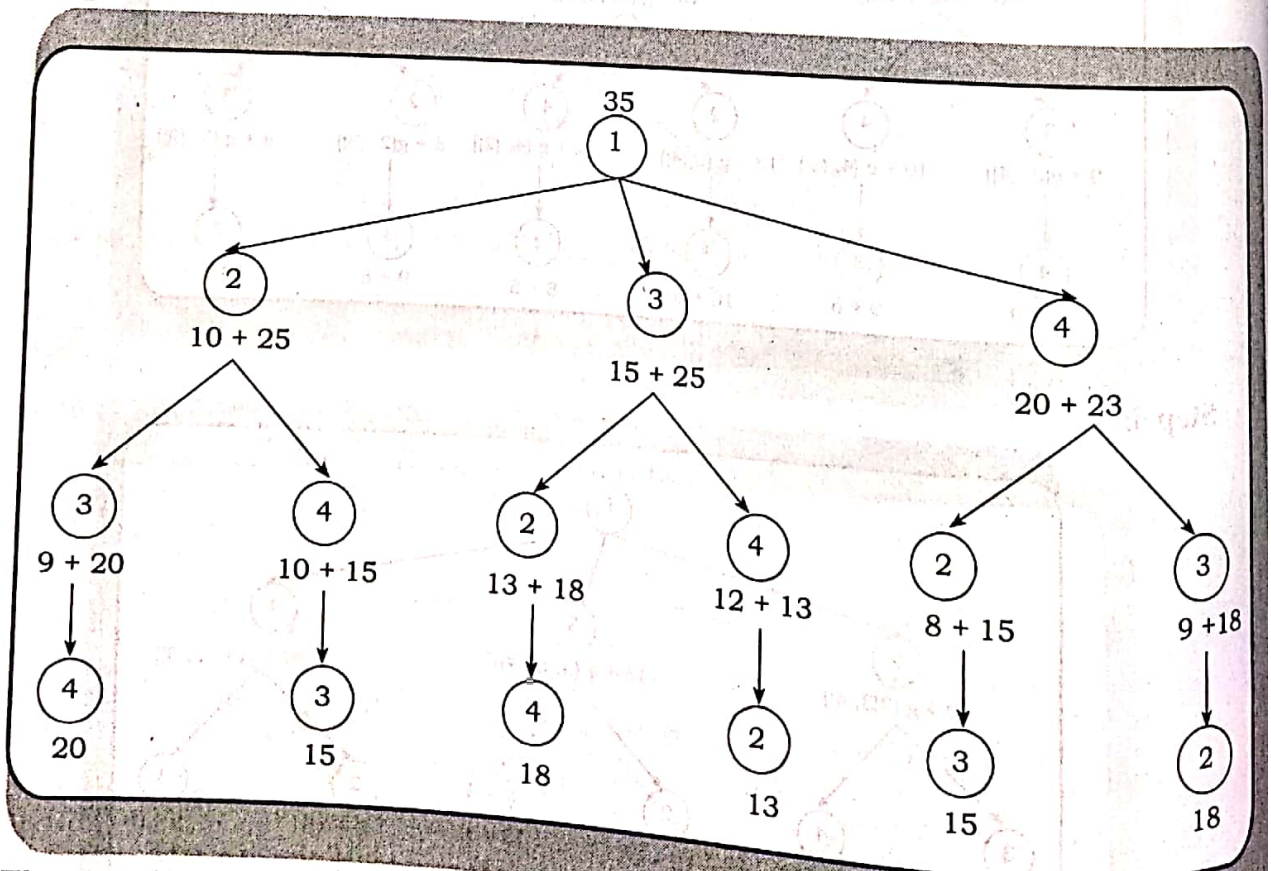
Step 5:



Step 6:

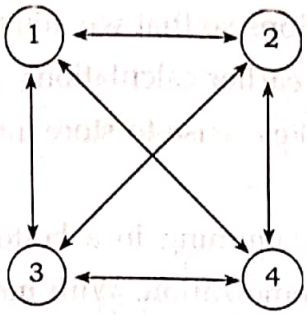


Step 7:



Thus Final Shortest path by using TSP = 35 Ans.

Example 2: **For the following graph find minimum cost tour for the traveling salesperson problem:**



The cost adjacency matrix = $\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$

Solution:

Let us start the tour from vertex 1:

$g(1, V - \{1\}) = \min \{c1k + g(k, V - \{1, K\})\}$ .....(1)

$2<k<n$

More generally writing:

$g(i, s) = \min \{cij + g(J, s - \{J\})\}$ .....(2)

Clearly, $g(i, \Phi) = ci1$, $1 \le i \le n$. So,

$g(2, \Phi) = C21 = 5$

$g(3, \Phi) = C31 = 6$

$g(4, \Phi) = C41 = 8$

Using equation – (2) we obtain:

$g(1, \{2, 3, 4\}) = \min \{c12 + g(2, \{3, 4\}, c13 + g(3, \{2, 4\}), c14 + g(4, \{2, 3\})\}$

$g(2, \{3, 4\}) = \min \{c23 + g(3, \{4\}), c24 + g(4, \{3\})\}$

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

$g(3, \{4\}) = \min \{c34 + g(4, \Phi)\} = 12 + 8 = 20$

$g(4, \{3\}) = \min \{c43 + g(3, \Phi)\} = 9 + 6 = 15$

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

$g(3, \{2, 4\}) = \min \{(c32 + g(2, \{4\}), (c34 + g(4, \{2\})\}$

$g(2, \{4\}) = \min \{c24 + g(4, \Phi)\} = 10 + 8 = 18$

$g(4, \{2\}) = \min \{c42 + g(2, \Phi)\} = 8 + 5 = 13$

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

$g(4, \{2, 3\}) = \min \{c42 + g(2, \{3\}), c43 + g(3, \{2\})\}$

$g(2, \{3\}) = \min \{c23 + g(3, \Phi\} = 9 + 6 = 15$

$g(3, \{2\}) = \min \{c32 + g(2, \Phi\} = 13 + 5 = 18$

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

$g(1, \{2, 3, 4\}) = \min \{c12 + g(2, \{3, 4\}), c13 + g(3, \{2, 4\}), c14 + g(4, \{2, 3\})\}$

$= \min \{10 + 25, 15 + 25, 20 + 23\} = \min \{35, 40, 43\} = 35$

The optimal tour for the graph has length = 35

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

Memoization Strategy

Memoization means recording the results of earlier calculations so that we don't have to repeat the calculations later. If our code depends on the results of earlier calculations, and if the same calculations are performed over-and-over again, then it makes sense to store interim results so that we can avoid repeating the math.

So far we have talked about implementing dynamic programming in a bottom up fashion. Dynamic programming can also be implemented using memoization. With memoization, we implement the algorithm recursively, but we keep track of all of the sub solutions. If we encounter a sub-problem that we have seen, we look up the solution. If we encounter a sub-problem that we have not seen, we compute it, and add it to the list of sub solutions we have seen. Each subsequent time that the sub-problem is encountered, the value stored in the table is simply looked up and returned.

Memoization offers the efficiency of dynamic programming. It maintains the top-down recursive strategy.

Dynamic Programming vs. Memoization

Dynamic programming algorithm usually outperforms a top-down memoization algorithm by constant factor, because there is no over-head for recursion and fewer overheads for maintaining the table. In situations where not every sub-program is computed, memoization only solves those that are needed but dynamic programming solves all the sub-problems.

In summary, the matrix chain multiplication problem can be solved in $O(n^3)$ times by either a top-down, memorized algorithm or a bottom-up dynamic programming algorithm.

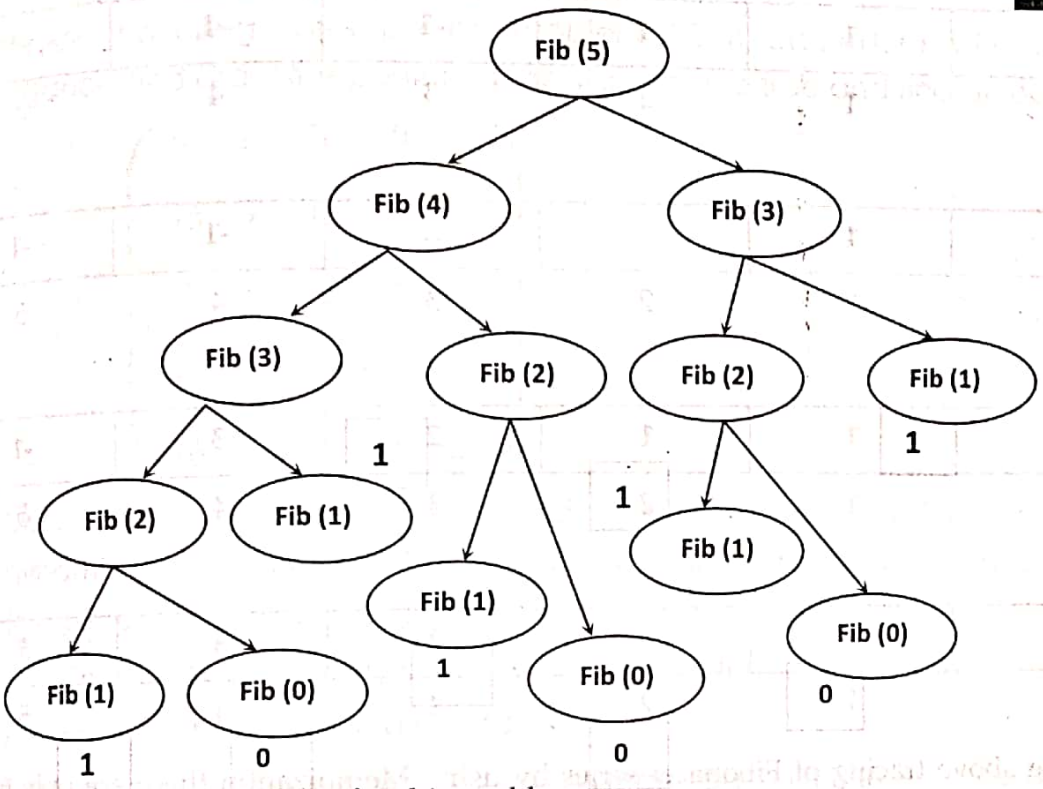
Let's take a problem of finding Fibonacci number by using recursion as below,

```
int Fibo(int n)
{
        if(n<=1)
                return n;
        else
                return Fibo(n-1)+Fibo(n-2);
}
```

Their recurrence relation is,

$$Fibo(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ Fibo(n-1) + Fibo(n-2) & \text{if } n \geq 2 \end{cases}$$

If we use dynamic programming for calculation Fibonacci number it works as below, since dynamic programming uses recursion extensively.



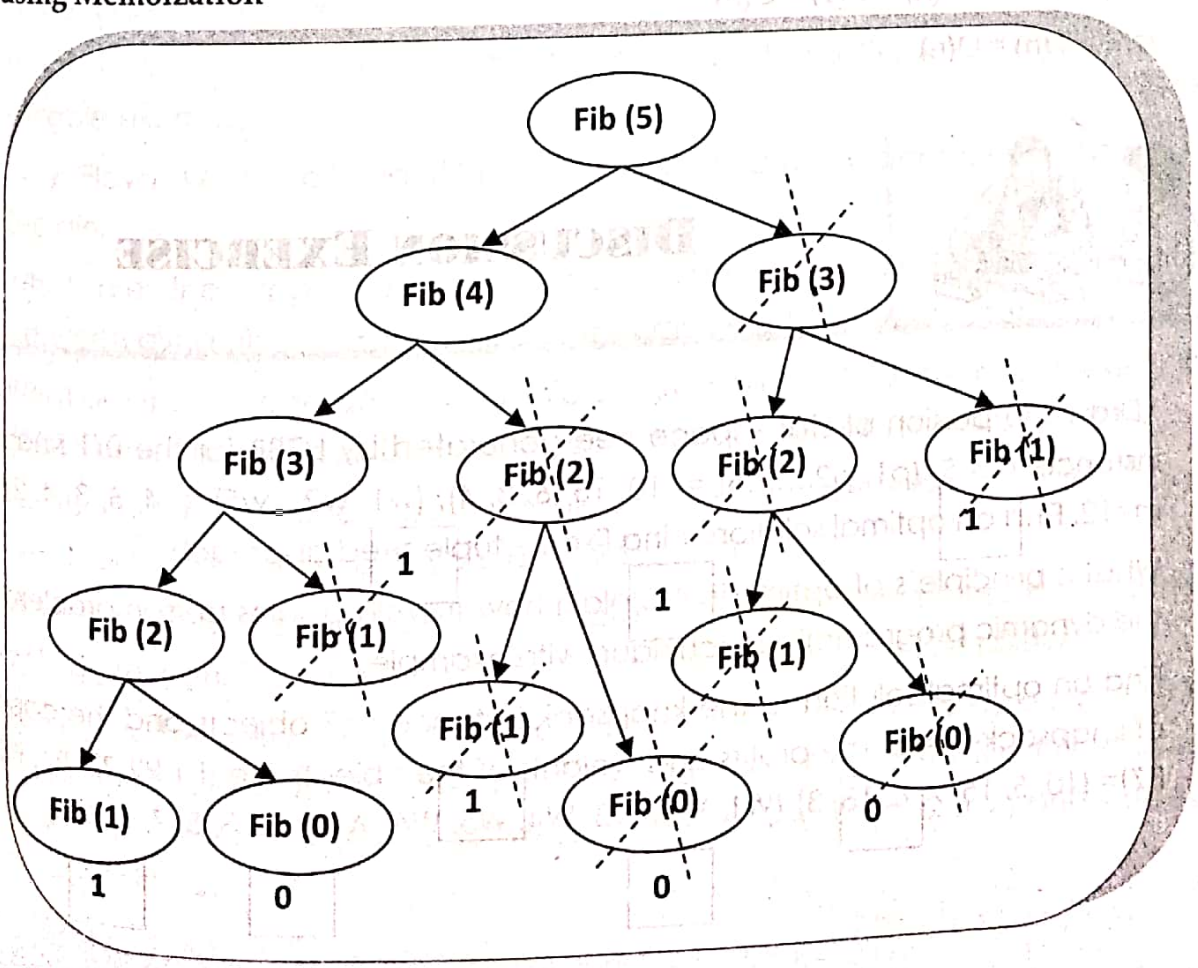
Since recurrence relation for this problem is,

$T(n)=2T(n-1) + 1$

By solving this we get,

$T(n)= O(2^n)$

Now using **Memoization**



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

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

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

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

From the above tracing of Fibonacci series by using Memoization there are only 6 calls are occur for n=5

Hence for a problem of size n there are (n+1) calls may occur.

Thus total time complexity,

$T(n) = n + 1 = O(n) + O(1) = O(n)$

$\Rightarrow \quad T(n) = O(n)$

DISCUSSION EXERCISE

1. \Draw the portion of state space tree generated by LCBB for the 0/1 Knapsack instance: n = 5, (p1, p2,...,p5) = (10, 15, 6, 8, 4), (w1, w2,...,w5) = (4, 6, 3, 4, 2) and m=12. Find an optimal solution using fixed – tuple sized approach.
2. What is principle's of optimality? Explain how travelling sales person problem uses the dynamic programming technique with example.
3. Find an optimal solution to the knapsack instance n=7 objects and the capacity of knapsack m=15. The profits and weights of the objects are (P1,P2,P3, P4, P5, P6, P7)= (10, 5, 15, 7, 6, 18, 3) (W1, W2, W3, W4, W5, W6, W7)= (2, 3, 5, 7, 1, 4, 1)

4. Explain travelling sales person problem LCBB procedure with the following instance and draw the portion of the state space tree and find an optimal tour.

$$\begin{pmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{pmatrix}$$

5. Describe the Dynamic 0/1 Knapsack Problem. Find an optimal solution for the dynamic programming 0/1 knapsack instance for n=3, m=6, profits are (p1, p2, p3) = (1, 2, 5), weights are (w1, w2, w3)=(2,3,4).
6. Explain how Matrix – chain Multiplication problem can be solved using dynamic programming with suitable example.
7. Differentiate between greedy approach and dynamic programming with suitable example.
8. What is dynamic programming? Explain characteristics of dynamic programming.
9. How dynamic programming differs from divide and conquer algorithm? Explain with suitable example.
10. What is all pair shortest path problem? Explain Floyd Warshwall algorithm with suitable example.
11. Why Floyd Warshwall algorithm is called dynamic programming paradigm? Explain.
12. What are the applications of dynamic programming? Explain differentiate between dynamic programming and recursive algorithm with suitable example.
13. What are the advantages of dynamic programming? Give the recursive definition of LCS problem. Find LCS between sequences S1="Abina", S2="Aarav".
14. What is all pair shortest path problem? Find all pair shortest path of given weighted graph by using Floyd Warshall algorithm.
15. What are the advantages of dynamic programming? Give the recursive definition of LCS problem. Find LCS between sequences S1="Dinesh", S2="Dikshya"
16. What are the characteristics of problem that can be solved by using dynamic programming algorithm? Give the recursive definition of solving 0/1 knapsack problem. Trace the algorithm for w={3, 4, 2, 2, 3}, v={12, 14, 6, 5, 6} and knapsack of capacity 12.

17. Define Memoization strategy with suitable example.
18. Differentiate between dynamic programming vs. memorization with suitable example.
19. Find optimal sequence of matrices A, B, C and D of order respectively 3 x 4, 4 x 2, 2 x 3 and 3 x 4 by using matrix chain multiplication.
20. Write down the elements of dynamic programming. Also mention their advantages and disadvantages.

❑❑❑