UNIT-3

Dynamic Programming

* "Dynamic Programming" simply refers to "Solving Problems (Programming) by using a dynamic memory".
* Dynamic Programming is a optimization technique of solving Recursion-based Problems with better time Complexity.
* DP is suitable for Recursive problems which:-
  - ↳ Solves a problem by dividing into Sub-Problems [divide-and-Conquor].
  - ↳ has overlapping Sub-Problems. [Same Sub-Problems occur more than once].
* The idea is to simply store the results of sub-problems, so that we dont have to re-compute them when needed later in the problem.

* This simple optimization, reduces time Complexities from Exponential to polynomial. [But space complexity will increase due to usage of memory for storing sub-Problems results].

General Method:-

* General method of Solving Problem via dynammic Programming approach:-
  - Break down the complex problem into simpler Sub-problems.
  - Store the results of Sub-problems whenever they are Computed.
  - use these results whenever the same Sub-problem occurs.
  - Compute the result of final problem with these Sub-problems.

* DP based problems Can be Solved in two approaches :-
  (i) Top-down Approach
  (ii) Bottom-up Approach.

(i) Top-down Approach:- [Recursion + Memorization]
* In this Approach, the original problem is solved by breaking it down into Smaller Sub-problems.
* The Solution to each Sub-problem is Computed only when needed, and the results are Stored in an array [memorized].
* This approach uses Recursion.

(ii) Bottom-Up Appraoch:- [Iteration + Tabulation]
* In this Approach, the Solution to original Problem is built by

Solving the sub-problems (from smallest to largest).

* The results of sub-problems are stored in a table [1-D/2-D Array] and these results are used to compute the bigger sub-problems [and finally original Problem].
* This approach uses iterative loops to build solutions.
* This approach is more efficient than top-down approach.

Since,

Top-down ⇒ (dividing) + (Building)

Bottom-up ⇒ (Building).

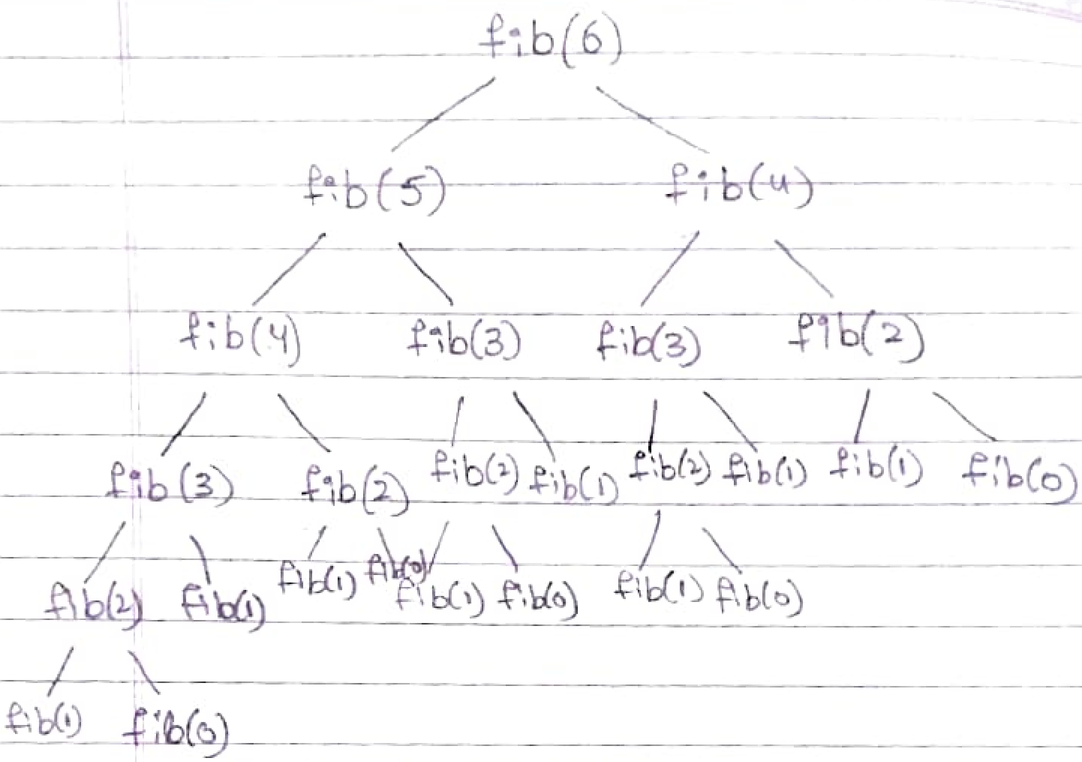
Extra time

Ex:- Fibonacci Sequence.

(1) General method. [Recursion]

```
fib (n) {
    if(n==0 || n==1) return (n);
    return (fib(n-1)+fib(n-2));
}
```

Consider, fib(6):-



⇒ fib(2), fib(3), fib(4) are calculated again and again, which increases the computation time.

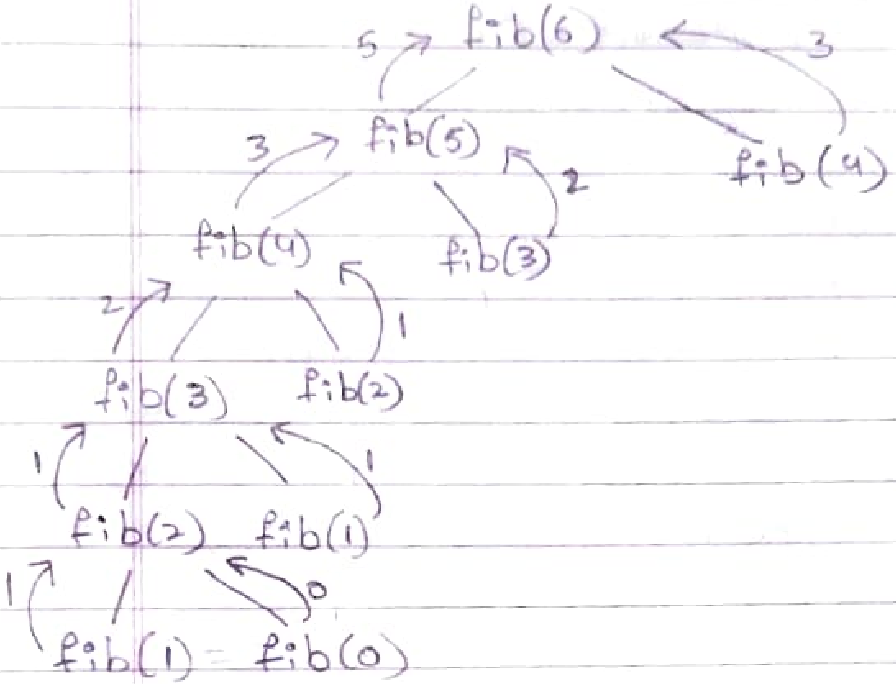
* Time Complexity :- $O(2^n)$
* Space Complexity :- $O(n)$

(2) DP : Top-down [Recursion + Memorization]

```
int fibResults[n]; // inialized with null
fib(n){
    if(n==0 || n==1) return (n);
    if(fibResults[n] != null) return fibResults[n];
    fibResults[n] = fib(n-1) + fib(n-2)
    return fibResults[n];
}
```

Consider, fib(6) :-

oResults = [null, null, fib(2), fib(3), fib(4), fib(5), fib(6)]
= [null, null, 1, 2, 3, 5, 8]



* time complexity :- $O(n)$
* space complexity :- $O(n)$

(3) DP : Bottom-up [Iteration + Tabulation]

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

fibResults = [0, 1, 1, 2, 3, 5, 8]

(0 + 1 = 1)

* time Complexity :- $O(n)$
* Space Complexity :- $O(n)$

⇒ Applications of DP :-

1. Optimal BST
2. 0/1 Knapsack Problem.
3. All pairs Shortest path problem
4. Traveling Sales person problem
5. Reliabity Design Problem.

0 / 1 Knapsack
(Exclude an Item) (a bag)

(Include Item)

classmate

Date
Page

0/1 Knapsack Problem :-

Problem Statement :-

Given (N) items where each item has some weight (w_i) and Profit (P_i) associated with it, and also given a bag with Capacity (W). [i.e., the bag can hold at most W weight in it].

The task is to put the items into the bag such that the sum of Profits associated with them is the maximum Possible.

Constraint :- We can either put an item completely into the bag (or) cannot put it at all. [i.e., it is not possible to put a part of an item into the bag]

(and) Total weight of items must be less than/equal to bag capacity

$\sum w[i] \leq W$

Solving Procedure by using DP :-

let (W) be the ~~bag~~ total weight capacity of Knapsack (K)

let (N) be the items to be filled into Knapsack (K)

let w[i], P[i] be the weight and Profit of ith item respectively.

⇒ Step ① :- Create a 2D table where rows represent items and columns represent the weights (from 0 to the maximum Knapsack capacity).

here, K[i][w] is the value of a cell with item (i) and weight (w).

⇒ **Step ②:-** Set the values in the first row and column of the table to (0), as there is no ~~weight~~ Profit ~~...~~ for items when the knapsack has (0) capacity (w=0) (or) when no item to store (i=0)

i.e., when i=0 (or) w=0, $\boxed{K[i][w]=0}$

i.e., K[0][w]=0, K[i][0]=0.

⇒ **Step ③:-** Now, there will be two cases to get the profit value (K[i][w]).

↳ **Case (i):-** If weight of ~~...~~ item (i) is less than / Equal to current knapsack weight capacity (w). then we use the below formula, which follows DP approach to compute profit of a cell:- i.e., when (w[i] ≤ w)

$$\boxed{K[i][w] = \max(P[i] + K[i-1][w - w[i]],\ K[i-1][w])}$$

↳ **Case (ii):-** If the first case fails, then:- fill the previous value from above row.

i.e., $\boxed{K[i][w] = K[i-1][w].}$

⇒ **Step ④:-** After the table is filled, we need to determine, ~~...~~ the item which must be included to get maximum Profit value.

∴ we have to build a sequence (0,1) which Portrays which item to include and which to not include.

∴ Start from maximum in the table at the last row and last column.

→ if the same Profit is Present in Previous row, the do not include that item (i.e., 0), Else include the item (i.e., 1).

→ After including an item, Subtract the corresponding Profit of that item and repeat the same with remaining Profit value.

<u>Algorithm</u>:-

```
Knapsack_DP(){
  int N, W, P[N+1], w[N+1];
  int K[N+1][W+1];   // 2D Table

  for(int i=0; i<=N; i++){
     for(int w=0; w<=W; w++){
        if(i==0 || w==0){
              K[i][w] = 0;
        }else if(w[i] <= w){
  K[i][w] = max(P[i]+K[i-1][w-w[i]], K[i-1][w]);
        }else{
              K[i][w] = K[i-1][w];
        }
```

Ex:- Given (N=4) items and a Knapsack of Capacity (W=8). use DP approach to find the items to be filled in Knapsack such that the total Profit is maximum.

Profits of (4) item:- $P[i] = \{1, 2, 5, 6\}$

weights of (4) items:- $W[i] = \{2, 3, 4, 5\}$

given,

Item	weight	Profit
1	2	1
2	3	2
3	4	5
4	5	6

Step ①:- Knapsack table with (N+1) Rows, (W+1) Columns

(i) \ (w)	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3
3	0	0	1	2	5	5	6	7	7
4	0	0	1	2	5	6	6	7	8

Step ②:- If there are no items to be filled i.e., (i=0), the Profit is (0) i.e, $K[i][w]=0$ irrespective of weight (w) of Knapsack.

If there is no Weight Capacity of Knapsack (w) i.e (w=0), the Profit is (0) irrespective of items (i) to be filled. (Cannot be filled)

∴ Values of 1st Row & Column are (0)

~~(i=1, w=1)~~

→ for K[1][1], w[1] = 2 and w = 1

∴ (w[1] <= w) is false

∴ K[1][1] = K[0][1] = 0

→ for (i=1, w=2), w[1]=2, w = 2

∴ (w[1] <= w) is true.

∴ K[1][2] = max(P[1] + K[0][2−2], K[0][2])

= max(1+0, 0) = 1

→ Since, now :- for (i=1) row,

w[1] <= w is always true

and max(P[1]+K[0][w−w[1]], K[0][2])

is always (1).

∴ remaining values for (i=1) row are (1)

→ for (i=2, w=1), w[2] = 3 and w=1

∴ w[2] <= w is false

∴ take Previous rows values.

i.e., K[i][w] = K[i−1][w]

until w ≠ 3 (upto w=2)

→ for (i=2, w=3), w[2]=3,

∴ w[2] <= w is true.

∴ K[2][3] = max(P[2] + K[1][3−3], K[1][3])

= max(2+0, 1) = 2

→ for (i=2, w=4), w[2] <= w is true.

∴ K[2][4] = max(2 + K[1][4−3], K[1][4])

= max(2+0, 1) = 1

∵ true for next cells of i=2

∴ K[2][5] = max(2 + K[1][5−3], K[1][5])

max(2+1, 1) = 3

∴ K[2][6] = max(2+ K[1][6−3], K[1][6])

= max((2+1), 1)

= 3

Since, now, $K[1][w-3]$ is always (1)
and $K[1][w]$ is always (1)
$\therefore$ it is (3) for rest.

$\to$ for $(i=3, w=1)$,
$w[3] <= w$ is false
$\therefore K[3][1] = K[2][1] = 0$.

Same case until $w = w[3] = 4$
i.e., upto $(w=3)$

$\to$ for $(i=3, w=4)$,
$w[3] <= w$ is true
$\therefore K[3][4] = \max(P[3] + K[2][4-4], K[2][4])$
$= \max(5+0, 2) = 5$

and so on....

<u>Last Step</u>:- finding sequence:-

i_1	i_2	i_3	i_4
0	1	0	1

8
$8 - 6 = 2$
$2 - 2 = 0$

$\therefore$ Item(2) and Item (4) must be included in Knapsack to get max profit of (8).

All Pairs Shortest Path Problem:-
[floyd-warshall Algorithm]

<u>Problem</u>:- Find the shortest paths b/w all pairs of nodes in a given weighted graph.

classmate

Date

Page

→ floyed-warshall Algorithm is the one which follows DP approach for finding shortest path b/w every pair of nodes of weighted graph.

Procedure:-

~~[illegible]~~ let (n) be the no. of nodes of given weighted graph.

Step ①:- Build an (n×n) matrix (A^0) which portrays the distance (weight) b/w every pair of node, such that there are no intermediate nodes in the paths. [i.e., direct neighbours]

i.e., A[source node, destination node] = weight/distance

Step ②:- Now, from matrix (A^0), build a matrix (A^1) by considering an ~~[illegible]~~ intermediate node for every path. Continue this ~~[illegible]~~ ~~[illegible]~~ and find matrices for every node as an intermidiate node (i.e., (n) nodes)

∴ There will be total (n+1) matrices.

→ The formula used for finding minimum weight/distance value from node (i) to node (j) is:-

[i = source node
j = dest node]

$$\boxed{A^{K}[i,j] = \min\left(A^{K-1}[i,j],\ A^{K-1}[i,K] + A^{K-1}[K,j]\right)}$$

[K ⇒ intermidiate node]

Algorithm:-

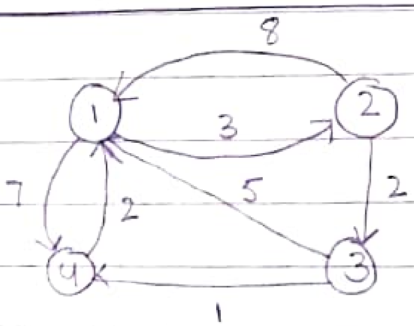
(we will be using a single matrix but, A^0 is the matrix initialization part of the Algorithm)

```
APSP(graph){
    int n; // no. of nodes
    int A[n][n];
    // initialization
    for each edge (u,v) in graph:
            A[u][v] = weight(u,v)

    for(k=1; k<=n; k++){
        for(i=1; i<=n; i++){
            for(j=1; j<=n; j++){
                A[i][j]=min(A[i][j], A[i][k]+A[k][j])
            }
        }
    }
```

Note:- for matrix[A^k], the (kth) row and kth column weights will be same as that of in matrix [A^{k-1}]

Ex:-



$\infty \rightarrow$ no path

$$A^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & 2 & \infty \\ 3 & 5 & \infty & 0 & 1 \\ 4 & 2 & \infty & \infty & 0 \end{array}$$

$\rightarrow$ destination

$\downarrow$ source

diagonal elements are (0) because no self loops are there

$$A^1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & 2 & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & \infty & 0 \end{array}$$

(1) as intermidiate node

→ 1'st Row and 1st Column is same as that of (A^0)

$A^1[2,3] = \min(A^0[2,3], A^0[2,1] + A^0[1,3])$
$= \min(2, 8+\infty) = 2$

$A^1[2,4] = \min(\infty, 8+7) = 15$

$A^1[3,2] = \min(\infty, 5+3) = 8$

$A^1[3,4] = \min(1, 5+7) = 1$

$A^1[4,2] = \min(\infty, 2+3) = 5$

$A^1[4,3] = \min(\infty, 2+\infty) = \infty$

$$A^2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 5 & 7 \\ 2 & 8 & 0 & 2 & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$

(2) as intermidiate node

→ 2nd Row and 2nd Column is same as that of (A^1)

$A^2[1,3] = \min(\infty, 3+2) = 5$

$A^2[1,4] = \min(7, 3+15) = 7$

$A^2[3,1] = \min(5, 8+8) = 5$

$A^2[3,4] = \min(1, 8+15) = 1$

$A^2[4,1] = \min(2, 5+8) = 2$

$A^2[4,3] = \min(\infty, 5+2) = 7$

$$A^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 5 & 6 \\ 2 & 7 & 0 & 2 & 3 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$

(3) as intermediate node //

→ 3rd Row and 3rd Column is same as that of (A^2)

$A^3[1,2] = \min(3, 5+8) = 3$

$A^3[1,4] = \min(7, 5+1) = 6$

$A^3[2,1] = \min(8, 2+5) = 7$

$A^3[2,4] = \min(15, 2+1) = 3$

$A^3[4,1] = \min(2, 7+5) = 2$

$A^3[4,2] = \min(5, 7+8) = 5$

$$A^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 5 & 6 \\ 2 & 5 & 0 & 2 & 3 \\ 3 & 3 & 6 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$

(4) as intermidiate node //

→ 4th Row and 4th Column is same as that of (A^3)

$A^4[1,2] = \min(3, 6+5) = 3$

$A^4[1,3] = \min(5, 6+7) = 5$

$A^4[2,1] = \min(7, 3+2) = 5$

$A^4[2,3] = \min(2, 3+7) = 2$

$A^4[3,1] = \min(5, 1+2) = 3$

$A^4[3,2] = \min(8, 1+5) = 6$

classmate

Date
Page

Travelling Salesman Problem :-

Problem :- A Salesman is given a set of cities, and the task is to find the shortest possible tour (path) that visits each city exactly once and returns to the starting city.

⇒ A bi-directional graph will be given where each node represent a city. Hence we need to find a closed loop which covers all nodes of the graph such that the total cost (Matrix portraying costs of edges will also be given) of edges is minimum.

[(ϕ) Actually represent null node at bottom of spanning tree.]

fromulae used :-

→ The cost from a node (i) to the starting node (1) is represented by :- (at the last move)

$$\boxed{g(i, \phi) = c_{i1}}$$

Similarly,

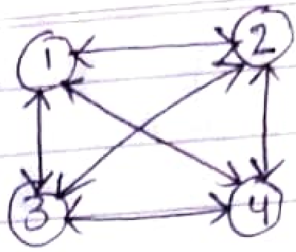
→ The cost from node (i) to set of remaining vertices is given by :-

$$\boxed{g(i, S) = \min_{j \in S} \{ c_{ij} + g(j, \underbrace{S - \{j\}}_{\text{Excluding}}) \}}$$

(S = remaining states/nodes to traverse) [recursive formula]

Ex :-

$$\text{Costs} = \begin{array}{c|c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \hline \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix} \end{array}$$



traverse the graph starting from node (1) such that the total cost of all edges of the path is minimum.

Sol:- we have,

$$g(i, S) = \min_{j \in S} \left(C_{ij} + S - \{j\} \right)$$

$\therefore$ if starting node is (1) $\Rightarrow i = 1$

$\therefore S = \{2,3,4\}$

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

$$\rightarrow g(2,\{3,4\}) = \min\{C_{23} + g(3,\{4\}), C_{24} + g(4,\{3\})\} \quad \text{—— (a)}$$

$$\rightarrow g(3,\{2,4\}) = \min\{C_{32} + g(2,\{4\}), C_{34} + g(4,\{2\})\} \quad \text{—— (b)}$$

$$\rightarrow g(4,\{2,3\}) = \min\{C_{42} + g(2,\{3\}), C_{43} + g(3,\{2\})\} \quad \text{—— (c)}$$

$$* \ g(3,\{4\}) = \min\{C_{34} + g(4,\phi)\} = \min\{12 + 8\} = 20 \quad \text{—— (i)}$$

$$\therefore \boxed{\begin{array}{l} g(4,\phi) = C_{41} = 8 \\ C_{34} = 12 \end{array}}$$

$$* \ g(4,\{3\}) = \min\{C_{43} + g(3,\phi)\} = \min\{9+6\} = 15 \quad \text{—— (ii)}$$

classmate

Date

Page

$\because \left[\begin{array}{l} g(3,\phi) = C_{31} = 6 \\ C_{43} = 9 \end{array}\right]$

$* \; g(2,\{4\}) = \min\{C_{24} + g(4,\phi)\} = \min\{8+10\}$

$\because \left[\begin{array}{l} g(4,\phi) = C_{41} = 8 \\ C_{24} = 10 \end{array}\right] \quad = 18$ —— (iii)

$* \; g(4,\{2\}) = \min\{C_{42} + g(2,\phi)\} = \min\{8+5\}$

$\because \left[\begin{array}{l} g(2,\phi) = C_{21} = 5 \\ C_{42} = 8 \end{array}\right] \quad = 13$ —— (iv)

$* \; g(2,\{3\}) = \min\{C_{23} + g(3,\phi)\} = \min\{9+6\}$

$\because \left[\begin{array}{l} g(3,\phi) = C_{31} = 6 \\ C_{23} = 9 \end{array}\right] \quad = 15$ —— (v)

$* \; g(3,\{2\}) = \min\{C_{32} + g(2,\phi)\} = \min\{13+5\}$

$\because \left[\begin{array}{l} g(2,\phi) = C_{21} = 5 \\ C_{32} = 13 \end{array}\right] \quad = 18$ —— (vi)

Put (v), (vi) in (c):-

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

$= \min(23, 27) = 23$ —— (A)

Put (iii), (iv) in (b):-

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

$= \min(31, 25) = 25$ —— (B)

Put (i), (ii) in (a):-

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

$= \min\{29, 25\} = 25$ —— (C)

Put (A), (B), (C) in (1), we get:-

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

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

$= 35$ //

$\therefore$ optimal path is :- $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$

whose cost is (35) //

classmate

Date

Page

Applications of Greedy method:-

1. Job Sequencing with deadlines
2. Knapsack Problem
3. Minimum Cost Spanning trees
4. Single Source Shortest Path Problem.

1. Job Sequencing with deadlines:-

It is a Classic Prioritized CPU Scheduling optimization Problem.

Problem:- Given a set of Jobs(tasks), each with a deadline and a profit associated with it. The task is to schedule (find sequence) the jobs, such that the total Profit is maximized while meeting the given deadlines.

Each Job takes one unit of time to Complete.

Terminologies:-

* Job:- A task to be completed within the deadline time, to gain its Corresponding Profit.
* Deadline:- Time limit under which a job is supposed to be Completed.
* Profit:- Some value got in return, after Completion of a job.

Greedy Approach of Solving this Problem is:-

(i) Set (order) the jobs based on their Profits, in descending order:-

This step ensures that we finally get maximum profits with the available time slots.

~~(ii) prepare a gantt chart and place the~~

(ii) Draw a gantt chart with (N) time slots.

A gantt chart is a 1-D table which portrays the sequence of execution of Jobs. Here (N) is the maximum deadline number given.

(iii) Assign each job to the latest possible time slot before its deadline.

This ensures that the jobs with smaller deadlines in the first (N) Jobs, are executed first. If the slot is already occupied, move to previous slot until an avialable slot is found.

⇒ Hence, Greedy method is prioritizing and selecting jobs with maximum profits and ensuring that all selected jobs get executed.

Ex:-

Jobs	J_1	J_2	J_3	J_4	J_5
Profits	20	15	10	5	1
Deadlines	2	2	1	3	3

find the optimal soln/sequence that gives max profits.

Sol:- By arrangin the Jobs in the decreasing order of thier profits, we get:-

Jobs	J_1	J_2	J_3	J_4	J_5
Profits	20	15	10	5	1
deadlines	2	2	1	3	3

(already in the order).

∴ Max deadline value is (3). We are allowed to have three time slots. i.e., only three Jobs can be Executed.

→ ~~Obviously we choose first three from above table but our task is to find the correct sequence which executes all three.~~

∴ By drawing a gantt chart with (3) time slots, we get.

J_2	J_1	J_4

0   1   2   3

→ for (J_1), deadline is (2) and (1-2) time slot is aviable, hence we assign it to (J_1)

→ for (J_2), deadline is (2) and (1-2) time slot is already occupied, hence we move to left. Since (0-1) is aviable, we assign it to (J_2)

→ for (J_3), the deadline is (1) and there is no suitable slot aviable, Hence we leave it.

Date
Page

→ for (J_4), deadline is (5). Hence can assign the last (2-3) time slot to it.

Hence, the optimal solution, i.e, the sequence which provides maximum profits is:- $J_2 \rightarrow J_1 \rightarrow J_4$

(and) Max Profit is:- $15+20+5=40$

The time Complexity is:- $O(n^2)$ [worst case]
[one (n) for sorting the table and other (n) for finding the correct sequence].

② Knapsack Problem (fractional)

Problem:- Given (N) items (objects) where each item has some weight (w_i) and Profit (P_i) associated with it, and also given a bag (Knapsack) with maximum Capacity (W).

The task is to Put the items into the Knapsack such that the sum of Profits associated with them is maximum [$\Sigma P_i x_i = max$] and sum of weights of the items is atmost (W).

[$\Sigma w_i x_i \leq W$]

max Profits.

classmate

Date

Page

here, we are multiplying the items with their profits and with their weights to get their actual profits and weights. This is because, the fractions are also allowed in this Problem [i.e., Item/object is divisible]

Constraint:-

* fractional Parts of objects are allowed. [objects are divisible].
* Items in final soln must give maximum Profits $\left[\sum p_i x_i = max\right]$
* weights of Items in final soln must be atmost (W) (max capacity of knapsack) $\left[\sum w_i x_i \leq W\right]$

Procedure for Solving using Greed Method:-

instead of selecting the item based on only profits or only weights, we consider Profit-weight ratio, which can be used to find optimal solution.

(i) Calculate the Profit-weight Ratio for each item/object.

(ii) Sort the table in decreasing order of P/w Ratios.

Our aim is to select items which has

higher P/w Ratio, first. [find (x_i) values]

(iii) Iterate through the Sorted items:-

* If adding the Entire item to the Knapsack doesn't exceed the weight Constraint [$\sum w_i x_i \leq W$], add the Entire item. [in this case, $x_i = 1$]

* If adding the Entire item does Exceed the weight Constraint, add a fraction of the item to fill up the remaining space in the Kapsack. [in this case, x_i = fraction]

(iv) Continue this until whole Knapsack is filled with (0) Capacity remaining. i.e, we are willing to increase the Profits more, by Considered fractions of items too, at the End, if some Capacity is left in the Knapsack.

Ex:-

Items(i)	1	2	3	4	5	6	7
Profits(P_i)	10	5	15	7	6	18	3
weight(w_i)	2	3	5	7	1	4	1

Knapsack Capacity (W) = 15

P/w Ratios are:-

$\frac{P_1}{w_1} = 5$, $\frac{P_2}{w_2} = 1.66$, $\frac{P_3}{w_3} = 3$, $\frac{P_4}{w_4} = 1$

$\frac{P_5}{w_5} = 6$, $\frac{P_6}{w_6} = 4.5$, $\frac{P_7}{w_7} = 3$

classmate

Date

Page

→ By arranging the items in decreasing order of their p/w Ratios, we get:-
∴ [we want max profit per unit weight]

Item (i)	5	1	6	3	7	2	4
Profit (P_i)	6	10	18	15	3	5	7
weight (w_i)	1	2	4	5	1	3	7
$\frac{P}{w}$	6	5	4.5	3	3	1.66	1
x_i	1	1	1	1	1	$\frac{2}{3}$	0

→ Iterate through the items:- (add item)
∴ max Capacity of Knapsack (W) = 15

* After adding item(5) ⇒ W = 15 − 1 = 14 (x_1=1)
* After adding item (1) ⇒ W = 14 − 2 = 12 (x_2=1)
* After adding item (6) ⇒ W = 12 − 4 = 8 (x_3=1)
* After adding item (3) ⇒ W = 8 − 5 = 3 (x_4=1)
* After adding item (7) ⇒ W = 3 − 1 = 2 (x_5=1)

Now, since next item (i = 2) has weight [w_i = (3)] and remaining Knapsack capacity is (W=2). Hence we cannot add the Entire object.

Therefore, we take a fraction of (i=2) and fill the remaining capacity (W=2)
∴ the fraction would be:-

$$\left[x_2 = \frac{W}{w_i} = \frac{2}{3}\right]$$

* After adding $\left(\frac{2}{3}\right)$ fraction of item(2)
⇒ W = 2 − 2 = 0.

* Since the max capacity (W) is filled, there is no space for item (4).
∴ (x_4 = 0)

∴ If we verify the max capacity constraint, we get the same (W) (initially)

i.e., $\sum x_i w_i = 1\times1 + 1\times2 + 1\times4 + 1\times5 + 1\times1 + \frac{2}{\cancel{3}}\times\cancel{3} + 0\times7$
$= 1+2+4+5+1+2+0 = 15 \leq W$
∴ Constraint satisfied

(and) the max Profit made is :-

$\sum x_i p_i = 1\times6 + 1\times10 + 1\times18 + 1\times15 + 1\times3 + \frac{2}{3}\times5 + 0\times7$
$= 6+10+18+15+3+3.33+0$
$= 55.33$ (max Profit)

(3) Minimum Cost Spanning Tree :-

- It is a subset of the edges of a connected (and) undirected graph that connects all the vertices together without any cycles and has the minimum possible total edge weight.
- * An undirected and connected graph can have more than one spanning trees.

i.e., The no. of spanning trees can be given by the formula : $^{|E|}C_{|V|-1}$

$|E|$ → no. of edges of given graph
$|V|$ → no. of vertices of given graph.

* If $|V|$ are the no. of vertices of a given graph, then all corresponding spanning trees of the graph will have exactly $(|V|-1)$ edges.

∴ If we verify the max capacity constraint, we get the same (W) (initially)

i.e, $\sum x_i w_i = 1\times1 + 1\times2 + 1\times4 + 1\times5 + 1\times1 + \frac{2}{3}\times3 + 0\times7$
$= 1+2+4+5+1+2+0 = 15 \le W$

∴ Constraint satisfied

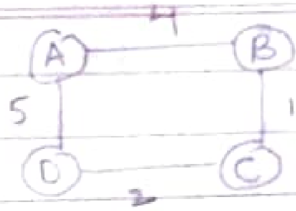
(and) the max profit made is :-

$\sum x_i P_i = 1\times6 + 1\times10 + 1\times18 + 1\times15 + 1\times3 + \frac{2}{3}\times5 + 0\times7$
$= 6+10+18+15+3+3.33+0$
$= 55.33$ (max Profit)

③ Minimum Cost Spanning Tree :-

* It is a subset of the edges of a connected (and) undirected graph that connects all the vertices together without any cycles and has the minimum possible total edge weight.
* An undirected and connected graph can have more than one spanning trees.
i.e., The no. of spanning trees can be given by the formula : ${}^{|E|}C_{|V|-1}$
  - $|E|$ → no. of edges of given graph
  - $|V|$ → no. of vertices of given graph.
* If $|V|$ are the no. of vertices of a given graph, then all corresponding spanning trees of the graph will have exactly $(|V|-1)$ edges.

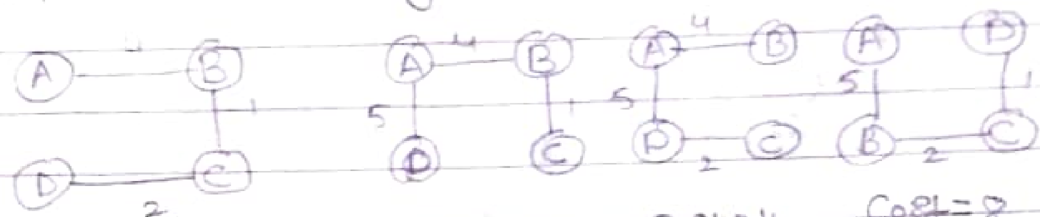
Ex:-



$|E| = 4$

$|V| = 4$

$\therefore$ possible spanning trees are:- ${}^4C_3 = 4$



cost = 7   cost = 9   cost = 11   cost = 8

$\therefore$ the spanning tree with cost (7) (minimum) is our required minimum cost spanning tree.

Problem:- Given a weighted, connected and underected graph G(V, E), find the spanning tree ~~___~~ such that the sum of weighted edges (total cost) is minimum.

* Since, for every graph we cannot draw each spanning tree and calculate all costs. Hence, there are Algorithms/methods which computes the required minimum spanning tree without having to know all the possible spanning trees.

* The Algorithms are:-
   ① Prim's Algorithm
   ② Krushkal's Algorithm.

① Prim's Algorithm:-

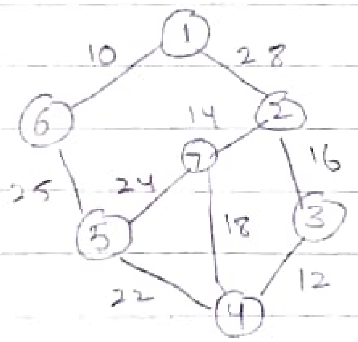
- It is a greedy algorithm which constructs the MCST by moving through the adjacent nodes with minimum edge weight.

(i) Choose a vertex to start with.

(ii) Select the adjacent edge with minimum weight and go to next node (vertex).
If a selected edge forms a closed loop, do not consider it and move to next option.

(iii) Repeat the step(2) until all nodes are traversed.
→ as a results there must be (n-1) edge for (n) nodes.

Ex:-

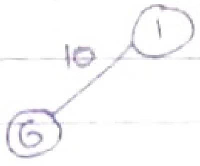


find MCST ?

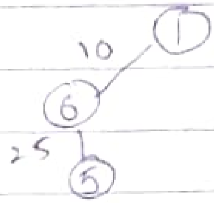
Sol: let (1) be starting vertex:-

1

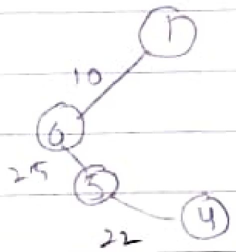
→ min edge is (10), so draw (10) edge and goto (6)



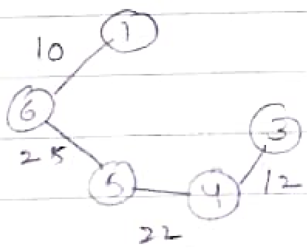
→ min Edge = 25



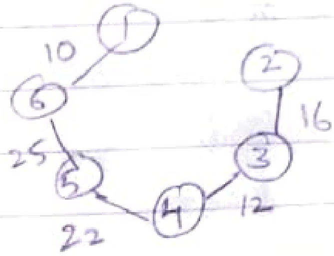
→ min Edge = 22



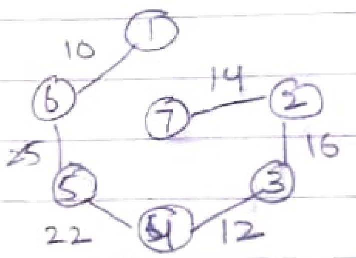
→ min Edge = 12



→ min Edg = 16



→ min Edge = 14



MCST

Cost = 99

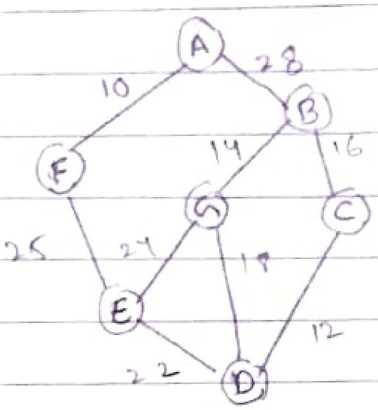
② **Krushkal's Algorithm :-**

It is a greedy algorithm which constructs the MCST by adding the edges in increased order of weights and reject those edges which forms a closed cycle/loop.

(i) Create a table of edges of given graph and place them in Ascending order.

(ii) Draw the nodes and Keep adding the edges in increasing order.
→ If addition of an Edge forms a Closed loop, dont Consider it.

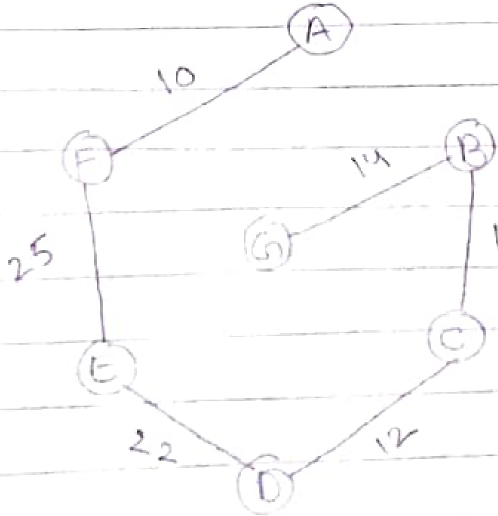
Ex:-



Edges	AB	BC	CD	DE	EF	FA	DG	GE	GB
weight	28	16	12	22	25	10	18	24	14

⇓ Sort

Edge	FA	CD	GB	BC	DG	D.E	GE	EF	AB
weight	10	12	14	16	18	22	24	25	28



DG X
GE X
AB X

Min cost spanning tree

∴ Cost = 10+25+22+12+16+14 = 99

classmate

Date

Page

(4) Single Source Shortest Path

* It is also known as Dijkstra's Algorithm

Problem:- Given a bidirectional, weighted graph and a source node to consider. The task is to find the shortest possible distances [min weights] from the source node to all remaining nodes of a graph.

(i) Start from source node and assign the distance (0) to it (∵ A→A distance is 0) [and] make the distances at every other non-visited nodes as (∞).

(ii) Update the distance of each neighbouring adjacent node based on the sum of distances from the source node.

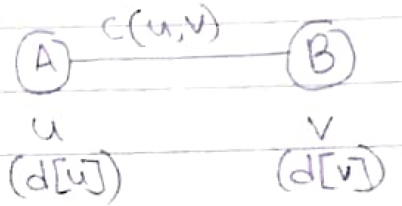
(iii) If any already visited node has the distance greater than the distance to current node, then replace the greater distance value with smaller distance value.

(iv) Repeat this process until all of the nodes are visited.

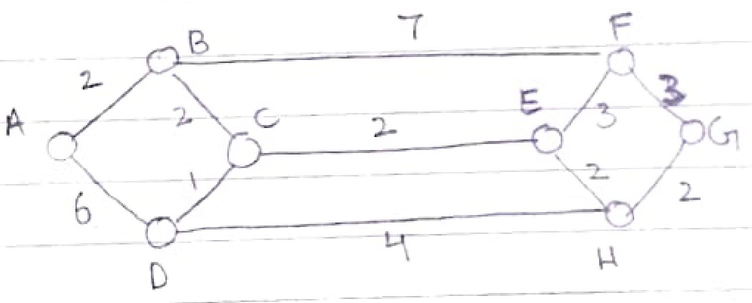
→ The resultant graph will have minimum weighted edges i.e., minimum distance from source node to every other node.

* The Actual formula/logic used at Every Relaxation Period (while updating distance values to adjacent nodes), is:-

$$\boxed{\begin{array}{l} \text{if}(d[u] + c(u,v) < d[v])\{ \\ \qquad d[v] = d(u) + c(u,v) \\ \} \end{array}}$$

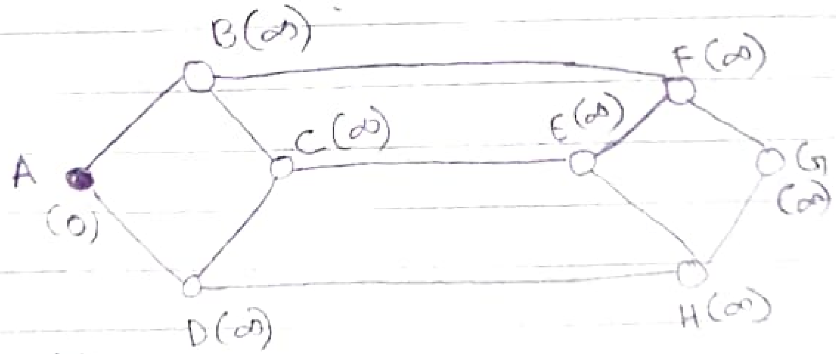


Ex:-

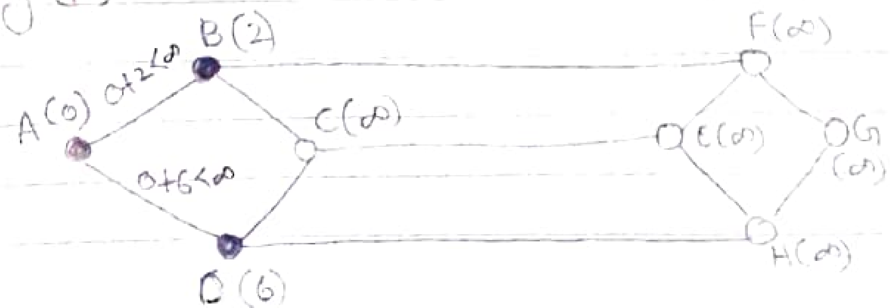


find the shortest paths from (A) to every other node.

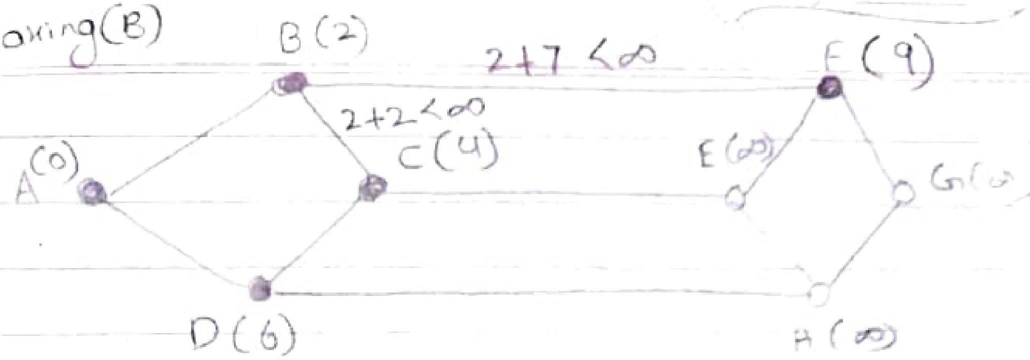
→



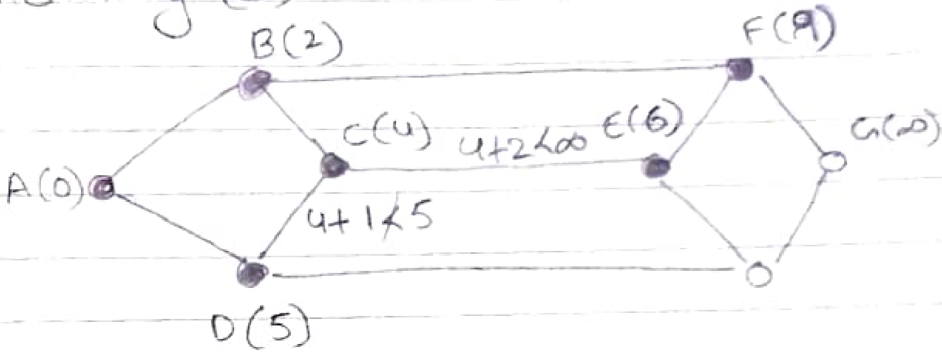
→ Relaxing (A)



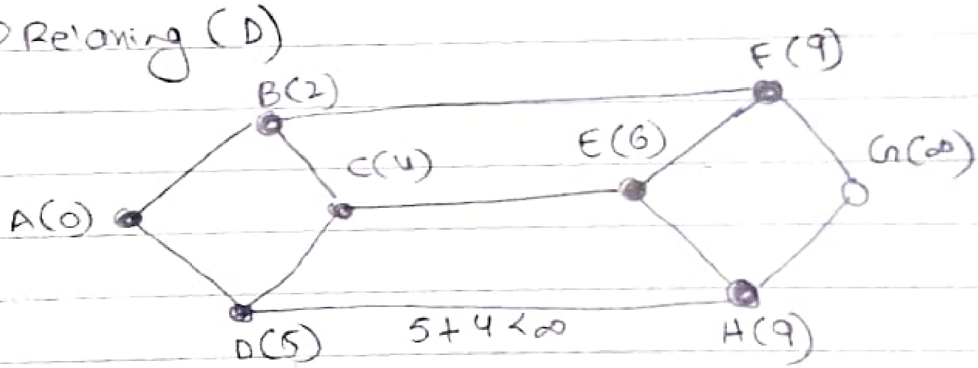
→ Relaxing (B)



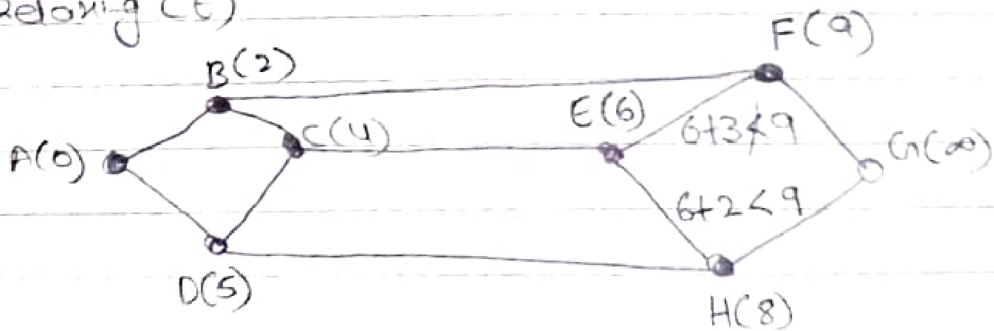
→ Relaxing (C)



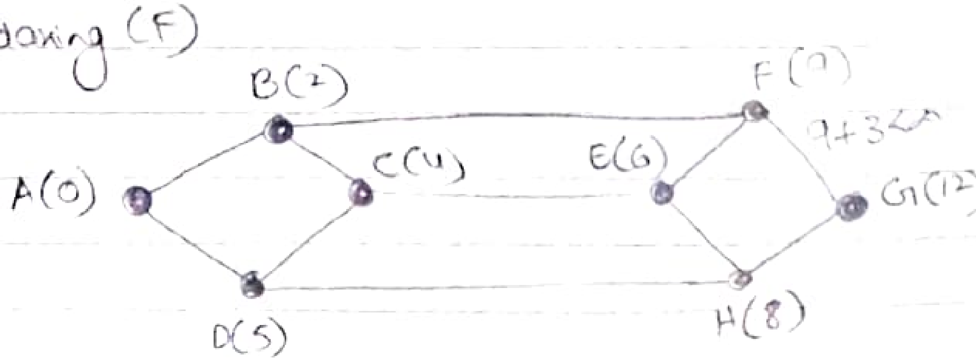
→ Relaxing (D)



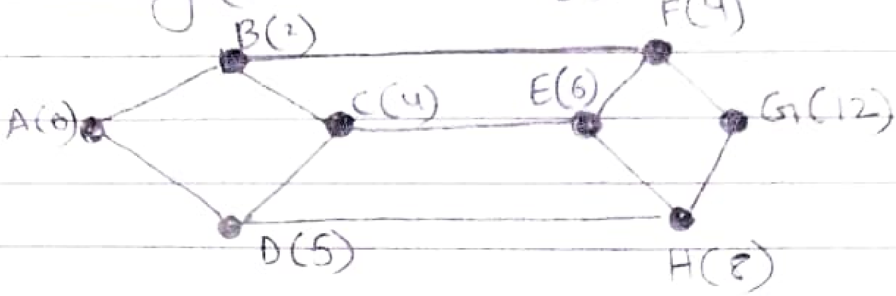
→ Relaxing (E)



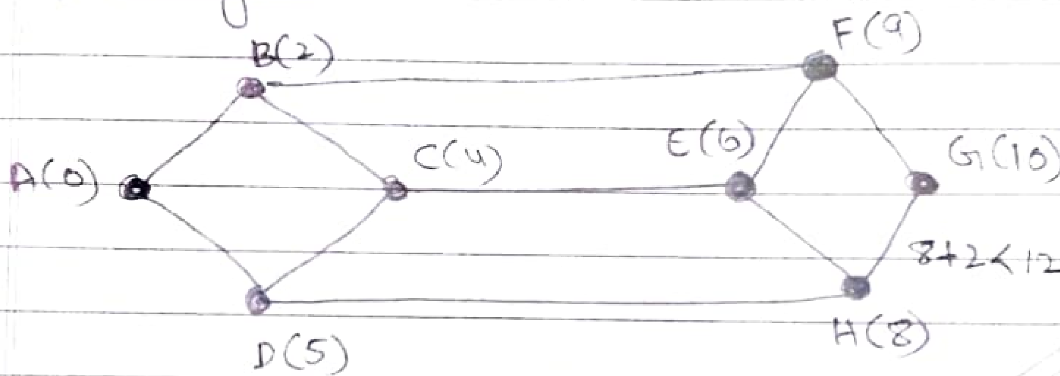
→ Relaxing (F)



→ Relaxing (G) (no change)



→ Relaxing (H)



Ex:- Shortest path from (A) to (G) is (10)
Shortest path from (A) to (H) is (8)
Shortest path from (A) to (D) is (5)