UNIT-II

Part-1 :- Disjoint Sets :-

* Two/more sets are said to be disjoint if they have no elements in common.

Ex:- $S_1 = \{1,7,8,9\}$
$S_2 = \{2,5,10\}$
$S_3 = \{3,4,6\}$

$\therefore S_1, S_2, S_3$ are disjoint sets.
$\because S_1 \cap S_2 \cap S_3 = \phi$

Representations of a Set :-

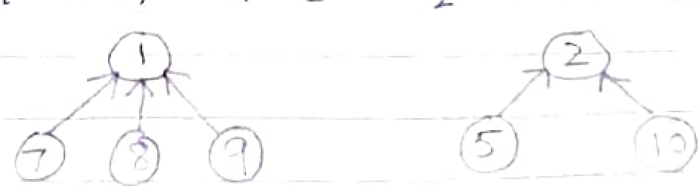
There are three Representations, viz:-
(i) Tree
(ii) Data
(iii) Array

(i) Tree Representation:-

* The first element of a set is the root and remaining elements are the direct childrens of the root.

* The Edge is in the direction of a parent from its child node.

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



(ii) Data Representation:-

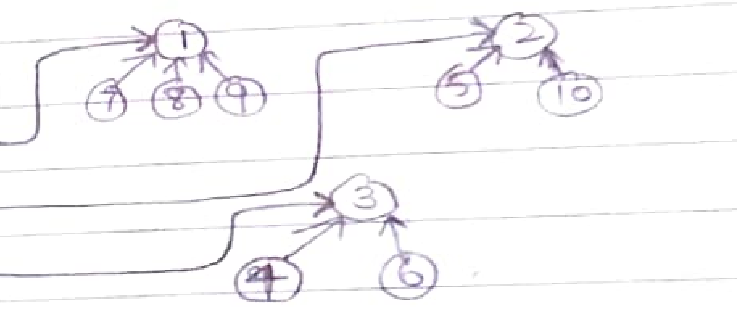
* ~~An Array is used to store the parent of each element.~~

* ~~For a root node, the parent value will be (-1).~~

* The address of root node of a set (in its tree reprentation) is stored in an array.
  i.e., Each address in the array points to the root node of a set.

Ex)—

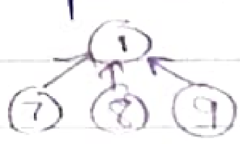
Set	Pointer
S_1	•
S_2	•
S_3	•



(iii) Array Representation :—

* An Array is used to store the parent of each Element.

* for a root node, the parent (P) will be (−1).



Ex)—

Element (i)	1	7	8	9
Parent (P[i])	−1	1	1	1

Disjoint sets operations:-

The two major operations that can be performed b/w two disjoint sets, are:-

(i) Union
(ii) Find.

(i) Union operation:-

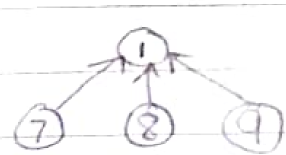
If (S_1) and (S_2) are two disjoint sets, the the union $(S_1 \cup S_2)$ is a set containing an element (x) such that, (x) is in either (S_1) or (S_2).

i.e., $S_1 \cup S_2 = \{x : x \in S_1 \text{ (or) } x \in S_2\}$

* In tree representation, the union $(S_1 \cup S_2)$ is achieved by setting the parent of the root of (S_2) as the root of (S_1) (or) vice-versa.

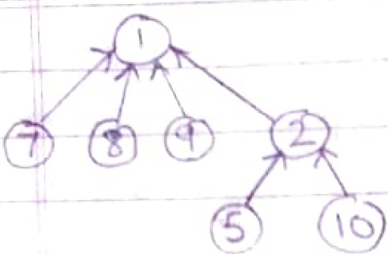
Ex:-

$S_1 = \{1, 7, 8, 9\}$

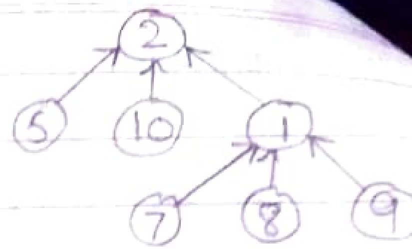


$S_2 = \{2, 5, 10\}$

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



[or]



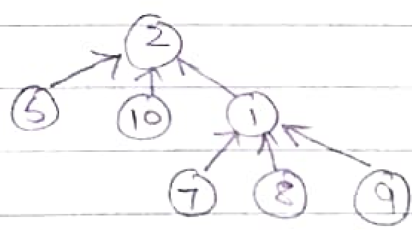
(ii) Find Operation:-

This operation is performed on an Element to find its root node.

[Determining which set, a particular element (x) belongs to.]

i.e., find(x) represents that (x) is an Element of a set, passed as an input and Expected to return the root of the respective tree in which the Element (x) resides.

Ex:-



→ find(5) = 2
→ find(1) = 2
→ find(7) = 2
→ find(9) = 2

//

Union and Find Algorithms:-

① Algorithms for union operation:-
there are two Algorithms, viz:-
(a) Simple union
(b) Weighted union

[both Algorithms follows tree representations]

(a) Simple Union Algorithm :-
In this Algorithm, the union $(S_1 \cup S_2)$ is achieved by setting the parent of the root of (S_2) as the root of (S_1) (or) vice-versa.

```
SimpleUnion (r1, r2) {
        P[r1] := r2;
}
```

Ex:-

from (Ex) of union operation :-

$r_1 = 1$, $r_2 = 2$ [or] $r_1 = 2$, $r_2 = 1$

$\therefore P[r_1] = r_2$ [or] $\therefore P[r_1] = r_2$

$\Rightarrow P[1] = 2$ $\Rightarrow P[2] = 1$

(b) Weighted union Algorithm :-
This Algorithm is the efficient version of simple union Algorithm, in which the tree having less no. of nodes, is choosen as the sub-tree.
i.e., weight of a tree is nothing but the no. of nodes of the tree (and) the tree with lower weight, is choosen as a subtree.

* In this Algorithm, the parent of a root node represents the no. of nodes of the respective tree [in negative value]

i.e, $P[r] = -$(no. of nodes in the tree)
$(r) \Rightarrow$ root of a tree

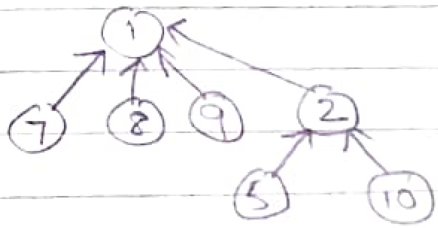
```
WeightedUnion (r1, r2){
    r3 := P[r1] + P[r2];
    if (P[r1] > P[r2]) then {
        // r1 has fewer nodes
        P[r1] = r2;
        P[r2] = r3;
    } else {
        // r2 has fewer nodes
        P[r2] = r1;
        P[r1] = r3;
    }
}
```

Ex:-

(S_1) (S_2)

$\boxed{r_1 = 1}$ $\boxed{r_2 = 2}$

$S_1 \cup S_2 \Rightarrow$



$S_1 \Rightarrow P[r_1] = P[1] = -4$

$S_2 \Rightarrow P[r_2] = P[2] = -3$

$S_1 \cup S_2 \Rightarrow P[r_1] + [r_2] = -4 + (-3) = -7$

(and) $P[2] = 1$

x	1	7	8	9	2	5	10
P[x]	-4	1	1	1	-3	2	2
	-7	1	1	1	1	2	2

(2) Algorithms for Find Operation:-

there are two Algorithms, Viz:-

(a) Simple Find

(b) Collapsing Find.

[Both Algorithms follows tree represetation].

(a) Simple Find Algorithms:-

In this Algorithm, the aim is to return the root of the tree while an Element(x) is provided as the input.

```
SimpleFind(x){
    while(P[x] >=0)do{
        x := P[x];
    }
    return x; //root
```

* If the height of the tree is (h), then the time Complexity of the Algorithm is:- $\boxed{O(h).}$

* This Algorithm is inefficient if the height of a tree is larger.

(b) Collapsing Find Algorithm:-

This Algorithm is an Efficient version of Simple find Algorithm, in which the tree is Continuously Collapsed finding

the root.
i.e., In the Path of finding the root, Every node's parent is changed to the root node.

* As a result, the height of the tree decreases [i.e, tree will be collapsed], due to which the time complexity will be lesser in the next trial.

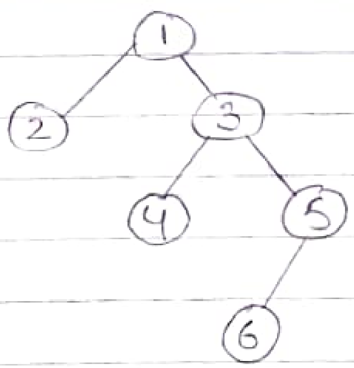
* The overall goal is to make the root of a tree as the parent of every other node in the tree [i.e, height(h) = 2]

* The root must be found first before collapsing the tree. Hence, initially, the SimpleFind Algorithm is applied to find the root.

* Therefore, we can say that this Algorithm is the futuristic Algorithm which decreases the time complexity for the future trials of finding the root.

```
CollapsingFind(x){
        r := x;
        While(P[r]>=0) do { //Find root
                r := P[r];
        }
        While(x != r) do { //Collapsing
                temp := P[x];
                P[x] := r;
                x := temp;
        }
        return r;
}
```

Ex:-



height (h) = 4

→ CollapsingFind(6)

$x = 6$

$r = 6$

Find(6) ⇒ $r = 1$

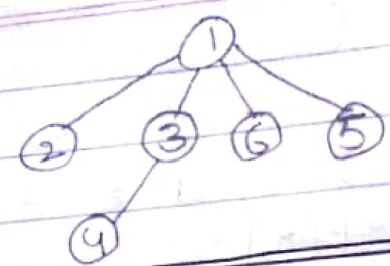
* $6 != 1$ ⇒ $temp = 5$
  $P[6] = 1$
  $x = 5$

* $5 != 1$ ⇒ $temp = 3$
  $P[5] = 1$
  $x = 3$

* $3 != 1$ ⇒ $temp = 1$
  $P[3] = 1$
  $x = 1$

$1 != 1$ ⇒ false

return (r=1).



height(h) = 3.

Part-2 : BackTracking

Backtracking is a general Problem-Solving algorithm which Searches for all possible Solutions that a given problem can have.

* The term 'Backtrack' Suggests that if the Current Solution does not Satisfies the Specified Constraints, then go back to previous Step[undo] and Explore other possibilities for Satisfiable Solution.
* Thus, Backtracking uses Recursive approach for Exploring all possible ways (paths) to find all possible Solutions.
* The 'Dynamic Programming' technique is used to find the optimal Solution from the Set of possible Solutions given by backtracking algorithm.
* Backtracking algorithm is an Efficient Version of Brute Force approach, which doest Explore the Solution which doest Satisfy the Specified Constraints.

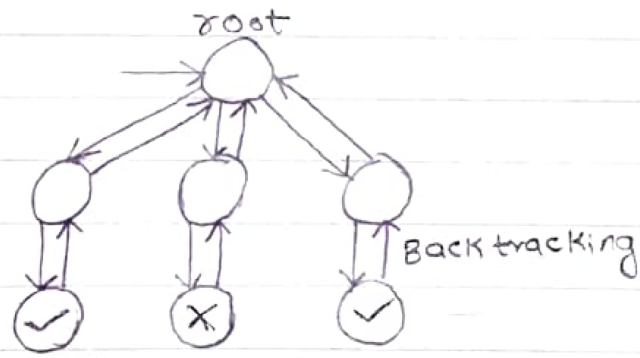
General Method:—

Backtracking Algorithms usually use tree Structure to Explore multiple Solutions, Known as State Space tree.

* The State Space tree (SST) is a tree representing all possible states (solutions and non solutions) of the problem, from the root as an initial state, to the leaf as a terminal state.

* The terminal state represents final valid / invalid solution.

* Backtracking Algorithm searches for a solution in depth-first-search manner.



(✓) ⇒ valid Soln
(X) ⇒ Invalid Soln

Ex:-

Problem:- Find all possible ways of arranging 2 boys and 1 girl on (3) Seats.

Constraint:- Girl should not be on the middle seat.

Sol^n

total possibilities :- $3! = 6$

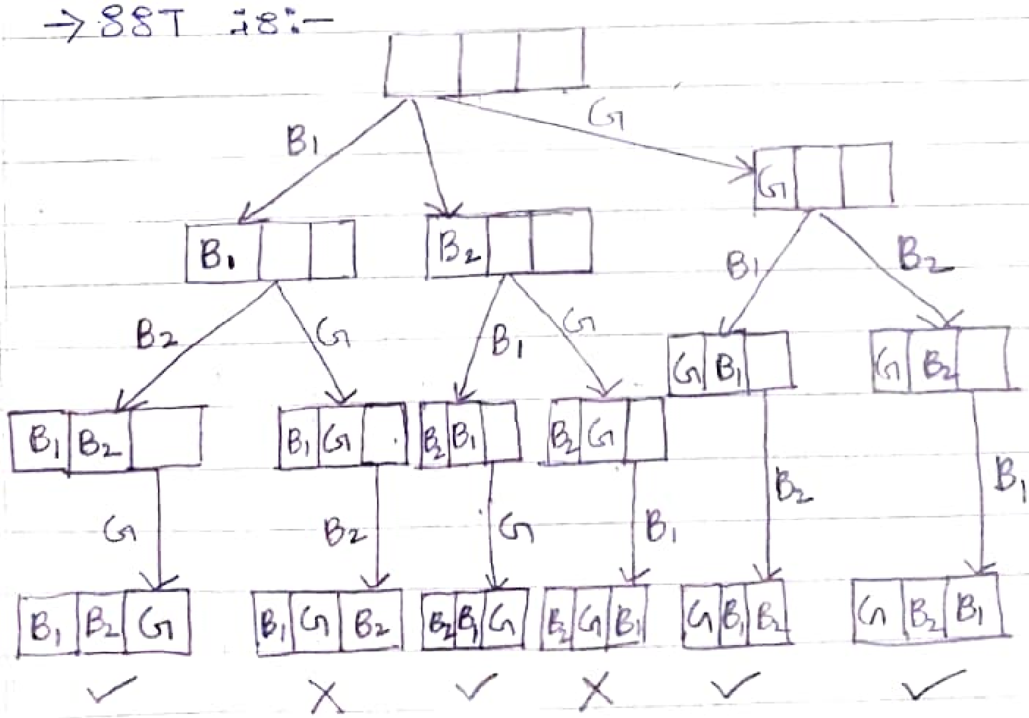
i.e., B_1, B_2, G_1 ✓ ✗ B_2, G_1, B_1

✗ B_1, G_1, B_2 G_1, B_1, B_2 ✓

B_2, B_1, G_1 ✓ G_1, B_2, B_1 ✓

∴ no. of valid sol^{n}s = 4.

→ SST is:-



SST

Applications of Backtracking Algorithm:-

1. N-Queen problem
2. Sum of subsets problem
3. Graph Coloring problem.

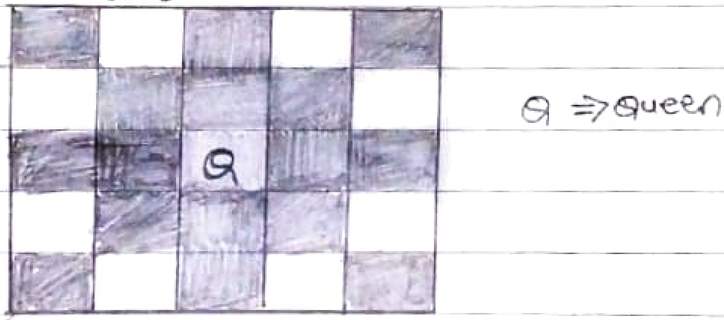
① N-Queen Problem:-

* This problem is inspired from the Chess board game.

Problem:- Get all possibilities of placing (N) queens on an (NxN) Chessboard such that, no two queens attack each other.

i.e., no two queens must be in a same row / column / in diagonal to each other.

Chess board



Possible moves of Queen

■ - Cannot place another queen

□ - Can place another queen.

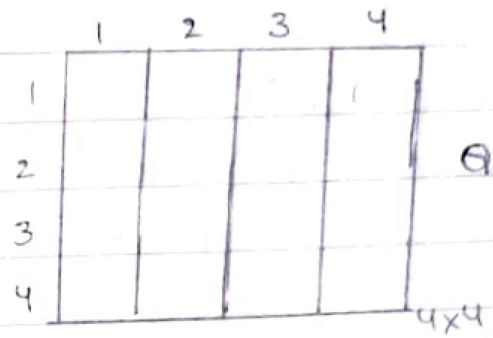
Constraints:- No two queens must attack each other.

i.e., No two queens must be in the same row / column / in diagonal to each other.

Algorithm:-

Ex:- 4-Queen Problem

Sol:-

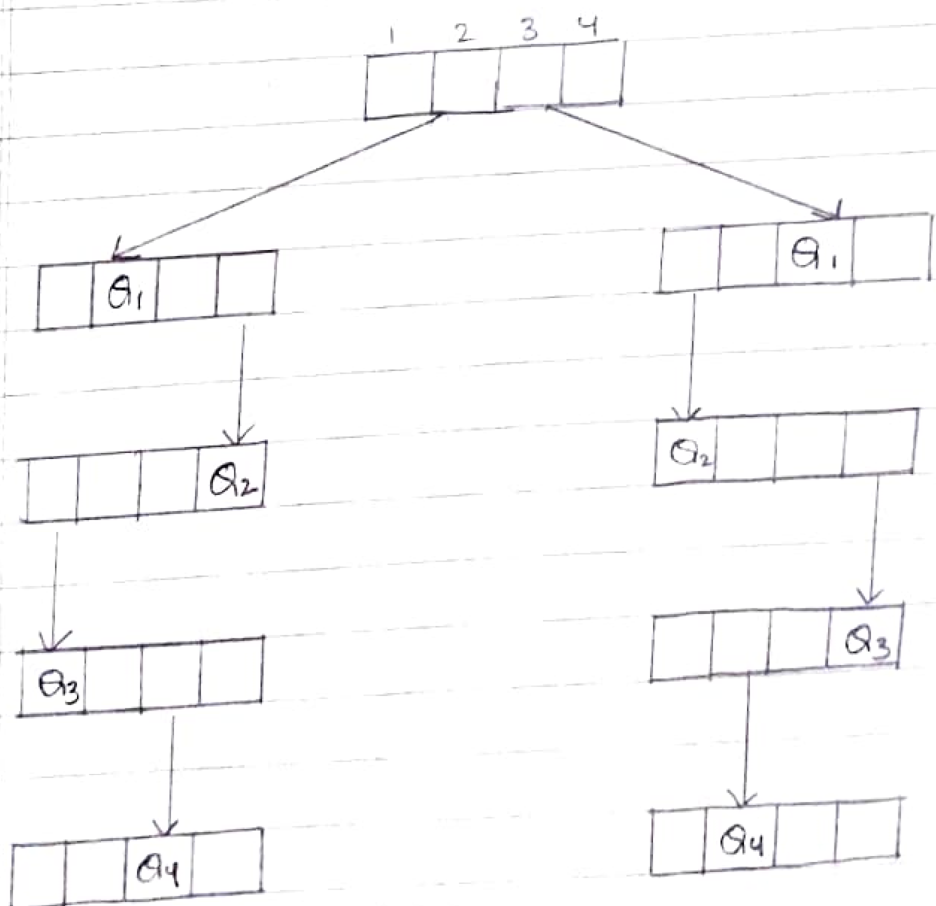


Q_1, Q_2, Q_3, Q_4

Solutions:-

Q_3	Q_1	Q_4	Q_2
1	2	3	4

Q_2	Q_4	Q_1	Q_3
1	2	3	4



SST

∴ the Solutions are:-

	1	2	3	4
1		Q_1		
2				Q_2
3	Q_3			
4			Q_4	

(1)

	1	2	3	4
1			Q_1	
2	Q_2			
3				Q_3
4		Q_4		

(2)

* Time complexity :- $O(N!)$
* Space complexity :- $O(N)$

(2) Sum of Subsets Problem :-

Problem :- Given a set of possitive integers, find the possible non-Empty subsets of any length ($0 < |subset| < |S|$), whose sum is equal to a given target sum.

Constraint :- The sum of elements of subset must be equal to the given target sum.

Algorithm :-

	1	2	3	4
1		Q_1		
2				Q_2
3	Q_3			
4			Q_4	

(1)

	1	2	3	4
1			Q_1	
2	Q_2			
3				Q_3
4		Q_4		

(2)

* Time Complexity :- $O(N!)$
* Space Complexity :- $O(N)$

② Sum of Subsets Problem:-

Problem:- Given a set of possitive integers, find the possible non-Empty subsets of any length $(0 < |subset| < |S|)$, whose sum is equal to a given target sum.

Constraint:- The sum of Elements of subset must be Equal to the given target sum.

Algorithm:-

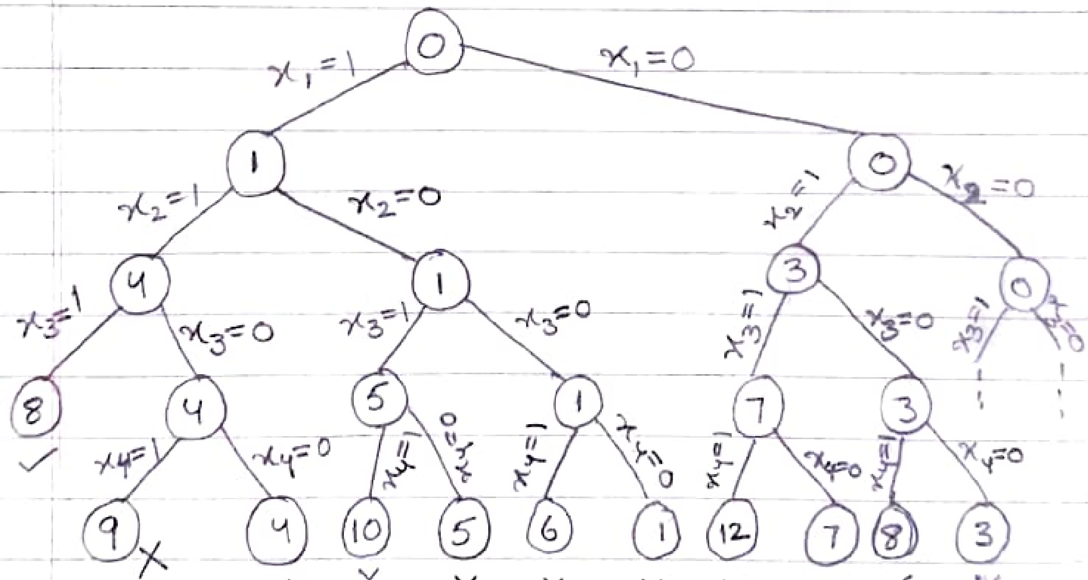
<u>Ex:-</u> $S = \{1, 3, 4, 5\}$ target sum = 8

possible ~~subsets~~ satisfying target sum:-

$S_1 = \{1, 3, 4\}$, $\{3, 5\} = S_2$

$\therefore (S_1)$ & (S_2) are two possible sol^ns

→ <u>SST</u>:- let $x_1 = 1$, $x_2 = 3$, $x_3 = 4$, $x_4 = 5$



(1) => include

(0) => Exclude

* time Complexity = $O(2^n)$
* Space Complexity = $O(n)$

③ Graph Coloring Problem:-

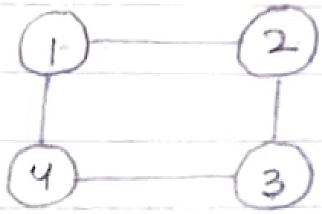
<u>Problem</u>:- Given a set of colors and an undirected graph $G = (V, E)$,

Find all possible ways of assigning Colors to the vertices of the graph(G), Such that no two adjacent vertices have same Color.

Constraint :- No two adjacent vertices of the graph (G) must have same Color.

Algorithm :-

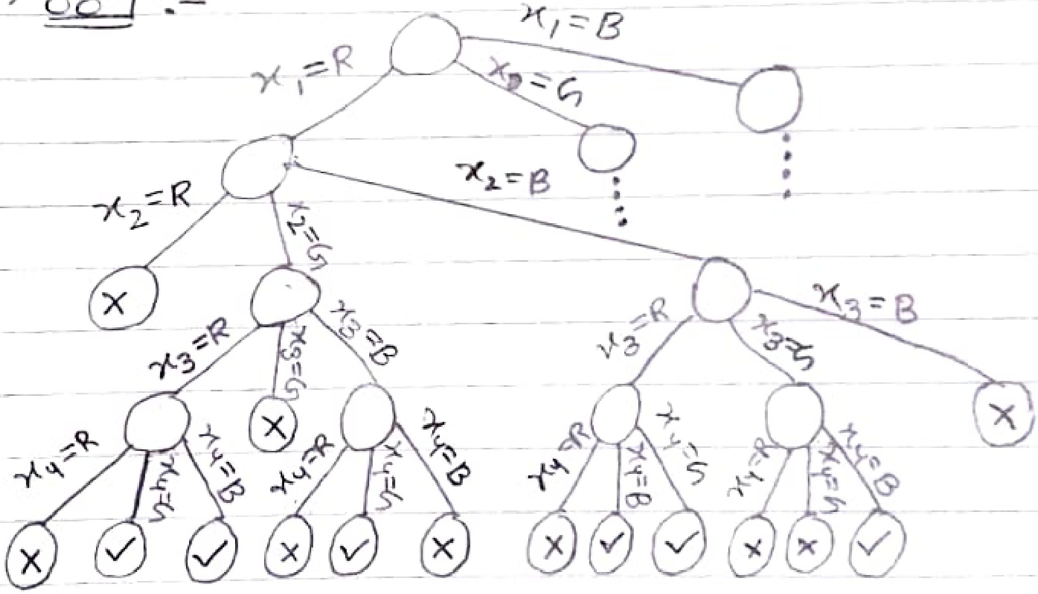
Ex:-



R ⇒ Red
G ⇒ Green
B ⇒ Blue.

Colors = {R, G, B}

→ SST:-



∴ Some of the possible solutions are:-

{1:R, 2:G, 3:R, 4:G}
{1:R, 2:G, 3:R, 4:B}
{1:R, 2:G, 3:B, 4:G}
{1:R, 2:B, 3:R, 4:B}
{1:R, 2:B, 3:R, 4:G}
{1:R, 2:B, 3:G, 4:B} ...

//