3. Greedy Method

* Greedy method is the most straight forward design technique.
* As the name suggests they are short sighted in their approach, taking decisions on the basis of the information immediately at the hand without worrying about the effect. These decision may have in the future.

Definition:

A problem with 'n' inputs can have some constraints, any subset that satisfy these constraints are called as a feasible solution.

* A feasible solution that either maximize (or) minimize a given objective function is called optimal solution

Control Abstract for the greedy method:

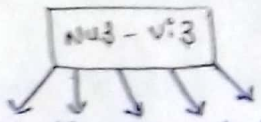
```
Algorithm greedy (a, n)
{
    // a [1:n]
    sol = 0;
    for i:= 1 to n do
    { x = select (a);
      if (feasible (sol, x)) then
        sol = union (sol, x);
    }
    return sol;
}
```

* Feasible is a function that determines whether 'x' can be included into the solution vector.
* A function 'union' combines 'x' with the solution and updates the objective function.

Ex: To travel from Nuzvid to Vijayawada we have different solutions which we generally call as feasible solutions.

Feasible solutions:

1) we can travel by bus
2) we can travel by bike
3) we can travel by car
4) we can travel by walk
5) we can travel by train

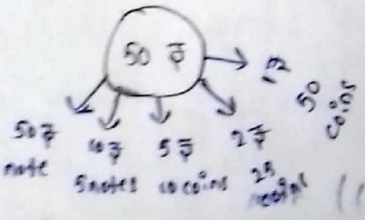


But our aim is to travel to the destination with minimum cost and time.

∴ Train is the optimal solution for the given problem.

Ex 2: Suppose we need 50₹ money from our friends, our aim is to getting money with minimum no.of notes, or coins.

Ans: The set of feasible solutions are.



But the optimal solution is, taking one 50₹ note Because it is the minimum number among all the feasible solutions.

Difference b/w Greed & Dynamic programming.

0/1 Knapsack problem:

Here A knapsack / a bag is given, we are given 'n' objects. from 1 to 'n', each object 'i' has a positive weight and positive profit as 'pi'

* The knapsack carry atmost weight 'm'. By solving the knapsack problem we have a capacity constraint (m) a fraction (x_i) $0 \le x \le 1$ of object i is placed into the knapsack then a profit $p_i x_i$ is earned when we try to solve this problem using the greedy approach, our goal is:

1) choose only those objects that gives maximum profit
2) The total weight of the selected object should be less than (or) equal to m

* In other words maximize $= \sum p_i x_i$ subject to $0 \le i \le n$ $w_i x_i$ is less than or equal to m.

where the knapsack can carry the fraction x_i of an object i such that $0 \le x_i \le 1$, $1 \le i \le n$.

Consider the following instances of knapsack problem.

n = 3, m = 20, (P_1, P_2, P_3) = 25, 24, 15; weights (w_1, w_2, w_3) = 18, 15, 10 respectively.

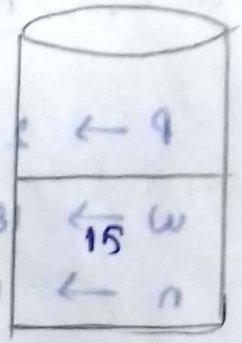
H = 20

n	P	w
1	25	18
2	24	15
3	15	10

Solution : $\frac{P}{\omega}$ dec order :

$$\frac{P_1}{\omega_1} = \frac{25}{18} = 1.38 \qquad \frac{P_2}{\omega_2} = \frac{24}{15} = 1.6 \qquad \frac{P_3}{\omega_3} = \frac{15}{10} = 1.5$$

	1.6	1.5	1.38
$\omega \rightarrow$	15	10	18
$P \rightarrow$	24	15	25
$n \rightarrow$	2	3	1



$x_2 = 1$, $x_3 = \frac{5}{10} = 0.5$, $x_1 = 0$ and Total weight = 20

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

$= 0 + 24 + 15 \times \frac{1}{2} = \underline{31.5}$

2) Consider the following instances of knapsack.

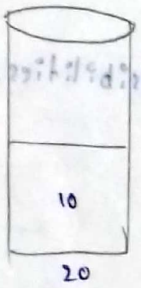
M = 20

n	w	p
1	18	30
2	15	21
3	10	18

solution 3: $\frac{P}{\omega}$ dec order:

$\frac{P_1}{\omega_1} \frac{30}{18} = 1.66$ $\frac{P_2}{\omega_2} \frac{21}{15} = 1.4$ $\frac{P_3}{\omega_3} = \frac{18}{10} = 1.8$

	1.66	1.66	1.4
$\omega \rightarrow$	18	18	15
$P \rightarrow$	30	30	28
$n \rightarrow$	3	1	2



$x_3 = 1$

$x_1 = \frac{10}{18}$

$x_2 = 0$

Total weight: $10 + 18 \times \frac{10}{18} = 20.$

Total profit: $18 \times 1 + 30 \times \frac{10}{18}$

$= 34.66$

Among all $\frac{P}{\omega}$ dec order got profit 34.66 which is the maximum profit. so it is the optimal solution.

3) find an optimal solution for. $n=7$, $m=15$ and

$P[1-7] = [10, 5, 15, 7, 6, 18, 3]$

$\omega[1-7] = [2, 3, 5, 7, 1, 4, 1]$

Algorithm:

```
Algorithm greedy knapsack (m, n)
// we have P[1-n] and w[1-n] contains the pro-
  -fit and weight respectively, of the n objects, ordered
  such that Pi/wi is greater than (or) equal to Pi+1/wi+1
  m is the knapsack size and x[1-n] is solution vector.
{
  for i:=1 to n do
    x[i] := 0.0;
  U := m;
```

```
for i=1 to n do
{
    if ( w[i] > U) then
        break;
    x[i] = 1.0;
    U = U - w[i];
}
if (i <= n) then
    x[i] = U / w[i];
}
```

* The time complexity of this algorithm is O(n)

Given that there are 7 objects, profits are

p[1-7] = [10, 5, 15, 7, 6, 18, 3]

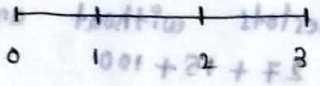
w[1-7] = [2, 3, 5, 7, 1, 4, 1]

order;

Job scheduling with dead lines:

Ex:

n	D	P
1	3	100
2	1	10
3	2	15
4	1	27



* Here we have 3 time slots, so we can execute 3 jobs in this time.

a) i) 1, 2, 3 $3 \le 1 \le 2$ x

ii) 1, 3, 4 $3 \le 2 \le 1$ x

iii) 1, 2, 4 $3 \le 1 \le 1$ x

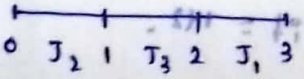
iv) 3, 2, 1 $3 \le 2 \le 1$ x

v) 1, 4, 3 $3 \le 1 \le 2$ x

vi) 1, 2, 4, 3 $3 \le 1 \le 1$ x

b) i) 2, 1, 3 $1 \le 3 \le 2$ x

ii) 2, 3, 1 $1 \le 2 \le 3$ ✓

$\Rightarrow$ 

profit = 10 + 15 + 100 = 125.

iii) 2, 4, 1 $1 \le 1 \le 3$ ✓

$\Rightarrow$ But on machine we can't execute J_2 and J_4 at a time so it's not a feasible

iv) 2, 1, 4 $1 \le 3 \le 1$ x

v) 2, 3, 4 $1 \le 2 \le 1$ x

vi) 2, 4, 3 $1 \le 1 \le 2$ ✓

$\Rightarrow$ But on machine we can't execute J_2 and J_4 at a time because they both have to complete at time slot 1 only which is not a possible condition.

c) i) 3, 1, 2 $2 \le 3 \le 1$ ×

ii) 3, 2, 1 $2 \le 1 \le 3$ ×

iii) 3, 2, 4 $2 \le 1 \le 1$ ×

iv) 3, 4, 2 $2 \le 1 \le 1$ ×

v) 3, 4, 1 $2 \le 1 \le 3$ ×

vi) 3, 1, 4 $2 \le 3 \le 1$ ×

d) i) 4, 1, 2 $1 \le 3 \le 1$ ×

ii) 4, 2, 1 $1 \le 1 \le 3$ ✓

→ But we can't execute J_4 and J_2 on machine at a time.

iii) 4, 1, 3 $1 \le 3 \le 2$ ×

iv) 4, 3, 1 $1 \le 2 \le 3$ ✓ * Here we can allocate the timeslots without any violation.

profit = 27 + 15 + 100
= 142

v) 4, 2, 3 $1 \le 1 \le 2$ ✓

→ But we can't execute J_4 and J_2 on machine at a time.

vi) 4, 3, 2 $2 \le 2 \le 1$ ×

∴ The feasible solutions are:

order 1 : 2, 4, 1 profit = 125

order 2 : 4, 3, 1 profit = 142.

* The optimal solution is sequence 4, 3, 1 as it got maximum profit 142.

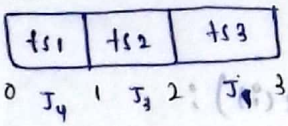
Another solution:

n	D	P
1	3	100
2	1	10
3	2	15
4	1	27

1) Arrange in profit descending order:

P →	100	27	15	10
D →	3	1	2	1
n →	1	4	3	2

* Allocate the job to the time slot which is equal to it's deadline, if it is busy then allocate it to the previous one



J_4 J_3 J_1

∴ Profit = 100 + 27 + 15
= 142.

Ex2:

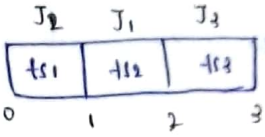
n	D	P
1	2	60
2	1	100
3	3	20
4	2	40
5	1	20

Ans:

1) profit decreasing order:

P →	100	60	40	20	20
D →	1	2	2	3	1
n →	2	1	4	3	5

2)



profit = 100 + 60 + 20
= 180

Ex3:

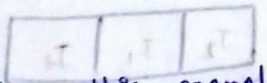
n	D	P
1	2	100
2	1	19
3	2	27
4	1	25
5	3	15

Ans: 1) profit decreasing order:

P →	100	27	25	19	15
D →	2	2	1	1	3
n →	1	3	4	2	5

2)

profit = 25 + 100 + 27
= 142.



* In this example after executing J_1 and J_3, we have to execute J_4 / J_2 But the deadline of these jobs is 1 but already we executed J_3 in time slot 1. so J_5 is executed at the time slot 3.

Algorithm for Job sequencing:

```
Algorithm Job_sequencing ( )
{
  // Arrange the jobs in the profit decreasing order
  for i:=1 to n do
    k = min (dmax, deadline(i));
    while ( k>=1) do
      if (time slot [k] is empty) then
        time slot [k] = job (i);
        break;
      end if
    k = k-1;
    end while
  End for
}
```

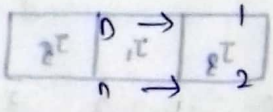
Ex:

n	D	P
1	2	60
2	1	100
3	3	200
4	2	40
5	1	20

Ans:

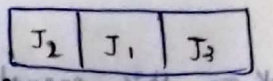
profit decreasing order:

P →	100	60	40	20	20
	2	2	3	1	
	1	4	3	5	



Tracing:

- for i:=1 to 5 do → true



k = min (dmax, deadline(i));

⇒ k = min (3, 1) ⇒ k=1;

→ while (1 ≥ 1) do → true

if (timeslot [1] is empty) → true

→* timeslot [1] = job (1);

break;

End while;

- for i:=2 to 5 do → true

k = min (3, 2) → k=2;

→ while (2 >= 1) → true

if (timeslot [2] is empty) → true

→ timeslot [2] = job (2);

break;

End while

- for i = 3 to 5 do → true

k = min (3, 2) → k = 2;

→ while (2 >= 1) → true

if (timeslot [2] is empty) → false

k = k - 1; → k = 1;

→ while (1 >= 1) → true

if (timeslot [1] is empty) → false

end if

k = 0;

→ while (0 >= 1) → false

end while

- for i = 4 to 5 do → true

k = min (3, 3) → k = 3;

→ while (3 >= 1) → true

if (timeslot [3] is empty) → true

→ timeslot [3] = job [4];

break;

end if;

End while;

- for i = 5 to 5 do → true

k = min (3, 1) → k = 1;

→ while (1 >= 1) → true

if (timeslot [1] is empty) → false

end if;

k = 0;

→ while (0 >= 1) → false (End while)

- for i = 6 to 5 do → false (End for)

Job sequencing with deadlines:

Here set of n jobs are given each job 'i' has deadline 'd_i', profit 'p_i' ($d_i \geq 0; p_i \geq 0$)

The dead line of a job is the time on which / before means, the job has to be completed. Each job takes one unit of time.

* If the job is completed on / before the deadline then the profit is gained otherwise there is no profit.
* A feasible solution is the set of jobs that are completed on/before the deadline.
* An optimal solution is a feasible solution which gives the maximum profit.
* Only one job at a time is exhibited.
* In order to execute the jobs the job sequence has to follow the condition $d_1 \leq d_2 \leq d_3 \leq \cdots \leq d_n$

Ex: Refer previous example. (with general method)

Ex2:

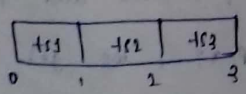
n	D	P
1	2	60
2	1	100
3	3	20
4	2	40
5	1	20

P →	100	60	40	20	20
D →	1	2	2	3	1
n →	2	1	4	3	5

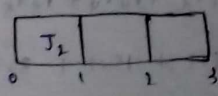
Explanation: Arrange / sort the jobs in the decreasing order of their profits. Find the max deadline in the problem.
⇒ in this ex dmax = 3.

~~1) select the first job in the~~

consider 3 timeslots, which are empty at first.



2) consider the first job in the order check it's deadline, in this example the deadline of job 2 is '1'. then allocate timeslot 1 to J_2.



3) we needn't complete the job on the first day of it's

deadline, so we try to complete the job on ~~before~~ the last day of deadline, if the last day of deadline is not empty check the previous day and if it's also not empty check it's previous day. If no previous day is empty then we skip the job.

4) select the next job in the order allocate the time-slot like that select the remaining jobs in the order, allocate the time slots.

5) finally we get the job sequence with high / maximum profit.

Algorithm : Refer the previous pages.

Time complexity :

The time complexity of the algorithm is $O(n^2)$

Minimum cost spanning tree

Spanning tree :

Spanning tree is a connected undirected graph without any cycle and it should contain all the vertices of the given graph.

* A complete graph with 'n' vertices is having n^{n-2} spanning trees.
* Minimum cost spanning tree is a tree with minimum weight.
* Here a graph is given, each edge in the graph is associated with cost / weight.
* If the graph contains more number of vertices, then it is difficult to find out all the possible spanning trees and then a minimum cost spanning tree
* In order to find out, minimam cost spanning tree, we have two methods. they are:
  1) Prim's Algorithm
  2) Kruskals Algorithm

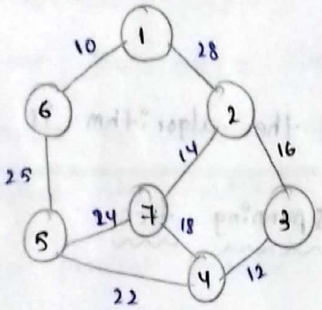
Prim's Algorithm:

* It is used to find out the MST (minimum cost spanning tree) using greedy method.

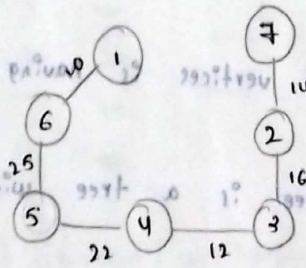
Procedure: Consider all the vertices of the given graph.

1) Select an edge with minimum cost / weight.
2) Select the adjacent minimum cost edge, repeat the process for all the vertices in the graph) until the spanning tree contains (n-1) edges.
3) While selecting an edge we have to take care for not forming a cycle.

Example: Find the minimum cost spanning tree.



Ans: 1) Construct spanning tree by selecting the minimum weight edge: i.e,



weight = 99

Algorithm prim's:

```
Algorithm prim (E, cost, n, t)
// E is the set of edges in G. cost[1:n, 1:n] is the cost
// Adjacency matrix of an n vertex graph such that
// cost[i,j] is either a positive real number or ∞
// if no edge exists. A minimum spanning tree is
// computed and sorted as a set of edges in the array
// t[1:n-1, 1:2]. t[i,1], t[i,2] is an edge in
// the minimum cost spanning tree. The final cost is
// returned.
```

```
{
  Let (k, l) be an edge of minimum cost in E;
  mincost := cost[k, l];
  t[1,1] := k;  t[1,2] := l;
  for i := 1 to n do      // Initialize near
    if ( cost[i,l] < cost[i,k] ) then
        near[i] := l;
    else
        near[i] := k;
  near[k] := near[l] := 0;
  for i := 2 to n-1 do
  {
    // Find n-2 additional edges for t
    Let j be an index such that near[j] ≠ 0
    and cost[j, near[j]] is minimum;
    t[i,1] := j;
    t[i,2] := near[j];
    mincost := mincost + cost[j, near[j]];
    near[j] := 0;
    for k := 1 to n do  // update near[]
      if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k,j]))
        then near[k] := j;
  }
  return mincost;
}
```

Time complexity :

The time complexity of prim's algorithm is $O(n^2)$

* Remove self loops and parallel edges if any in the graph (, st point) (if there is a parallel edge remove the max cost edge consider the minimum cost edge)

Tracing :

Vertex	near
2	0
3	0
4	0
5	0
6	0
7	0

(k,l) = (1,6)

mincost = 10 ;

t[1,1] = 1 , t[1,2] = 6 ;

- for i:=1 to 7 do → true

  if (cost(i,1) < cost(i,k))

  → if (cost(1,6) < cost(1,1))

  ⇒ if (10 < ∞) → True

  near[1] = 6 ;

- for i:=2 to 7 do → true

  if (cost(i,1) < cost(i,k))

  → if (cost(2,6) < cost(2,1))

  ⇒ if (∞ < 28) → false

  else

  near[2] = 1 ;

- for i:=3 to 7 do → true

  if (cost(3,6) < cost(3,1))

  → if (∞ < ∞) → false

  else

  near[3] = 1 ;

- for i:=4 to 7 do → true

  if (cost(4,6) < cost(4,1))

  → if (∞ < ∞) → false

  else

  near[4] = 1 ;

- for i:=5 to 7 do → true

  if (cost(5,6) < cost(5,1))

  → if (25 < ∞) → true

  near[5] = 6 ;

- for i:=6 to 7 do → true

  if (cost(6,6) < cost(6,1))

→ if (∞ < 10) → false

else

near [6] = 1;

- for i=7 to 7 do → true

if (cost (7,6) < cost (7,1))

⇒ if (∞ < ∞) → false

else

near [7] = 1;

→ for i=8 to 7 do → false

near [1] = 0;

near [6] = 0;

* for i=2 to 6 do ⟷ true

2, 3, 4, 5, 7

cost (2,1) = 28; cost (3,1) = ∞, cost (4,1) = ∞

cost (5,6) = 25, cost (7,1) = ∞

so minimum is cost (5,6) and j = 5;

t [2,1] = 5;

t [2,2] = 6;

mincost = 10 + 25 = 35; near [5] := 0;

> for k=1 to 7 do → true

near [1] ≠0 → false

> for k=2 to 7 do → true

if (near [2] ≠0 and (cost [2,1] > cost [2,5]))

⇒ if (near [2] ≠0 and (28 > ∞)) → false

> for k=3 to 7 do → true

if (near [3] ≠0 and (cost [3,1] > cost [3,5]))

⇒ if (near [3] ≠0 and (∞ > ∞)) → false

> for k=4 to 7 do → true

if (near [4] ≠0 and (cost [4,1] > cost [4,5]))

→ if (near [4] ≠0 and (∞ > 22)) → true

near [4] = 5;

→ for k=5 to 9 do → true

if (near[5] ≠ 0) → false

→ for k=6 to 9 do → true

if (near[6] ≠ 0) → false

→ for k=7 to 9 do → true

if (near[7] ≠ 0 and cost(7,i) ≤ cost(7,8))

→ if (near[7] ≠ 0 and ∞ ≥ 24) → true

near[7] = 7;

→ for k=8 to 9 do → false

→* for i=2 to 6 do → true

2, 3, 4, 7

cost(2,3) = 25, cost(3,4) = 20, cost(4,7) = 32,

cost(3,5) = 24;

so minimum cost is cost(4,5) and i = 4;

t[3,1] = 4;

t[3,2] = 5;

mincost = 32 + 25 + 99;

near[4] = 0;

→ for k=1 to 9 do → true

if (near[1] ≠ 0) → false

→ for k=2 to 9 do → true

if (near[2] ≠ 0 and cost(2,1) ≤ cost(2,4))

→ if (near[2] ≠ 0 and 25 > ∞) → false

→ for k=3 to 9 do → true

if (near[3] ≠ 0 and cost(3,7) ≤ cost(3,4))

→ if (near[3] ≠ 0 and ∞ > 12) → true

near[3] = 4;

→ for k=4 to 9 do → true

if (near[4] ≠ 0) → false

> for k=5 to 7 do → true

if (near[5] ≠0) → false

> for k=6 to 7 do → true

if (near[6] ≠0) → false

> for k=7 to 7 do → true

if (near[7] ≠0 and cost(7,5) > cost(7,4))

⇒ if (near[7] ≠0 and 24 > 18) → true

near[7] = 4;

> for k=8 to 7 do → false

* for i=4 to 6 do → true

2, 3, 7

cost(2,1) = 28, cost(3,4) = 12, cost(7,4) = 18

so minimum cost is cost(3,4) and j=3;

t[4,1] = 3;

t[4,2] = 4;

near[3] = 0;

mincost = 57 + 12 = 69;

> for k=1 to 7 do → true

if (near[1] ≠0) → false

> for k=2 to 7 do → true

if (near[2] ≠0 and cost(2,1) > cost(2,7))

⇒ if (near[2] ≠0 and 28 > 16) → true

near[2] = 3;

> for k=3 to 7 do → true

if (near[3] ≠0) → false

> for k=4 to 7 do → true

if (near[4] ≠0) → false

> for k=5 to 7 do → true

if (near[5] ≠0) → false

> for k=6 to 7 do → true

if (near[6] ≠0) → false

> for k = 7 to 7 do → true

if (near [7] ≠ 0 and cost (7, 4) > cost (7, 3))

⇒ if (near [7] ≠ 0 and 18 > ∞) → false

> for k = 8 to 7 do → false

* for i = 5 to 6 do → true

2, 7

cost (2, 3) = 16 , cost (7, 4) = 18 ;

so minimum cost is cost (2, 3) → j = 2 ;

t [5, 1] = 2 ;

t [5, 2] = 3 ;

min cost = 69 + 16 = 85 ;

near [2] = 0 ;

> for k = 1 to 7 do → true

if (near [1] ≠ 0) → false

> for k = 2 to 7 do → true

if (near [2] ≠ 0) → false

> for k = 3 to 7 do → true

if (near [3] ≠ 0) → false

> for k = 4 to 7 do → true

if (near [4] ≠ 0) → false

> for k = 5 to 7 do → true

if (near [5] ≠ 0) → false

> for k = 6 to 7 do → true

if (near [6] ≠ 0) → false

> for k = 7 to 7 do → true

if (near [7] ≠ 0 and cost (7, 4) > cost (7, 2)) ;

⇒ if (near [7] ≠ 0 and 18 > 14) true

near [7] = 2 ;

> for k = 8 to 7 do → false

* for i = 6 to 6 do → true

7 , cost (7, 2) = 14 ; j = 7 ;

t[6,1] = 7 ;

t[6,2] = 2;

mincost = 86+14 = 99;

near[7] = 0;

> for k=1 to 7 do → true

if (near[1]!=0) ; → false

> for k=2 to 7 do → true

if (near[2] ≠0) → false

> for k=3 to 7 do → true

if (near[3] ≠0) → false

> for k=4 to 7 do → true

if (near[4] ≠0) → false

> for k=5 to 7 do → true

if (near[5] ≠0) → false

> for k=6 to 7 do → true

if (near[6] ≠0) → false

> for k=7 to 7 do → true

if (near[7] ≠0) → false

> for k=8 to 7 do → false.

* for i=7 to 6 do → false

return 99;

∴ minimum cost = 99

Kruckal's Algorithm:

It is the another way to find out the MST using greedy method

1) Here a weighted undirected graph is given
2) In order to construct the MST, Arrange / sort the edges of the graph in the increasing order of their cost / weight.
3) select the first minimum cost edge in the order and add it to the spanning tree.
4) Next select the next minimum cost edge in the order that need not be an adjacent edge and then add it to the spanning tree. Repeat the above process until the spanning tree contains 'n-1' edges where 'n' is the no.of vertices of the given graph.

→ Refer the previous examples.

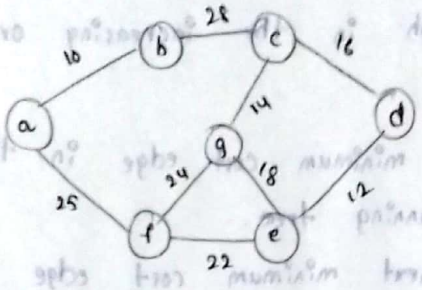
Algorithm:

```
Algorithm kruskals ( V, E)
{
// where V is the set of vertices and E is the
// set of edges.
    A = ∅ ; // A temporary set for storing edges of MST
    for each v ∈ V do
        make-disjoint-set (v);
    sort E by weight increasing order
    for each (v1, v2) ∈ E
    {
        if find (v1) != find (v2) then
            A = A ∪ {(v1, v2)}
            union (v1, v2) ;
    }
    return A ;
}
```

Time complexity :

Time complexity of the Kruskal's Algorithm is $O(\epsilon \log \epsilon)$ where ϵ is the no. of edges in the graph.

Tracing :



Here V = 7 and E = the set of all edges.

Algorithm Kruskals (V, E)

{

A = ∅ ;

for each v ∈ V do

make_disjoint (v) ;

⇒ {a} {b} {c} {d} {e} {f} {g}

sort E by weight increasing order ;

10	12	14	16	18	22	24	25	28
(a,b)	(d,e)	(c,g)	(c,d)	(g,e)	(f,c)	(f,g)	(a,f)	(b,c)

for each $(v_1, v_2) \in E$

{

- if find (a) ≠ find (b) → true

A = A ∪ {(v_1, v_2)} ;

A = {(a b)} ;

union (v_1, v_2) ;

{a,b} {c} {d} {e} {f} {g}

- if find (d) ≠ find (e) → true

A = A ∪ {(v_1, v_2)} ;

⇒ A = {(a b) (d e)}

union (v_1, v_2) ;

⇒ {a,b} {c} {d,e} {f} {g}

- if find (c) ≠ find (g) → true

  A = A ∪ {(v1, v2)} ;

  → A = {(a b) (d e) (c g)}

  Union (v1, v2);

  → {a, b} {c, g} {d, e} {f}

- if find (c) ≠ find (d) → true

  A = A ∪ {(v1, v2)} ;

  → A = {(a b) (d e) (c g) (c d)}

  Union (v1, v2) ;

  → {a, b} {c, d, e, g} {f}

- if find (g) ≠ find (e) → false

- if find (f) ≠ find (e) → true

  A = A ∪ {(v1, v2)} ;

  → A = {(a b) (d e) (c g) (c d) (f e)}

  Union (v1, v2) ;

  → {a, b} {c, d, e, f, g}

- if find (f) ≠ find (g) → false

- if find (a) ≠ find (f) → true

  A = A ∪ {(v1, v2)}

  → A = {(a b) (d e) (c g) (c d) (f e) (a f)}

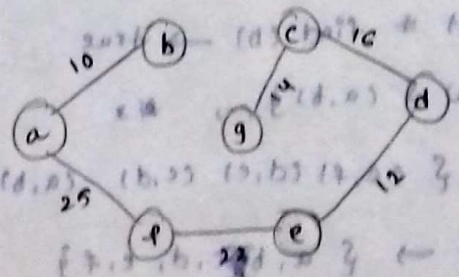
  Union (v1, v2) ;

  → {a, b, c, d, e, f, g}

- if find (b) ≠ find (c) → false

∴ A = {(a, b) (d, e) (c, g) (c, d) (f, e) (a, f)}

Graph:



Dijkstra's algorithm (or) single source shortest path:

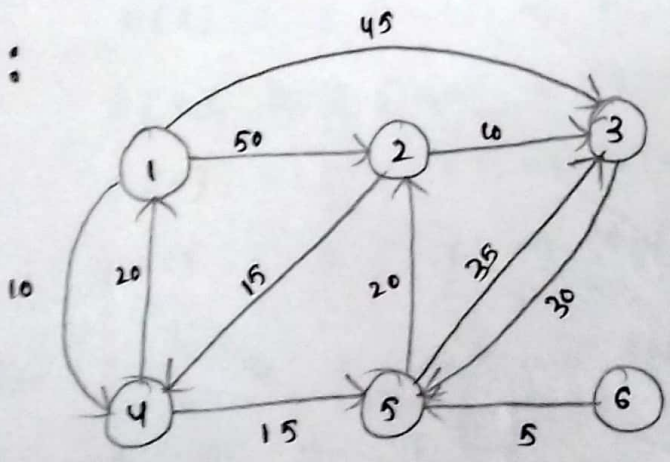
* Here a weighted directed graph is given and a source is given, we have to find out the shortest path from the source to all the other nodes in the ~~othe~~ graph.

* To find out the shortest path from source to the other node, we have to check.

1) Is there a path from source to that node
2) Is there more than one path from source to that node
3) If there is no path from the source to the other node to reach, then the cost is '∞'.
4) If there is a path from source to the other node then, we have to find out the shortest distance to reach to that node.

* Dijkstra's Algorithm (or) shortest path problem is used when we want to traverse from one city to all the other cities in the country.

Example:



Source = 1

1 - 4 = 10

1 - 4 - 5 = 25

1 - 4 - 5 - 2 = 45

1 - 3 = 45

1 - 6 = ∞

Explanation :

* Here the source is 1 and destinations are all the other nodes in the graph.
* First of all find out a node from source that we can quickly reach.
* In this example the node that we can quickly reach is 4.
  i.e., 1 to 4 = 10
* Next find out the next node that we can quickly reach via node 4. i.e., 5
  So 1 - 4 - 5 = 25
* Like this we try to find out the path from source to the destinations via the nodes that we can quickly reach.

1 - 4 - 5 - 2 = 45

1 - 3 = 45

1 - 6 = ∞.

Algorithm Dijkstra's ( )

{

let V = {1, 2, ---, n} & source = 1;

S = {1};

for i := 2 to n do

D[i] = c[1, i]

for i := 1 to n do

{

choose a vertex w ∈ V - S such that

D[w] is minimum;

S = S ∪ {w};

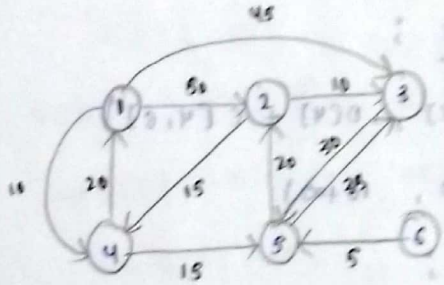
for each vertex v ∈ V - S

D[v] = min (D[v], D[w] + c[w, v])

}

}

Ex:



Tracing:

Algorithm Dijkstra's ( )

{ let V = {1, 2, 3, 4, 5, 6} & source = 1;

S = {1};

* for i := 2 to 6 do

D[2] = c[1, 2] ⇒ D[2] = 50;

D[3] = c[1, 3] ⇒ D[3] = 45;

D[4] = c[1, 4] ⇒ D[4] = 10;

D[5] = c[1, 5] ⇒ D[5] = ∞;

D[6] = c[1, 6] ⇒ D[6] = ∞;

• for i := 1 to 6 do ⟶ true

	2	3	4	5	6
D[i]	50	45	10	∞	∞

so $w = 4$;

$S = S \cup \{4\} \Rightarrow S = \{1, 4\}$;

> for each vertex $v \in V - S$

2, 3, 5, 6

$D[2] = \min(D[2], D[4] + C[4,2])$

$\Rightarrow D[2] = \min(50, 10 + \infty)$

$\Rightarrow D[2] = 50$;

> for each vertex $v \in V - S$

$D[3] = \min(D[3], D[4] + C[4,3])$

$\Rightarrow D[3] = \min(45, 10 + \infty)$;

$\Rightarrow D[3] = 45$;

> $D[5] = \min(D[5], D[4] + C[4,5])$

$\Rightarrow D[5] = \min(\infty, 10 + 15)$

$\Rightarrow D[5] = 25$;

> $D[6] = \min(D[6], D[4] + C[4,6])$

$\Rightarrow D[6] = \min(\infty, 10 + \infty)$

$\Rightarrow D[6] = \infty$;

- for $i = 2$ to 6 do → true

	2	3	5	6
w	50	45	25	∞

so, $w = 5$;

$S = S \cup \{5\} \Rightarrow S = \{1, 4, 5\}$

> for each vertex $v \in V - S$

2, 3, 6

$D[2] = \min(D[2], D[5] + C[5,2])$

$\Rightarrow D[2] = \min(50, 25 + 20)$

$\Rightarrow D[2] = 45$;

$D[3] = \min(D[3], D[5] + C[5,3])$

$\Rightarrow D[3] = \min(45, 25 + 35)$;

D[3] = 45;

> D[6] = min (D[6] , D[5] + C[5,6])

⇒ D[6] = min (∞ , 25 + ∞)

D[6] = ∞ ;

- for i=3 to 6 do → true

2	3	6
46	45	∞

so w = 3

S = S ∪ {3} ⇒ S = { 1, 2, 4, 5 }

> for each vertex v ∈ V-S

3 , 6

D[3] = min (D[3] , D[2] + C[2,3])

⇒ D[3] = min (45 , 45 + 10)

D[3] = 45 ;

> D[6] = min (D[6] , D[2] + C[2,6])

⇒ D[6] = min (∞ , 45 + ∞)

⇒ D[6] = ∞ ;

- for i=4 to 6 do. → true

3	6
45	∞

so w = 3

S = S ∪ {3} ⇒ S = { 1, 2, 3, 4, 5 }

> for each vertex v ∈ V-S

6

D[6] = min (D[6] , D[3] + C[3,6])

⇒ D[6] = min (∞ , 45 + ∞)

⇒ D[6] = ∞ ;

- for i=5 to 6 do → true

6 → ∞

so w = 6 , S = S ∪ {6} ⇒ S = { 1, 2, 3, 4, 5, 6 }

- for each vertex $v \in V-S$ do → false.

- for $i=6$ to 6 do → true.

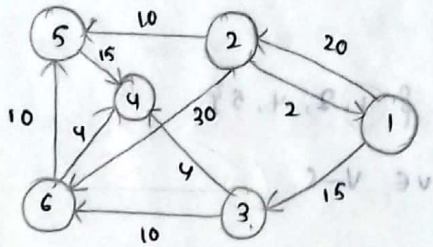
  but no vertices.

- ~~for $i=6$ to 6 do~~ → ~~false~~ true

  ~~but no vertices~~

- for $i=4$ to 6 do → false

Ex:



Tracing :

Algorithm Dijkstra's ( )

{

let $V = \{1,2,3,4,5,6\}$ & source = 1;

$S = \{1\}$

* for $i=2$ to 6 do → true.

$D[2] = C[1,2] = 20;$

$D[3] = C[1,3] = 15;$

$D[4] = C[1,4] = \infty;$

$D[5] = C[1,5] = \infty;$

$D[6] = C[1,6] = \infty;$

- for $i=1$ to 6 do → true

2	3	4	5	6
20	15	∞	∞	∞

so, $w = 3$

$S = S \cup \{w\} \Rightarrow S = \{1,3\}$

- > for each vertex $v \in V-S$ do → true

2	4	5	6
20	∞	∞	∞

$D[2] = \min(D[2], D[3] + C[3,2]);$

for each vertex $v \in V - s$ do → false

- for $i = 6$ to 6 do → true

  but no vertices

- for $i = 7$ to 6 do → false

Time complexity :

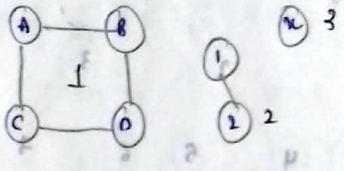
The time complexity of Dijkstra's algorithm is $O(E \log V)$ where E is the no. of edges and V is the no. of vertices.

Connected components :

Connected components is a sub graph in which any two vertices are connected to each other by paths and which is connected to no additional vertices of the super graph

(or)

A connected component of an undirected graph is a sub graph in which any two vertices are connected by path.



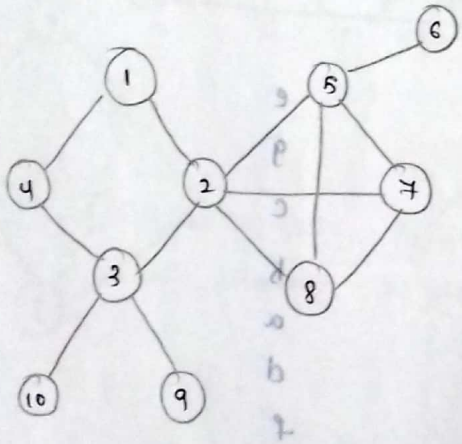
Bi connected components :

A graph is said to be Bi connected if it doesn't contain any articulation point.

Articulation point :

It is a vertex in the graph whose deletion dis-connects the graph into two (or) more non-empty components.

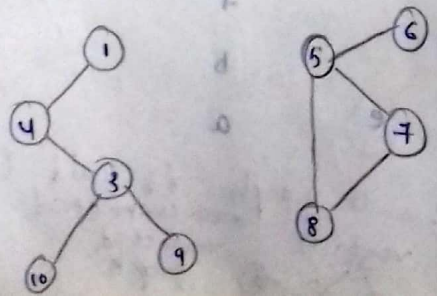
Ex :



Here the vertex '3' is an articulation point.

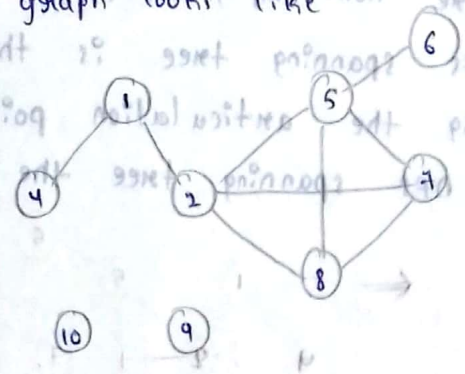
Also 2, 5 are articulation points in the graph.

* If the articulation point '2' is deleted from the graph, then the graph looks like as follows.



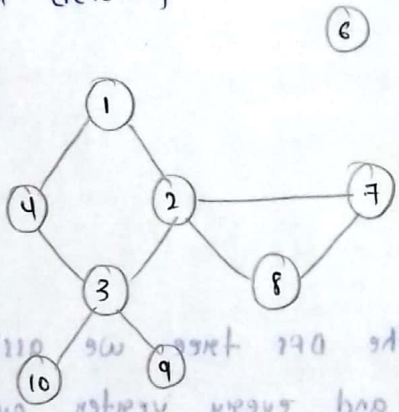
The graph is disconnected into 2 non-empty components by deleting articulation point 2.

* If the articulation point '3' is deleted then the graph looks like



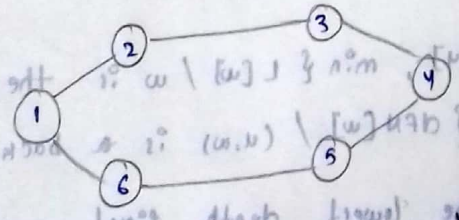
The graph is disconnected into 3 connected components after deleting '3'

* After deleting '5'



The graph is disconnected into 2 non-empty components after deleting '5'

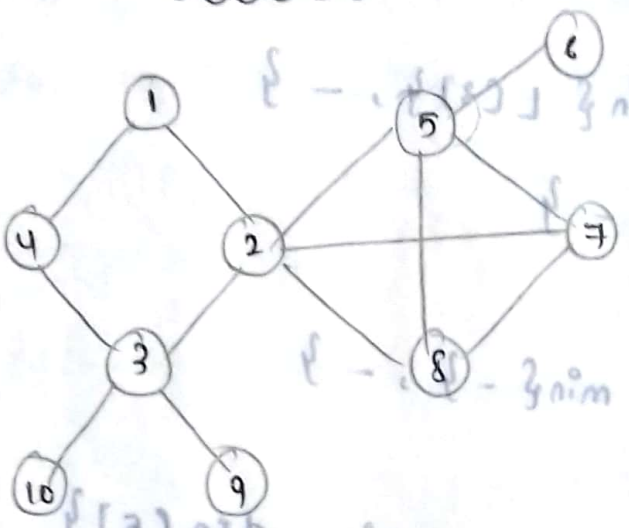
Example of Biconnected components:



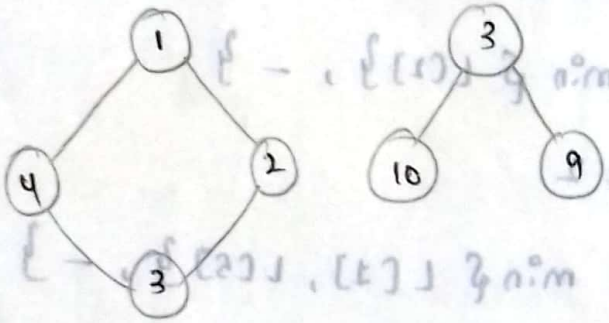
* The above is an example of Biconnected component. Because it does not contain any articulation points in the graph.

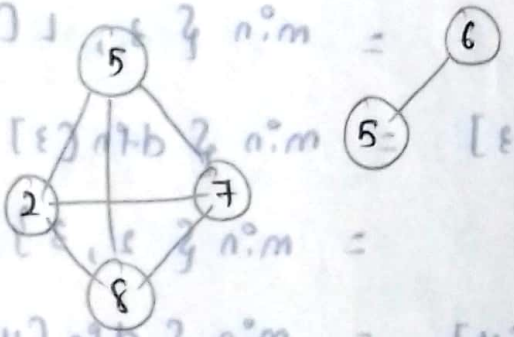
Biconnected components :

Ex :

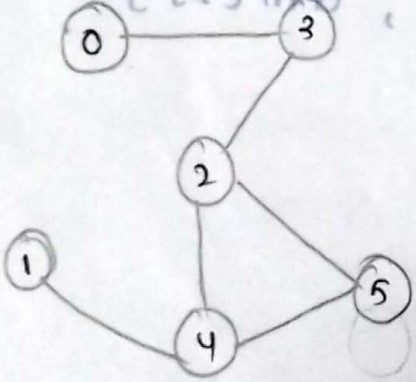


Biconnected components are :

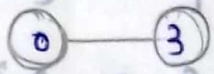


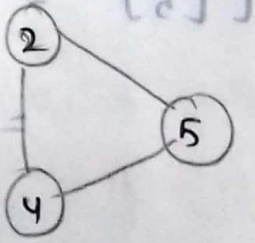


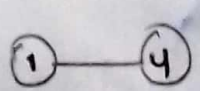
Ex 2 :



Biconnected components are :







Set: Set is a collection of elements with similar property

Disjoint sets:

Two sets are said to be disjoint if they don't have any element in common.

Ex: $S_1 = \{1, 2, 3, 4\}$, $S_2 = \{5, 6\}$

$S_1 \cap S_2 = \emptyset$

S_1, S_2 are disjoint sets

Disjoint set operations:

Union: The union of the two sets is a set containing all the elements of the first set and all the elements of the second set.

Ex: $S_1 = \{1, 7, 8, 9\}$, $S_2 = \{5, 2, 10\}$

$S_1 \cup S_2 = \{1, 7, 8, 9, 5, 2, 10\}$

Find:

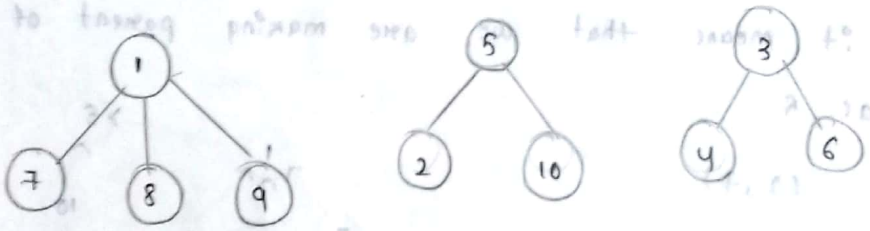
Find means finding of the set to which the element belongs to.

Ex: 1) Element 7 belongs to set 1

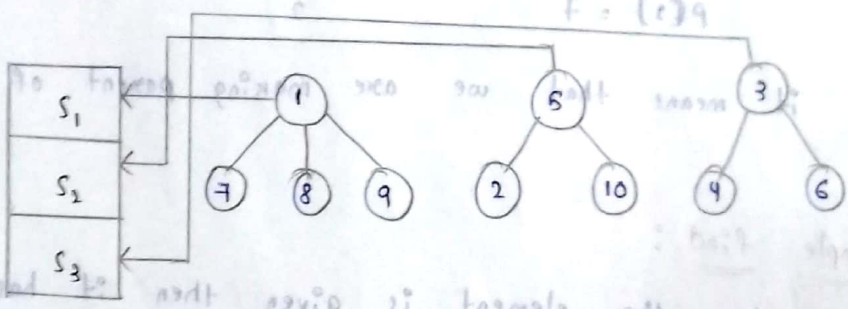
2) element 10 belongs to set 2 (in the above ex)

Representation of sets in memory:

Tree representation:



Data representation:



Array representation:

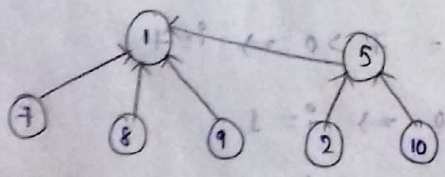
Here 'i' is the element and 'p' is the parent of the element. For the root element, parent is -1.

i	1	2	3	4	5	6	7	8	9	10
p	-1	5	-1	3	-1	3	1	1	1	5

Simple union:

The union of the two trees can be formed by making one tree as the subtree of other tree.

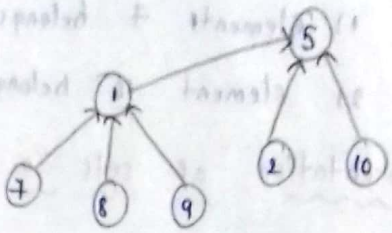
Ex: $S_1 = \{1, 7, 8, 9\}$ and $S_2 = \{5, 2, 10\}$ then $S_1 \cup S_2$



Algorithm :

```
Algorithm union (i,j)
{
    P[i] = j;
}
```

Tracing :



Ex : i=1, j=5

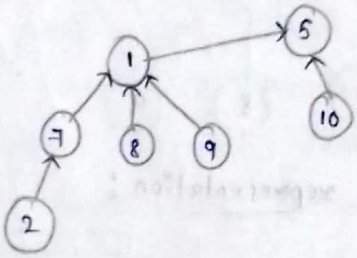
P[1] = 5

it means that we are making parent of 1 as 5.

Ex2 : (2,7)

i=2 and j=7.

P[2] = 7



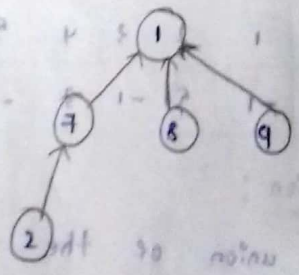
it means that we are making parent of 2 as 7.

Simple find :

when the element is given then it has to return the root of the element.

Algorithm :

```
Algorithm simplefind (i)
{
    while ( P[i] ≥ 0)
    {
        i = P[i];
    }
    return i;
}
```



Tracing : i=2

P[i] = P[2] = 7 > 0 ⇒ i=7.

P[7] = 1 > 0 ⇒ i=1

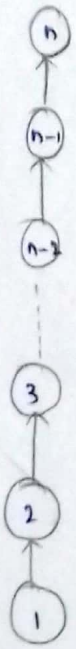
P[1] = -1 < 0 ⇒ i=1

∴ parent of 2 is 1.

Degenerate tree :

* The tree is called as degenerate tree
* The drawback here is searching takes more time
* To overcome this, weighted union is introduced.

union (1, n)



Ex : Let us assume that

{1}, {2}, {3}, {4} --- {n-1}, {n}

are sets with single element.

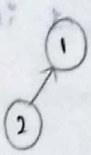
Then, union (1, 2)

union (1, 3)

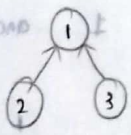
22/07/19

Weighted union :

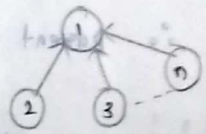
union (1, 2)



union (1, 3)



union (1, n)



Algorithm for weighted union :

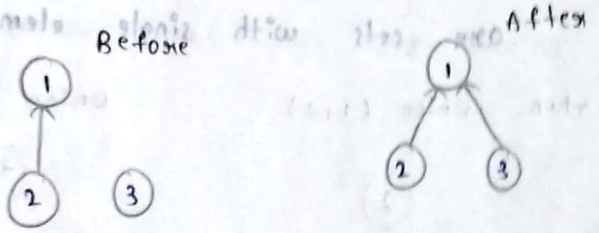
```
Algorithm weightedunion (i, j)
{
    temp = p[i] + p[j];
    if ( p[i] > p[j]) then
    {
        p[i] = j;
        p[j] = temp;
    }
    else
    {
        p[j] = i;
```

```
        P[i] = temp;
    }
}
```

* Here the weight of the tree is nothing but no. of nodes in the tree. The tree i.e., having more weight remains as the root and the tree i.e., having less weight becomes as the child of the more weighted tree.

Tracing:

Ex 1:



i = 1 , j = 3

P[i] = P[1] = -2 P[j] = P[3] = -1

temp = P[i] + P[j] = -2 - 1 = -3

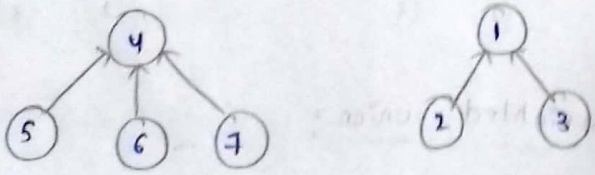
P[i] i.e., -2 > P[j] i.e., -2 > -1 (false)

else condition i.e., P[j] = P[3] = i = 1

P[i] = temp = -3

∴ parent of 3 is 1, and parent of 1 is -3

Ex 2:



i = 1, j = 4

P[i] = P[1] = -3, P[j] = P[4] = -4

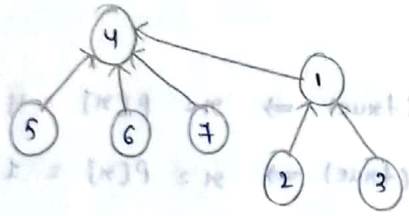
temp = P[i] + P[j] = -7

P[i] > P[j] i.e., -3 > -4 (true)

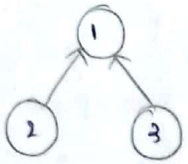
then P[i] = j => P[1] = 4

P[j] = temp => P[4] = -7.

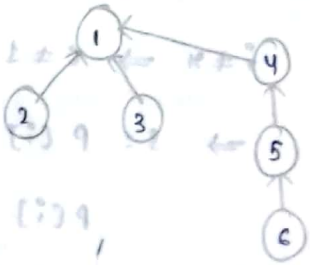
∴ Tree is



Ex 3:



Final



i = 1, j = 4

P[i] = P[1] = -3 P[j] = P[4] = -3

temp = P[i] + P[j] = -3 - 3 = -6.

P[i] > P[j] (false)

then, P[j] = i ⇒ P[4] = 1

P[i] = temp ⇒ P[1] = -6.

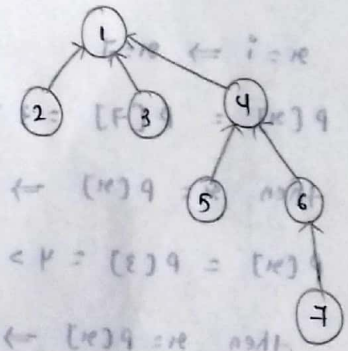
23/07/19

Collapsing find:

To find the root of the tree.

Algorithm:

Algorithm collapsing find (i)




```
{
  r = i;
  while ( P[r] > 0)
      r = P[r];
  while ( i ≠ r) do
  {
      s = P[i];
      P[i] = r;
      i = s;
  }
  return s;
}
```

Tracing : $i = 6$

$x = i \Rightarrow x = 6$

$P[x] = P[6] = 4 > 0$ (true) $\Rightarrow x = P[x] = 4$

$P[x] = P[4] = 1 > 0$ (true) $\Rightarrow x = P[x] = 1$.

$P[x] = P[1] = -7 > 0$ (false) loop terminates

Now $i = 6$, $x = 1$

$i \neq x \Rightarrow 6 \neq 1$ (true)

$\Rightarrow s = P[i] \Rightarrow s = P[6] = 4$

$P[i] = x \Rightarrow P[6] = 1$.

$i = s \Rightarrow i = 4$

Now $i = 4$, $x = 1$

$i \neq x \Rightarrow 4 \neq 1$ (true)

$\Rightarrow s = P[i] \Rightarrow s = P[4] = 1$

$P[i] = x \Rightarrow P[4] = 1$

$i = s \Rightarrow i = 4$.

Now, $i = 1$ and $x = 1$

$i \neq x \Rightarrow 1 \neq 1$ (false) loop terminates.

return s i.e., $s = 1$.

24/07/19

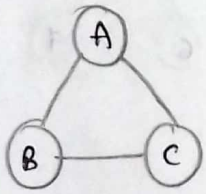
Spanning trees:

It is a connected undirected subgraph without any cycle and should not contain all the vertices of the given graph.

* The no.of spanning trees possible for a complete graph with n vertices is "n^{n-2}".

Examples:

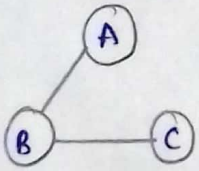
1)

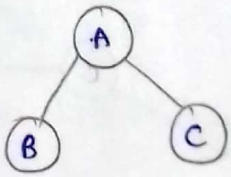


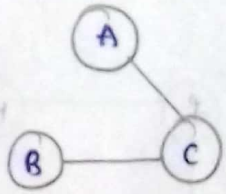
No.of vertices (n) = 3

No.of spanning trees = $3^{3-2} = 3$

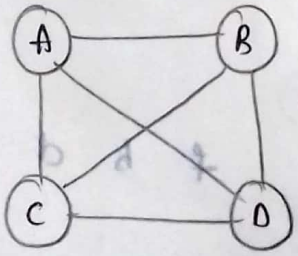
Spanning trees:





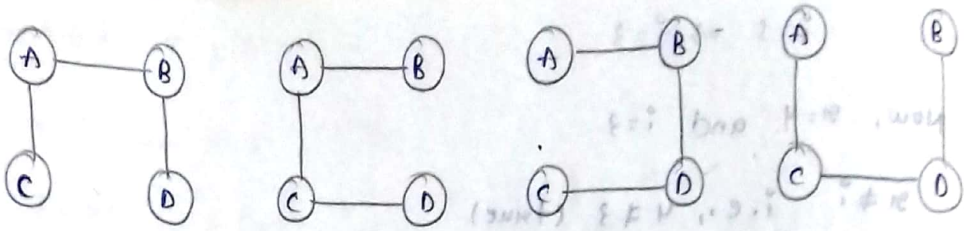


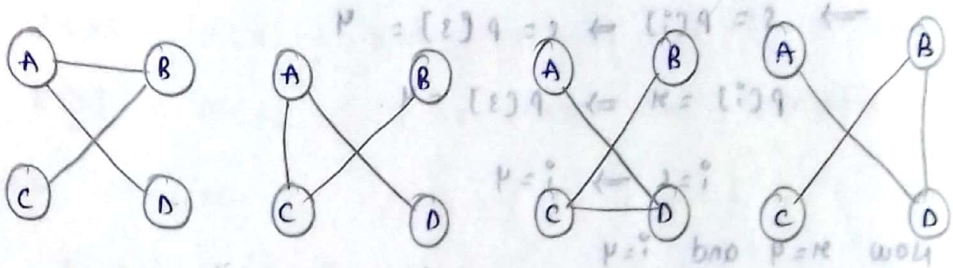
2)

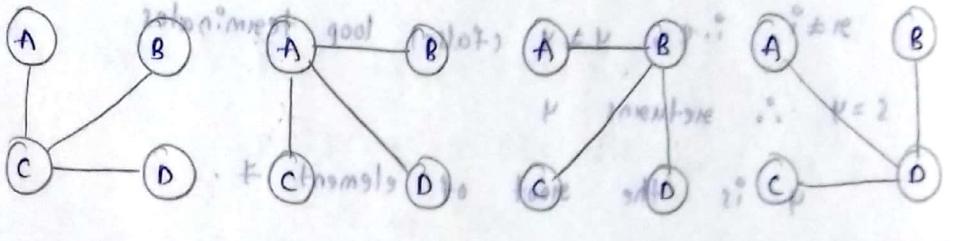


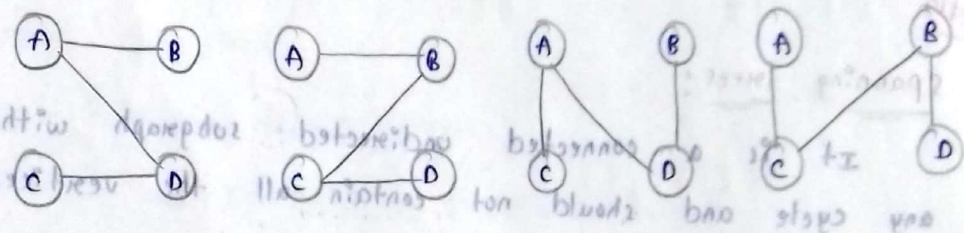
No.of vertices (n) = 4

No.of spanning trees = $4^{4-2} = 16$

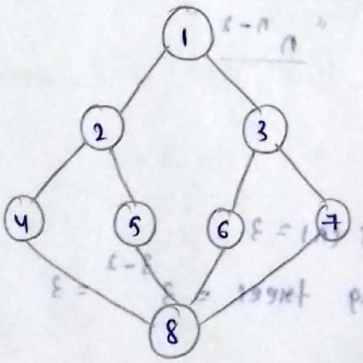


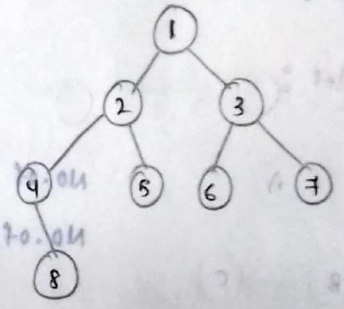




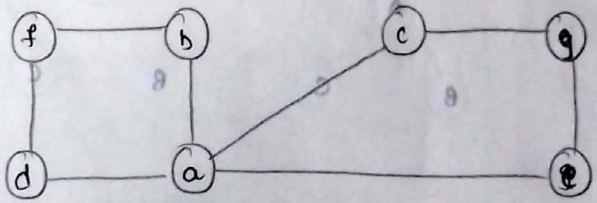


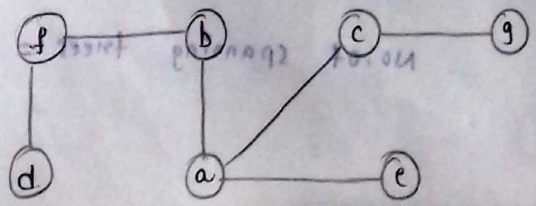
✳ BFS (Breadth first search tree) :





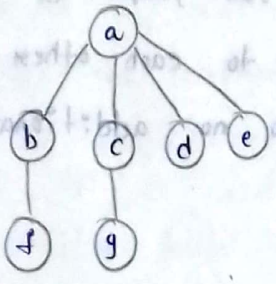
Ex 2 :





f b d a c e g

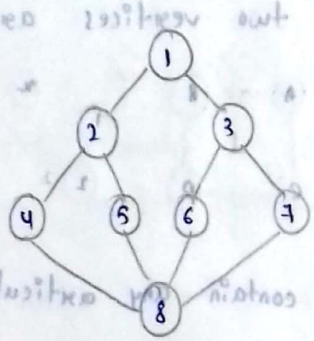
* More than 1 BFS is possible for the graph

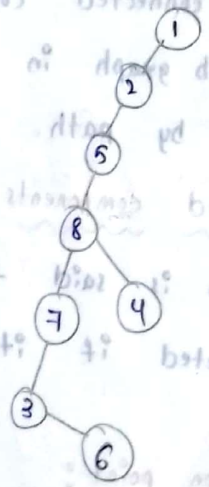


a b c d e f g

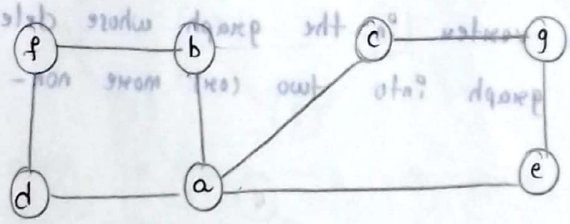
* Depth first search tree (DFS)

Ex 1:

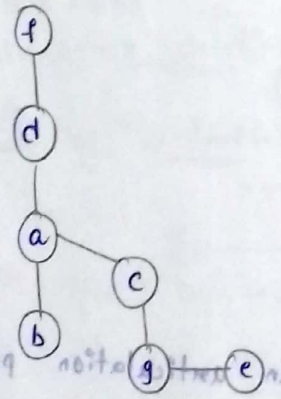




Ex 2:

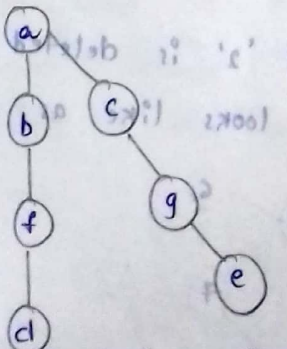


1)

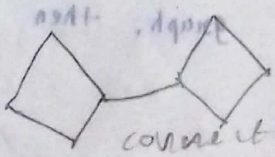


e g e b a d f

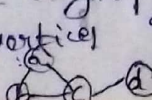
2)

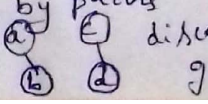


e g c d f b a



Connected Graph:- a graph is said to be connected in which any two vertices are connected by paths

ex:-  connected graph

 disconnected graph