Dynamic Programming :-

Dynamic Programming is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a given problem's solution to solutions of smaller subproblems.

→ Computing Binomial Coefficient :-

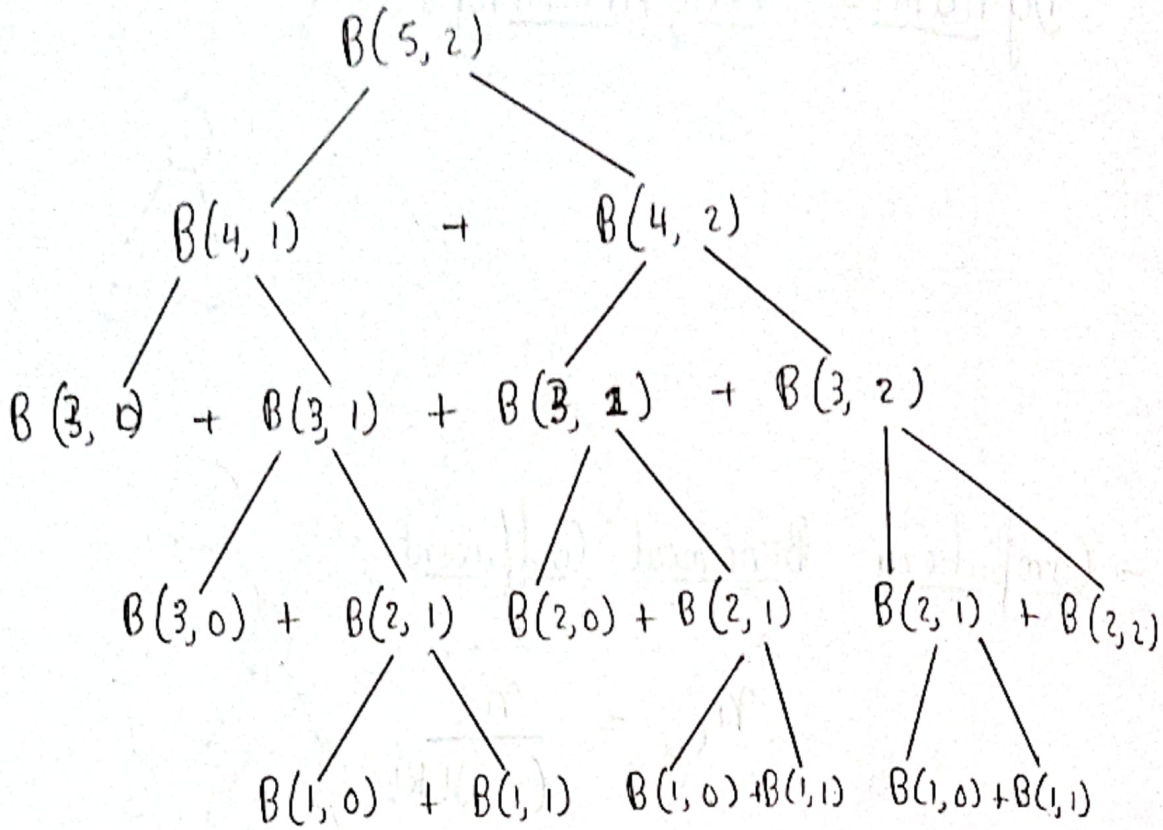
$$n_{C_K} = \frac{n!}{(n-K)!\,K!}$$

$$C(n, 0) = n_{C_0} = 1$$

$$C(n, n) = n_{C_n} = 1$$

$$C(n, K) = {}^{n-1}C_{K-1} + {}^{n-1}C_{K}$$

```
int binocoeff (n, k)
{
    if (k==0 || k==n)
        return 1;
    Else
        return (binocoeff (n-1, k-1) + binocoeff (n-1, k))
}
```

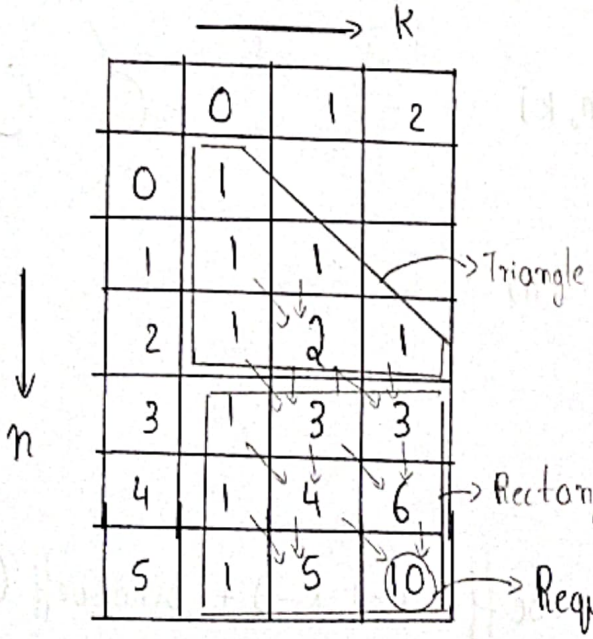


Number of addition : 9

Here, there are some overlapping problem i.e computed more than once (Ex:- B(3,1), B(2,1) etc..)

→ Converting to Dynamic Programming:-

Compute C(5,2)



Number of additions = 7

Algorithm :-

```
Binomial (n, K)
// Input :- A pair of non-negative integers n>K>0
// Output :- The value of C(n, K)
for i ← 0 to n do
    for j ← 0 to min(i, K) do
        if (j = 0 @ j = i)
            C[i,j] ← 1
        Else
            C[i, j] ← C[i-1, j-1] + C[i-1, j]
    end for
end for
return C[n, K]
```

Analysis :-

» Basic Operation :- Addition

Triangle Part ↗ ↗ Rectangle Part

$$A(n,K) = \sum_{i=0}^{K} \sum_{j=0}^{i} 1 + \sum_{i=K+1}^{n} \sum_{j=0}^{K} 1$$

$$= \sum_{i=0}^{K} (i - i + 1) + \sum_{i=K+1}^{n} K - i + 1$$

$$= \sum_{i=0}^{K} i + K \sum_{i=K+1}^{n} 1$$

$$= \frac{K(K+1)}{2} + K(n-K)$$

$\therefore A(n,K) \in \theta(nK)$

Space Complexity $= \theta((n+1)(K+1))$

⇒ Compute C(6,3) using Dynamic Programming :-

→ K

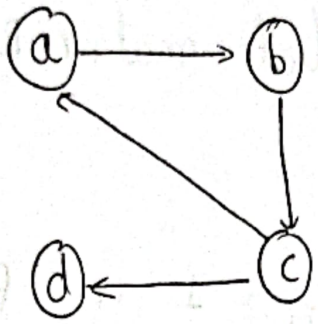
	0	1	2	3
0	1			
1	1	1		
2	1	2	1	
3	1	3	3	1
4	1	4	6	4
5	1	5	10	10
6	1	6	15	20

n ↓

→ C(6,3) = 20

⇒ Warshall's Algorithm :-

- Used to find Transitive Closure
- Applicable on directed graphs.
- Takes adjacency matrix as input



	a	b	c	d
a	0	1	0	0
b	0	0	1	0
c	1	0	0	1
d	0	0	0	0

Adjacency Matrix

→ Transitive Closure Matrix :-
(Reachability matrix)

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	1	1	1	1
d	0	0	0	0

Warshall algorithm generates a series of matrices to generate the Transitive Closure matrix

$R^{(0)}, R^{(1)}, R^{(2)} \ldots\ldots R^{(n)}$

$R^{(0)}$ → Vertices reachable without any intermediate vertices i.e adjacency matrix

$R^{(1)}$ → Vertices reachable by using 1^{st} node

$R^{(2)}$ → vertices reachable using 1^{st}, 2^{nd} node

$R^{(n)}$ → vertices reachable using 1^{st}, 2^{nd}, ... n^{th} node

We compute each of intermediate matrix $R^{(k)}$ using immediate predecessor $R^{(k-1)}$

(x) i.e if for any i,j if $r_{ij} = 1$ in $R^{(k-1)}$ then $r_{ij} = 1$ in $R^{(k)}$ also

(or)

(x) $\forall i,j$, $r_{ij} = 1$ in $R^{(k)}$, if $r_{ik}^{(k-1)} = 1$ and $r_{kj}^{(k-1)} = 1$

$R^{(0)} \Rightarrow$

	a	b	c	d
a	0	1	0	0
b	0	0	1	0
c	1	0	0	1
d	0	0	0	0

$R^{(1)} \Rightarrow$

	a	b	c	d
a	0	1	0	0
b	0	0	1	0
c	1	(1)	0	1
d	0	0	0	0

$R^{(2)} \Rightarrow$

	a	b	c	d
a	0	1	(1)	0
b	0	0	1	0
c	1	1	0	1
d	0	0	0	0

$R^{(3)} \Rightarrow$

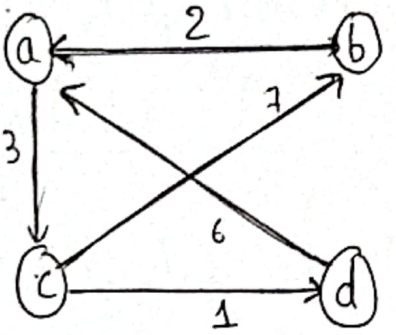
	a	b	c	d
a	(1)	1	1	(1)
b	(1)	(1)	1	(1)
c	1	1	(1)	1
d	0	0	0	0

$R^{(4)} \Rightarrow$

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	1	1	1	1
d	0	0	0	0

→ Floyd's Algorithm :-

- For finding All-Pair Shortest Path
- Applicable Directed / Undirected Graphs
- The matrix used is called Weight matrix.



Weight Matrix

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

→ Shortest-Path Matrix:-

Distance Matrix

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	7	7	0	1
d	6	16	9	0

$$d_{i,j}^{(K)} = \text{Min}\left(d_{ij}^{(K-1)},\ d_{iK}^{(K-1)} + d_{Kj}^{(K-1)}\right)$$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

$D^{(0)}$

	a	b	c	d
a	0	∞	3	∞
b	2	0	(5)	∞
c	∞	7	0	1
d	6	∞	(9)	0

$D^{(1)}$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	(9)	7	0	1
d	6	∞	9	0

$D^{(2)}$

	a	b	c	d
a	0	(10)	3	(4)
b	2	0	5	(6)
c	9	7	0	1
d	6	(16)	9	0

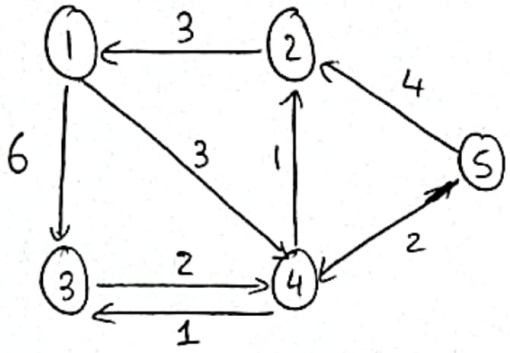
$D^{(3)}$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	(7)	7	0	1
d	6	16	9	0

$D^{(4)}$

⇒ Final distance Matrix

Ex :-



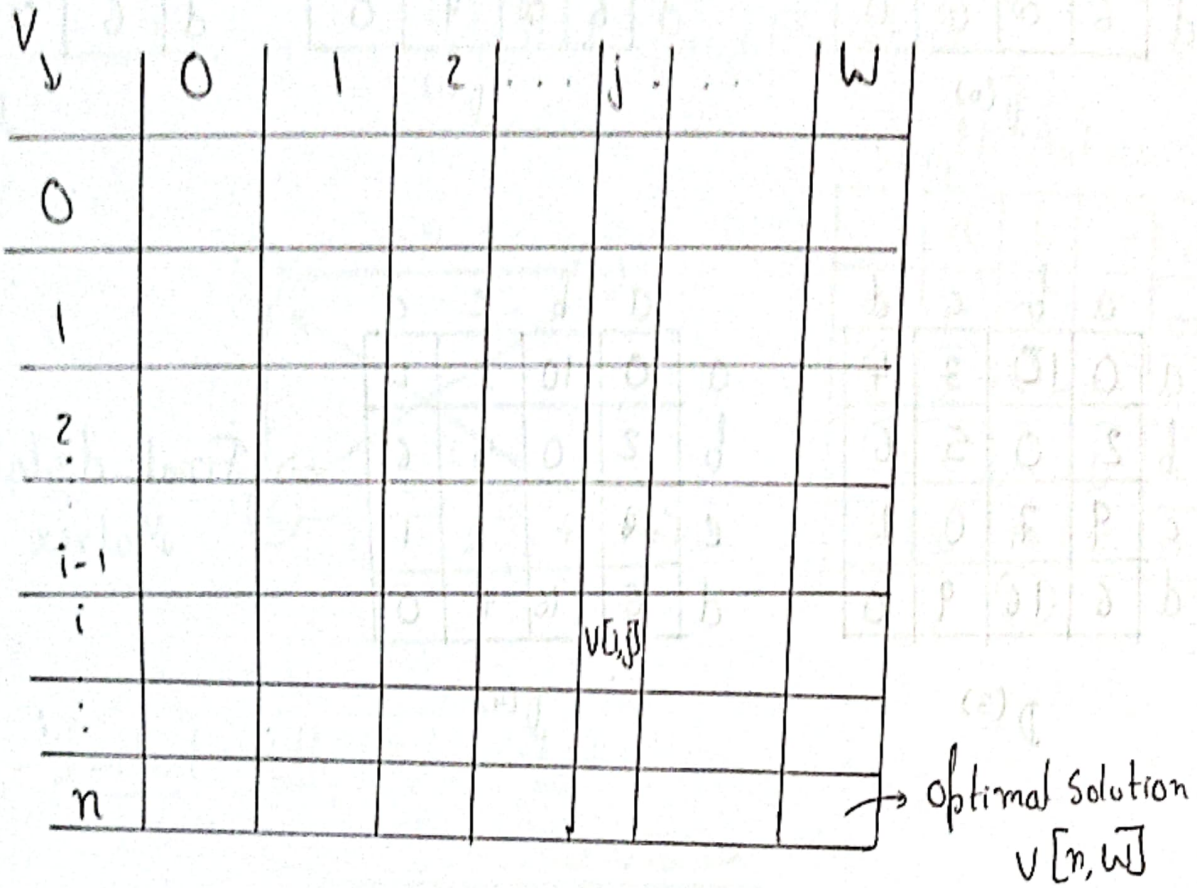
Distance Matrix

	1	2	3	4	5
1	0	4	4	3	∞
2	3	0	7	6	∞
3	6	3	0	2	∞
4	4	1	1	0	∞
5	6	3	3	2	0

⇒ Knapsack Using Dynamic Programming :-

(0-1 Knapsack)

Given, n → items, W → capacity, w_i → weights of item, v_i → values of



$V[i, j] = V[i-1, j]$ when $j - w_i < 0$

$V[i, j] = \max\{V[i-1, j],\ v_i + V[i-1, j-w_i]\}$ when $j - w_i \geq 0$

Initial Condition :- $V[i, 0] = 0 \quad \forall\ i \geq 0$

$V[0, j] = 0 \quad \forall\ j \geq 0$

Question :-

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	1	15

$n = 4, \quad W = 5$

Soln :-

⊘	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	15	25	27	37	45

→ Optimal solution = 45

↓

Items Contributing = 2, 3, 4

Subset of items Contributing = {2, 3, 4}

1	2	3	4
0	1	1	1

Question-2 :-

Item	Weight	Value
1	3	25
2	2	20
3	1	15
4	4	40
5	5	50

$n = 5 \quad W = 6$

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	25	25	25	25
2	0	0	20	25	25	45	45
3	0	15	20	35	40	45	60
4	0	15	20	35	40	55	60
5	0	15	20	35	40	55	65

→ Optimal Solution = 65

Contributing Subset = {3, 5} → Composition

Items :- 1 2 3 4 5
0 0 1 0 1

0 → The item is not included

1 → Item is included.

=> Algorithm :-

Knapsack_bottom_up(w[1... n], V[1... n], W)

//

for i ← 0 to n do

    V[i, 0] ← 0

for j ← 0 to W do

    V[0, j] ← 0

for i ← 0 to n do

    for j ← 0 to W do

        if j - w[i] ≥ 0

            V[i, j] ← Max{V[i, j], V[i] + V[i-1, j-w]}

        Else

            V[i, j] ← V[i-1, j]

return V and V[n, w]

→ To find Composition :-

```
numofitem ← 0
subset[ ]

j ← w
for i ← n downto 1 do
    if { V[i, j] ≠ V[i-1, j]
        numofitem ← numofitem + 1
        subset[numofitem] ← i
        j ← j - w[i]
end for
```

Note :- In Knapsack using Bottom-Up Approach (Iterative)

⊗ All the subproblems are solved and solved only once
But, all the solutions of subproblems do not contribute for getting solution for given problem.

→ In Knapsack using Top-Down Approach (Recursive) :-

⊗ All the subproblems are solved and solved more than once
But, all the solutions of subproblems contribute for getting solution for given problem.

→ Knapsack using Memory Functions :-

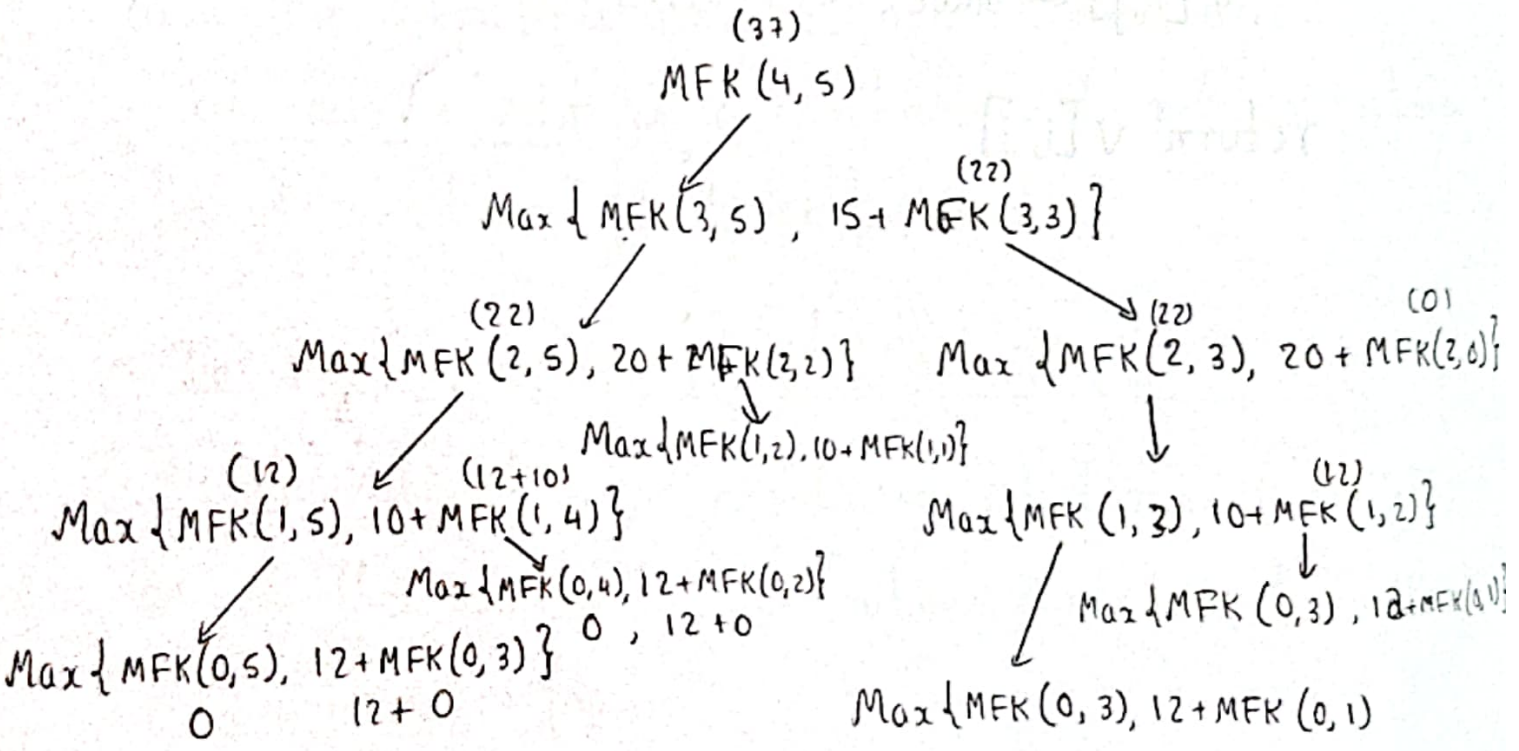
```
MFKnapsack (i, j)
// Input :- Integer i, which is the number of items
            Integer j, which indicates the Knapsack Capacity
// Output :- The value of an optimal solution (subset) of the first
             'i' items.
if V[i, j] = -1
    if j - w[i] ≥ 0
        value ← Max{ MFKnapsack (i-1, j), V[i] + MFKnapsack (i-1, j-w[i]) }
    Else
        value ← MFKnapsack (i-1, j)
    V[i, j] ← value
return V[i, j]
```

Question :-

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

W = 5

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	+12	12	12
2	0	-1	12	22	-1	22
3	0	-1	-1	22	-1	32
4	0	-1	-1	-1	-1	37



=> Greedy Technique :-

The solution must be

Feasible

Optimal

Irrevokable

-> Change Making Problem :-

Given some coins, and an amount, select the least number of coins to make the amount.

Exc 1 :- $d_1 = 25$ (quonta) $d_2 = 10$ (dime) $d_3 = 5$ (nickle) $d_4 = 1$ penny (1)

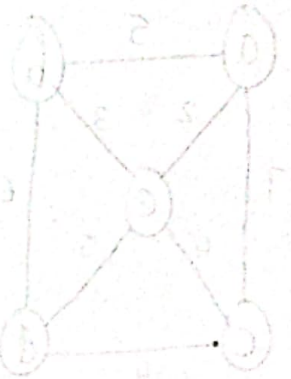
Amount :- 48 cents.

$d_1 \rightarrow 1 \Rightarrow 48 - 25 = 23$

$d_2 \rightarrow 2 \Rightarrow 23 - 20 = 3$

$d_4 \rightarrow 3 \Rightarrow 3 - 3 \times 1 = 0$

Total :- 6 Coins



Ex 2 :- Amount :- 10 dollars

Denomination :- $d_1 \to 7$ $d_2 \to 5$ $d_3 \to 1$

For this instance, greedy technique does not provide optimal solution

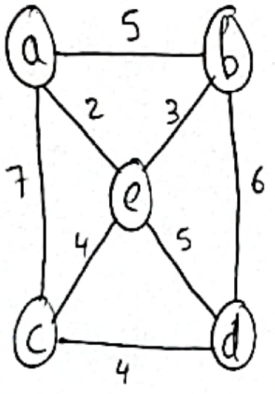
i.e $d_1 \to 1$ & $d_2 \to 3$ $\Rightarrow$ 4 coins

But, optimal solution is

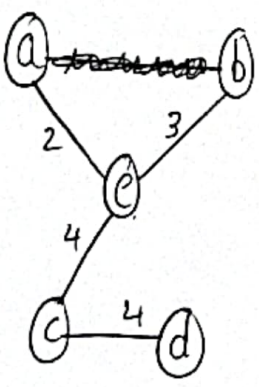
$d_2 \to 2$ $\Rightarrow$ 2 coins

Prim's Algorithm to find Minimum Cost Spanning Tree:-

Example :-



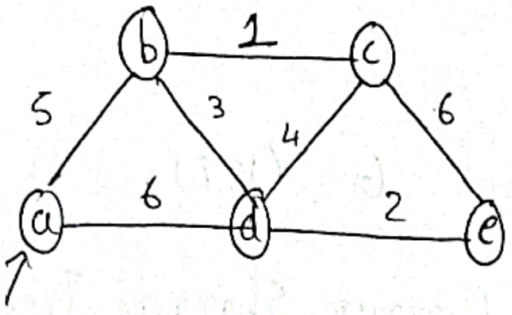
Tree Vertices	Remaining Vertices (Priority Queue)	Fringe vertex
a(–,–)	b(a,5) e(a,2) c(a,7) d(–,∞)	
e(a,2)	b(a,5) b(e,3) c(a,7) c(e,4) d(e,5)	
b(e,3)	c(a,7) c(e,4) d(e,5) d(b,6)	
c(e,4)	d(c,4) d(e,5) d(b,6)	
d(c,4)		



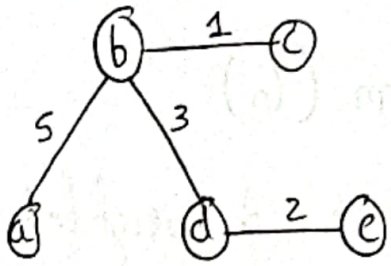
$\Rightarrow \{(a,e)(e,c)(c,d)(e,b)\}$

Edge Set forming Minimum Spanning Tree.

Example :- 2 :-



$\Rightarrow$



Minimum Spanning Tree

Tree Vertices	Remaining Vertices (Priority Queue)
a(–, –)	b(a, 5), d(a, 6), c(–, ∞), e(–, ∞)
b(a, 5)	c(b, 1), d(b, 3), d(a, 6), e(–, ∞)
c(b, 1)	d(b, 3) d(a, 6) d(c, 4) e(6, 6)
d(b, 3)	e(d, 2) e(c, 6)
e(d, 2)	

Minimum Spanning Tree Edge Set

$= \{ (a,b) (b,c) (b,d) (d,e) \}$

=> Algorithm :-

Prim (G)

// I/P :- A weighted connected graph $G = (V, E)$

// o/p :- Set of edges composing a Minimum Spanning Tree of G

$V_T \leftarrow \{V_0\}$

$E_T \leftarrow \emptyset$

for $i \leftarrow 1$ to $|V| - 1$ do

Find a minimum weight Edge $e^* = (v^*, u^*)$ among all the edges (v, u) such that 'v' is in V_T and 'u' is in $V - V_T$

$V_T \leftarrow V_T \cup \{u^*\}$

$E_T \leftarrow E_T \cup \{e^*\}$

end for

return E_T

⇒ Kruskal's Algorithm :-

Kruskal(G)

// I/P :- A weighted corrected graph, $G = (V, E)$

// O/P :- E_T, the set of edges composing of Minimum Spanning Tree of G

Sort Edges in Ascending order of their weights.

$w(e_1) \leq \ldots w(e_{|E|})$

$E_T \leftarrow \emptyset$

ecounter $\leftarrow 0$

$K \leftarrow 0$

while ecounter $< |V|-1$ do

    $K \leftarrow K+1$

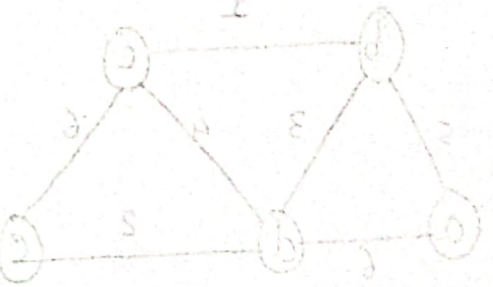
    if $E_T \cup \{e_{ik}\}$ is cyclic

        $E_T \leftarrow E_T \cup \{e_{ik}\}$

    ecounter $\leftarrow$ ecounter + 1

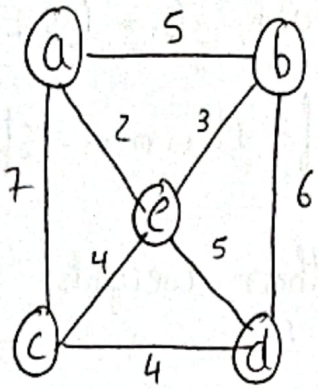
end while

return E_T

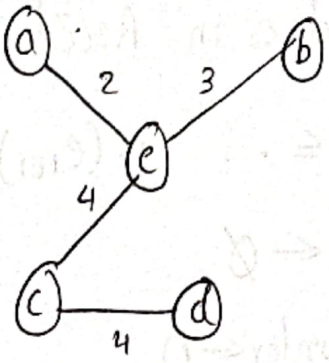


⇒ Kruskal's Algorithm ~~algorithm~~ Examples :-

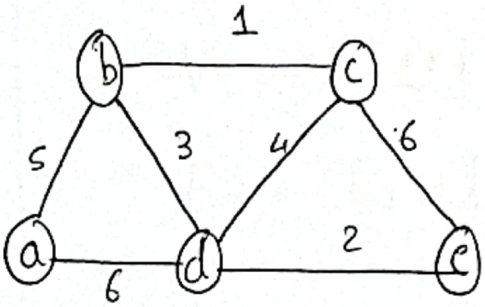
Ex :- (1)



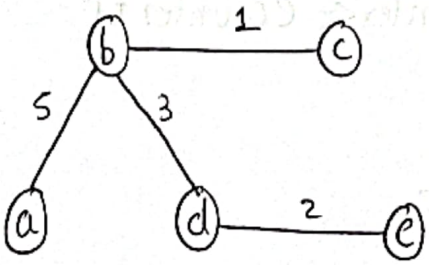
ae	bc	ec	cd	ab	ed	bd	ac
2	3	4	4	5	5	6	7
✓	✓	✓	✓	x	x	x	x



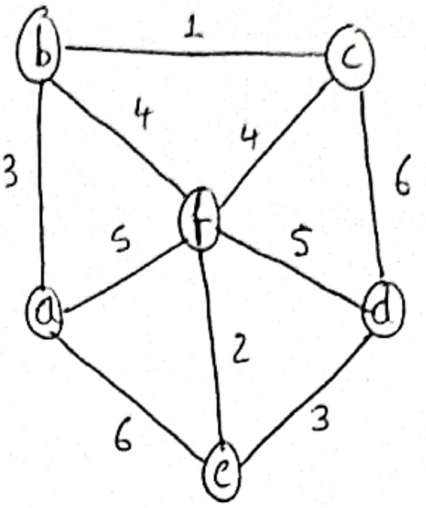
Ex :- (2)



bc	de	bd	dc	ab	ad	ce
1	2	3	4	5	6	6
✓	✓	✓	x	✓	x	x



Ex - 3 :-



bc	fe	de	ab	bf	cf	af	fd	cd	ae
1	2	3	3	4	4	5	5	6	6
✓	✓	✓	✓	✓	x	x	x	x	x