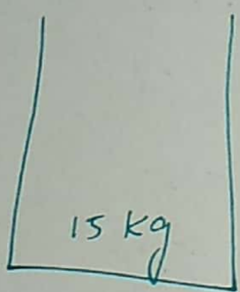
Knapsack Problem

Fractional Knapsack (Greedy)

0/1 Knapsack (Dynamic programming)

Fractional Knapsack Problem



Problem statement:

How you will select the items so that you will get maximum profit?

Maximization problem

Max Profit

objects:	1	2	3	4	5	6	7
Profit (P):	5	10	15	7	8	9	4
weight (w):	1	3	5	4	1	3	2

Objects	Profit (P)	Weight (w)	Remaining weight
3	15	5	15-5=10
2	10	3	10-3=7
6	9	3	7-3=4
5	8	1	4-1=3
4	$7 \times \frac{3}{4} = 5.25$	3	3-3=0

Total Profit = 47.25

Minimum weight

Objects	Profit (P)	Weight (w)	Remaining weight
1	5	1	15 - 4 = 14
5	8	1	14 - 1 = 13
7	4	2	13 - 2 = 11
2	10	3	11 - 3 = 8
6	9	3	8 - 3 = 5
4	7	4	$5 - 4 = \boxed{1}$
3	$15 \times \frac{1}{5} = 3$	1	$1 - 1 = \boxed{0}$

Total Profit = 46

Maximum P/w Ratio

Objects :	1	2	3	4	5	6	7
Profit (P) :	5	10	15	7	8	9	4
Weight (w) :	1	3	5	4	1	3	2
(P/w) :	5	3.3	3	1.75	8	3	2

Objects	Profit (P)	Weight (w)	Remaining weight
5	8	1	15-1 = 14
1	5	1	14-1 = 13
2	10	3	13-3 = 10
3	15	5	10-5 = 5
6	9	3	5-3 = 2
7	4	2	2-2 = 0

$\boxed{\text{Total Profit} = 51}$

Maximum profit is attained by Profit/Ratio. This is the fractional Knapsack which is solved by Greedy method.

Algorithm

fo. Algorithm: Greedy-Fractional-Knapsack ($w[1 \cdots n]$, $p[1 \cdots n]$, W)

```
for i = 1 to n
do x[i] = 0
weight = 0
for i = 1 to n
   if weight + w[i] ≤ W then
   x[i] = 1
   weight = weight + w[i]
   else
     x[i] = (W - weight) / w[i]
     weight = W
     break
return x
```

Analysis

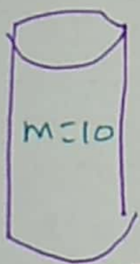
If the provided items are already sorted in descending order of $\frac{p_i}{w_i}$, then the whileloop takes time $O(n)$. Therefore the total time including the sort is $O(n \log n)$.

Example

Weight (W_i)	7	3	4	5
Profit (P_i)	42	12	40	25

Bag capacity (m) = 10 $\{\text{constraint} - \sum x_i w_i <= m\}$

P_i/w_i	6	4	10	5
x_i $(0 < x_i <= 1)$	6/7	0	1	0



$10 - 4 = 6$

$6 - 6 = 0$

$$\boxed{\begin{aligned}\sum x_i w_i &= (6/7)^* 7 + 0^* 3 + 1^* 4 + 0^* 5 = 10 \\ \sum x_i P_i &= (6/7)^* 42 + 0^* 12 + 1^* 40 + 0^* 25 = 76\end{aligned}}$$

Knapsack Using Brute force Technique
(Exhaustive search)

Capacity (W) = 10

S.No	1	2	3	4
Weight (w_i)	7	3	4	5
Value (v_i)	\$42	\$12	\$40	\$25

Subset	Total weight	Total value
φ	0	\$0
{1}	7	\$42
{2}	3	\$12
{3}	4	\$40
{4}	5	\$25
{1,2}	10	\$54
{1,3}	11	Not feasible
{1,4}	12	Not feasible
{2,3}	7	\$52
{2,4}	8	\$37
{3,4}	9	\$65
{1,2,3}	14	Not feasible
{1,2,4}	15	Not feasible
{1,3,4}	16	Not feasible
{2,3,4}	12	Not feasible
{1,2,3,4}	19	Not feasible

Total Items (n) = 4

Time Complexity $O(n) = 2^4 = 16.$

$O(n) = 2^n$ //.

0/1 Knapsack Problem using Dynamic Programming.

$m = 8 \quad P = \{1, 2, 5, 6\}$

$n = 4 \quad w = \{2, 3, 4, 5\}$

V

Pi	wi		0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5	6	6	7	8

$V[i, w] = \max\{V[i-1, w], V[i-1, w-w[i]] + P[i]\}$

$x_1 \quad x_2 \quad x_3 \quad x_4$

$\{0 \quad 1 \quad 0 \quad 1\}$

$8 - 6 = 2$ profit

$2 - 2 = 0$

$$V[4,1] = \max\{V[3,1],\ \underbrace{V[3, 1-5] + 6}_{3,\ -4}\}$$

undefined

Note

i - row

w - column

V[i-1] - prev row value

w[i] - wt of an object

P[i] - profit of an object

Algorithm

Algorithm DP_Binary_Knapsack (V, W, M)

// Description: Solve binary knapsack problem using dynamic programming

// Input : Set of items X, set of weight W, profit of items V and knapsack capacity M

// Output: Array V, which holds the solution of problem

Scanned with CamScanner

```
for i ← 1 to n do
  V [i, 0] ← 0
end
for i ← 1 to M do
  V [0, i] ← 0
end
for i ← 1 to n do
  for j ← 0 to M do
    if w[i] ≤ j then
      V [i, j] ← max { V [i-1, j], v [i] +
                         V [i-1, j - w[i]] }
    else
      V [i, j] ←  ~~v [i-1],~~  V [i-1, j]   // w[i] > j
    end
  end
end
```

Algorithm Trace_Knapsack (w, v, M)

// w is array of weight of n items
// v is array of value of n items
// M is the knapsack capacity

```
SW ← { }
SP ← { }
i ← n
j ← M
while (j>0) do
   if (v[i, j] == v[[i-1, j] then
    i ← i-1
   else
    v [i, j] ← v [i, j] - vi
    j ← j - w[i]
    SW ← SW + W[i]
    SP ← SP + V [i]
 end
end.
```

Running time complexity using DP can be solved by n (number of items) & M (capacity of knapsack)

$\boxed{O(nM) \text{ is the time complexity}}$

Example

$$V[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ V[i-1,j] & \text{if } j < w_i \\ \max\{V[i-1,j],\ v_i + V[i-1, j-w_i]\} & \text{if } j \ge w \end{cases}$$

j →

	Item detail		0	1	2	3	4	5
i = 0			0	0	0	0	0	0
i = 1	$w_1 = 2$	$v_1 = 3$	0	0				
i = 2	$w_2 = 3$	$v_2 = 4$	0					
i = 3	$w_3 = 4$	$v_3 = 5$	0					
i = 4	$w_4 = 5$	$v_4 = 6$	0					

Filling first column $j = 1$

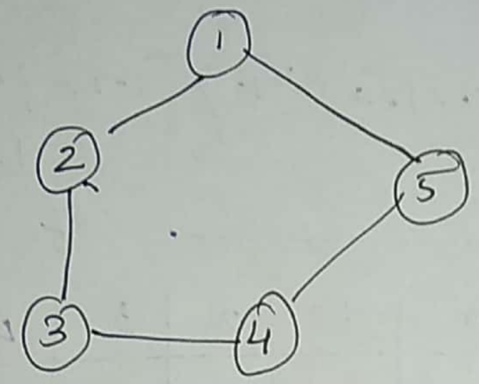
$V[1,1] \Rightarrow i = 1, j = 1, w_i = w_1 = 2$

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

$V[1,1] = V[0,1] = 0$

Minimum Spanning Tree

What is a spanning Tree?



$G = (V, E)$

G - Graph

V - Vertices

E - Edges

- Spanning Tree of G?

$G' = (V', E')$

$$\boxed{\begin{array}{l} V' = V \\ E' \in E \\ E' = |V| - 1 \end{array}}$$

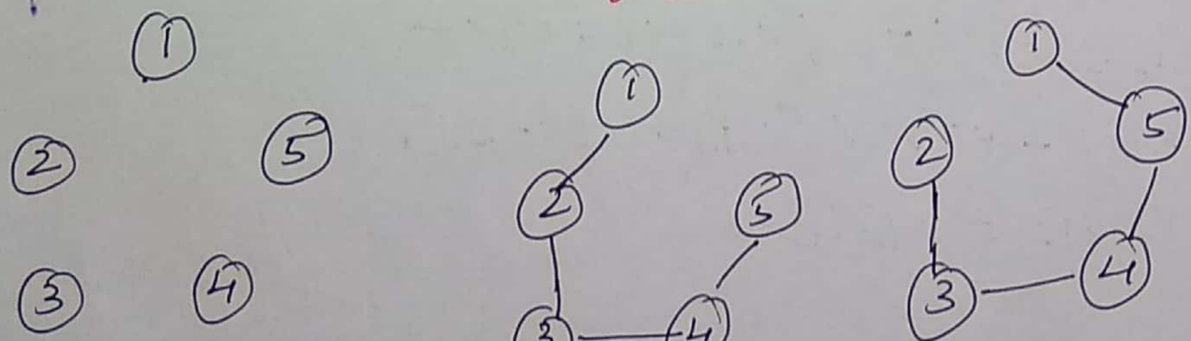
Conditions for spanning tree

-> Same No: of vertices

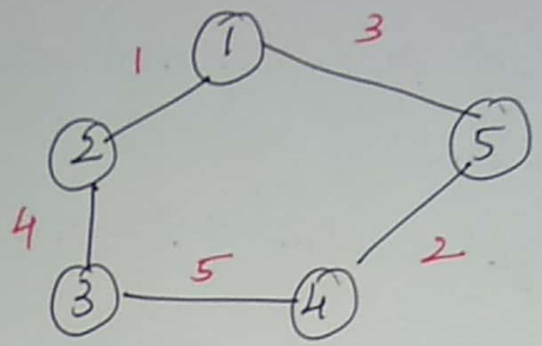
-> No: of edges would be no: of vertices - 1

-> A graph can have more than one spanning tree.

Example

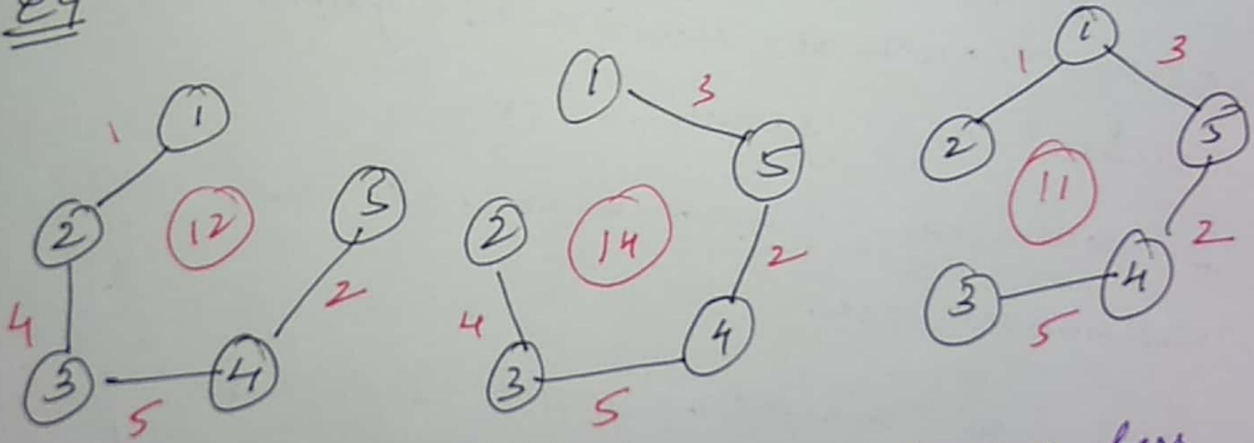


What is Minimum spanning Tree?

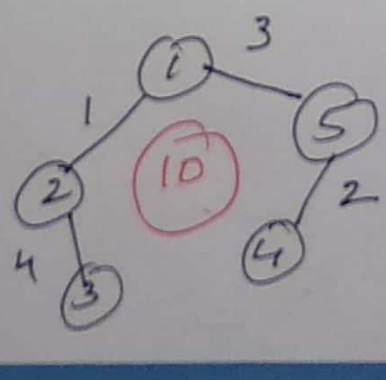


Suppose the Graph contains weights, the minimum cost of the total weights is the minimum spanning tree

Eg



Consider the edge which is having less cost



→ minimum //.

Properties of Spanning Tree

→ Removing one edge from spanning tree will make it disconnected

→ Adding one edge to the ST will create a loop

→ If each edge has distinct weight then there will be only one & unique MST

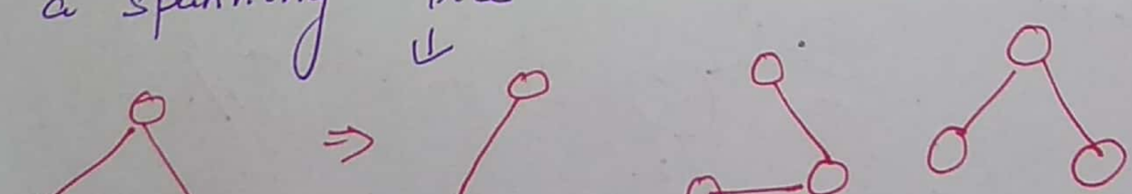
→ A complete undirected graph can have n^{n-2} no: of spanning tree

→ Every connected undirected graph has atleast one spanning tree

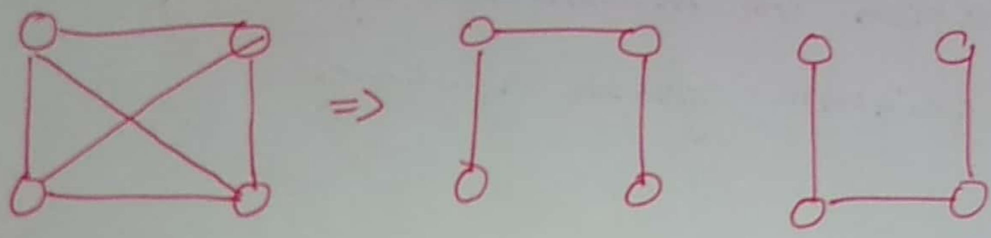
→ Disconnected graph does not have any spanning tree

→ From a complete graph by removing max $(e-n+1)$ edges we can construct a spanning Tree

↓



$\cancel{3}-\cancel{3}+1 = 1$



$n^{n-2} = 16$

$6 - 4 + 1$
$= 2 + 1 = 3$

Prim's Algorithm

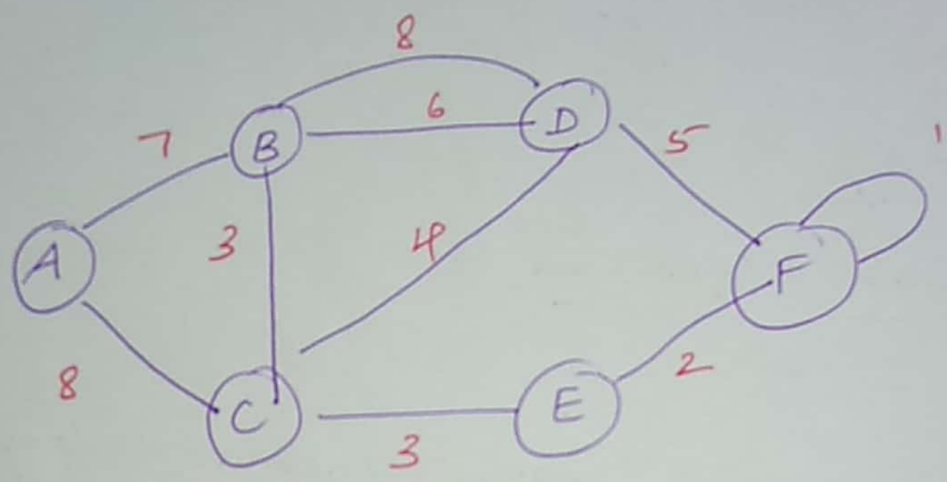
Steps

⇒ 1. First check if Graph contains a loop edge or parallel edge

⇒ 2. If it contains loop edge, remove them

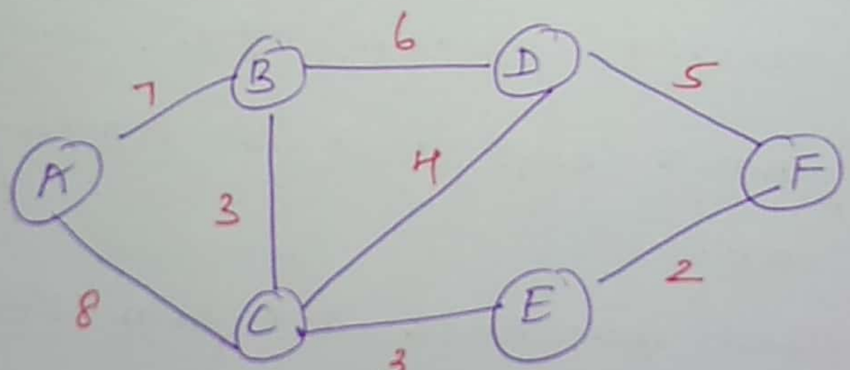
⇒ 3. If it contains parallel edge, remove the parallel edge that is having more cost.

⇒ 4. Now select any node and from that find the edge having minimum const and in all steps we have to check the previous steps edges that are not selected for minimum cost. Follow the steps till vertices in minimum spanning tree and original graph is same.



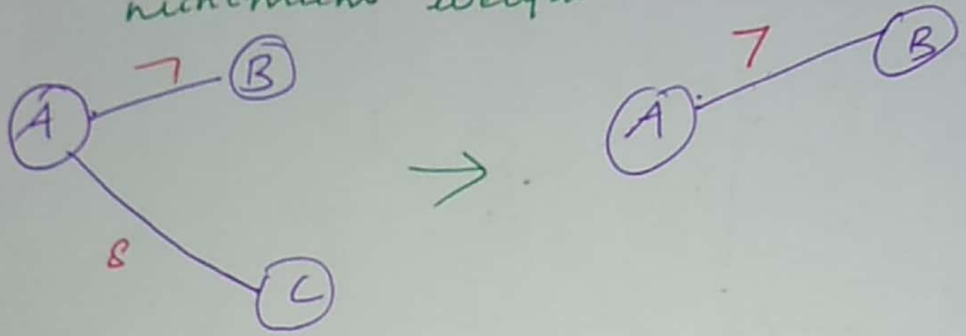
Remove all loops and parallel edges

8 & 6
take minimum edge



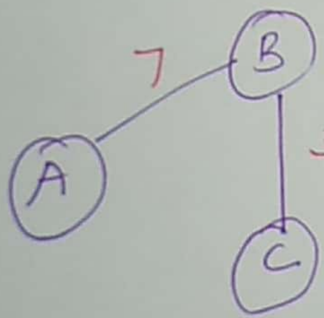
Choose any arbitrary node as the roote node (vertex)

Check out all the outgoing edges from this root node. from that choose the minimum weight

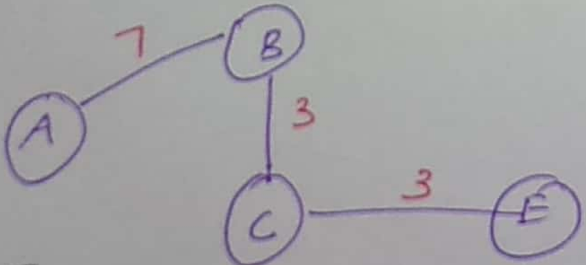


Now check all the outgoing edges from A as well as B

8, 3 & 6 (minimum) we have to choose



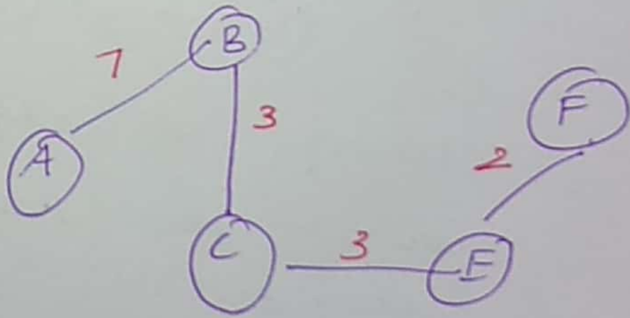
8, 6, 3 & 4



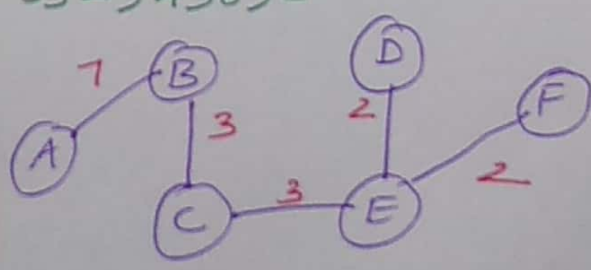
Time complexity

$\boxed{O((V+E)\log V)}$

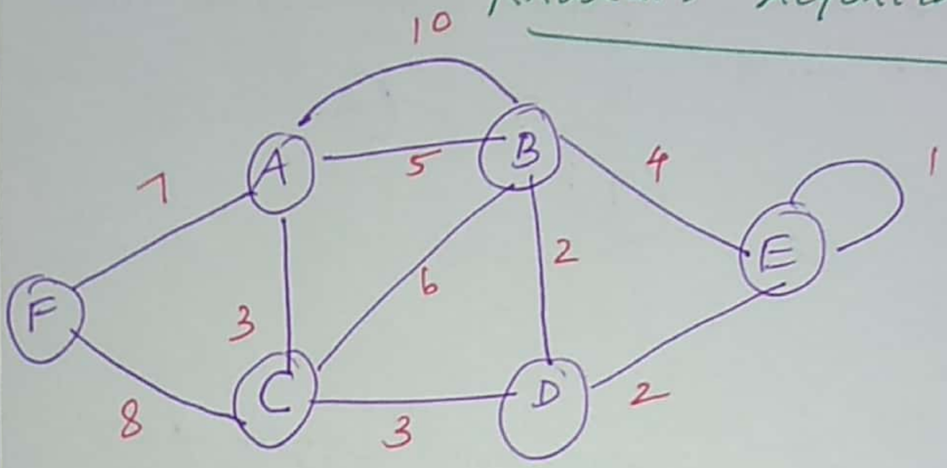
8, 6, 2, 2, 4



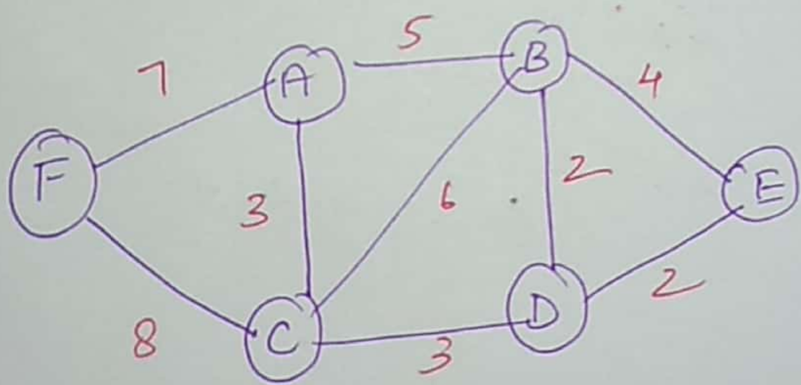
5, 2, 4, 6, 8



Kruskal's Algorithm – Greedy



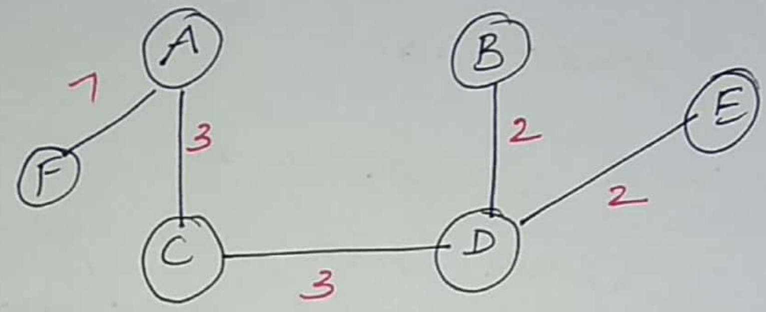
1) Remove all loops & parallel edges.



2) Arrange all edges according to edge weight in increasing order of weight.

Scanned with CamScanner

BD = 2 ✓
DE = 2 ✓
AC = 3 ✓
CD = 3 ✓
BE = 4 ✗
AB = 5 ✗
BC = 6 ✗
AF = 7 ✓
FC = 8



Property

→ MST does not contain any cycles
→ If n number of vertices in a graph then it should have same number of vertices and n-1 edges.

Time Complexity

$$\boxed{O(E \log E) \text{ or } O(E \log V)}$$

Scanned with CamScanner

Greedy Approach

Feasible: It has to satisfy the problem's constraints.

Locally optimal: It has to be the best local choice among all feasible choices available on that step.

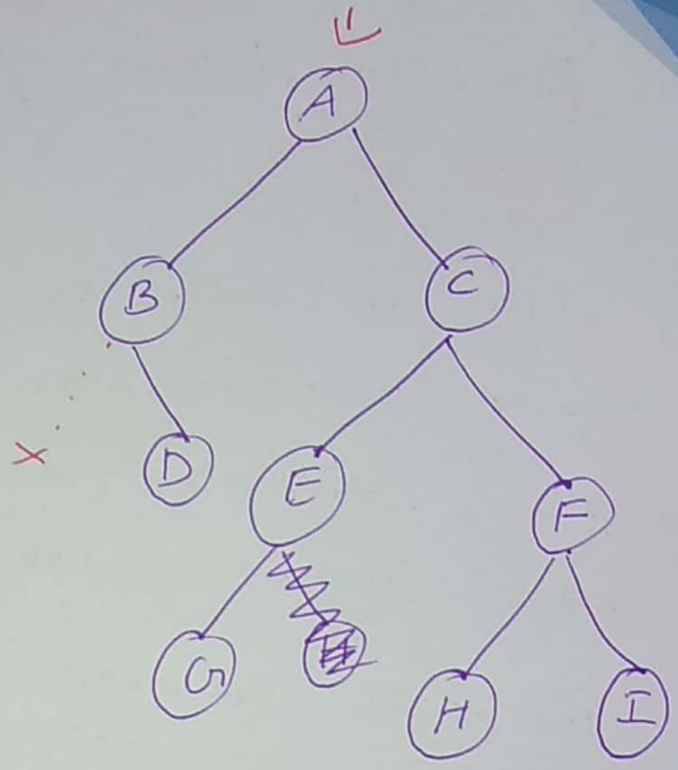
Irrevocable: Once made, it cannot be changed on subsequent steps of the algorithm.

Tree Traversal

Inorder: Left Root Right

Preorder: Root Left Right

Postorder: Left Right Root



Inorder:

BDAGECHFI

Preorder:

ABDCEGFHI

Postorder:

DBGEHIFCA

Inorder Traversal

Algorithm

Until all nodes are traversed-

Step 1 - Recursively traverse left subtree

Step 2 - Visit root node

Step 3 - Recursively traverse right subtree

Pre-order Traversal

Algorithm

Until all nodes are traversed

Step 1 - Visit root node

Step 2 - Recursively traverse left subtree

Step 3 - Recursively traverse right subtree

Post-order Traversal

Algorithm

Untill all nodes are traversed-

Step 1 - Recursively traverse left subtree

Step 2 - Recursively traverse right subtree

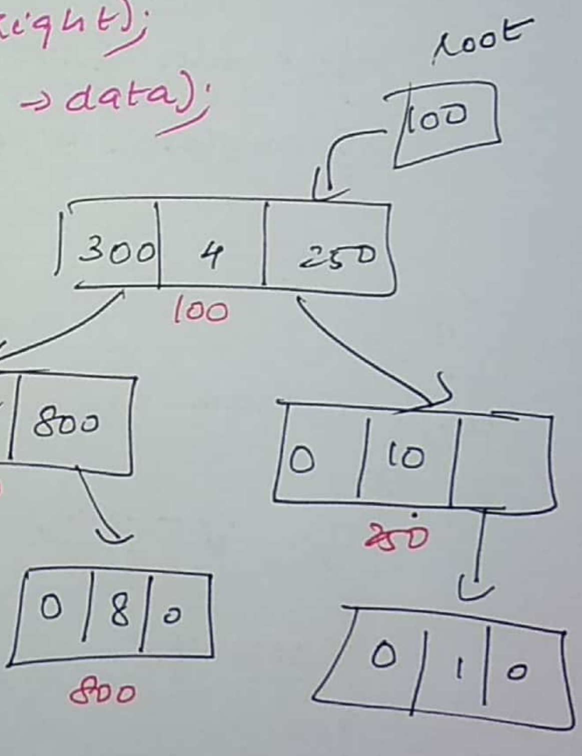
Step 3 - Visit Root Node

Program

```
void main()
{
 struct node * root;
 printf(" Pre order is : ");
 preorder (root);
```

```
Inorder (root);
postorder (root);
void Preorder (struct node * root)
{
    if (root ==0)
    {
      return;
    }
    else
    {
      printf ("%d", root-> data);
      preorder (root -> left);
      preorder (root -> right);
    }
}
void Inorder (struct node * root)
{
    if (root ==0)
    {
      return;
    }
    else
    {
      Inorder (root -> left);
      Printf ("%d", root -> data);
      Inorder (root -> right).
```

```
    }
  }
void Postorder (struct node *root)
{
   if (root == 0)
   {
     return;
   }
   else
   {
     postorder (root -> left);
     postorder (root -> right);
     printf ("%d", root -> data);
   }
  }
}
```



Inorder = Left Root Right

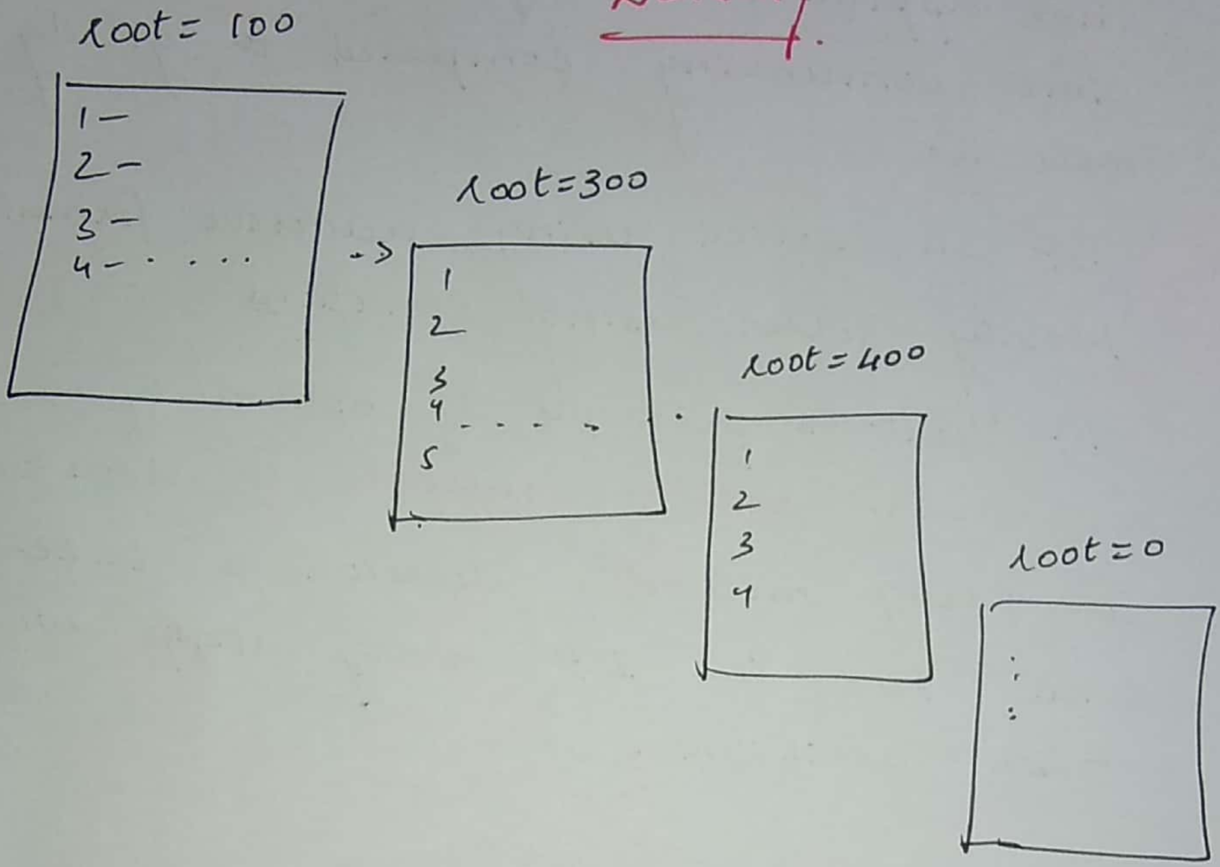
= $\boxed{7\ 5\ 8\ 4\ 10\ 1}$

Preorder = Root L Right

= $\boxed{4\ 5\ 7\ 8\ 10\ 1}$

Postorder = L Right Root = $\boxed{7\ 8\ 5\ 1\ 10\ 4}$

Working.



Dynamic Programming - Introduction.

Greedy & Dynamic both are used for solving optimization either maximum & minimum problem.

In Greedy we find out a predefined method for ~~solvin~~, optimum solution but in dynamic we find out all possibility for optimum solution.

→ This approach is different and little time consuming compared to greedy method.

→ DP are solved using recursive formulas.

→ Mostly solved using iterations

→ DP follows principle of optimality ⇒ It means taking sequence of decisions

→ In Greedy method decision is taken once but in DP every stage we take decisions.