Unit -2

Disjoint sets :-

The disjoint set data structure is also known as union-find data structure and merge-find set. It is a data structure that contains a collection of disjoint (or non-overlapping sets.

set :- A set is a collection of some elements.

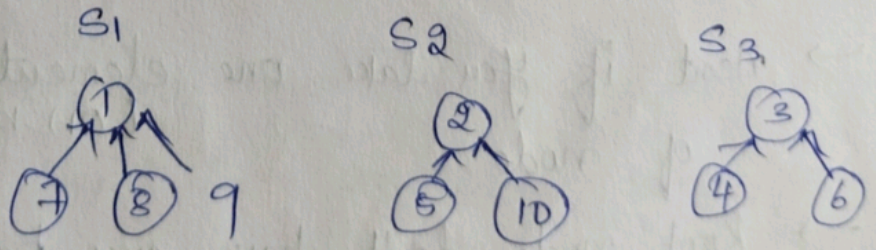
Sets are represented by pair wise **disjoint sets**, if S_i and S_j are two sets and $i \neq j$, then there is no common element for S_i and S_j

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

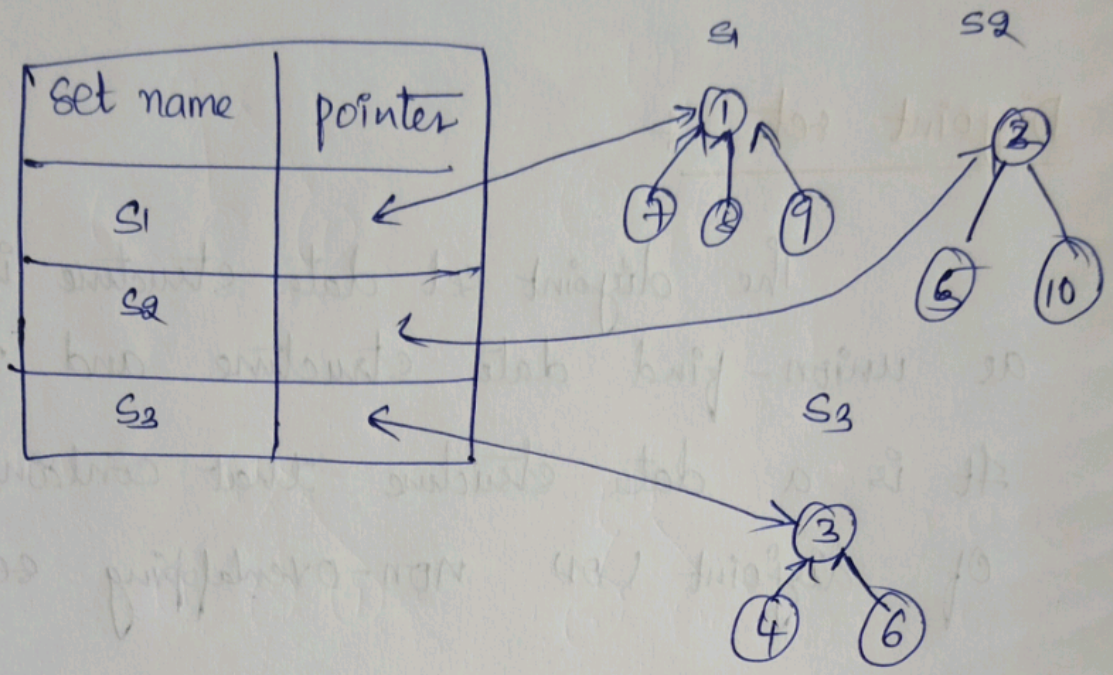
Representations :-

→ Tree representations $S_1 = \{1, 7, 8, 9\}$ $S_2 = \{2, 5, 10\}$

$S_3 = \{3, 4, 6\}$



→ Data Representations



In data representation, pointer of each set has to represent the root node of the tree

→ Array representation

array representation

	1	2	3	4	5	6	7	8	9	10
P[i]	-1	-1	-1	3	2	3	1	1	1	2

→ Take all the elements in a set

→ ment if you take one element, identify the parent (or) node of node

→ Root node don't have any parent.

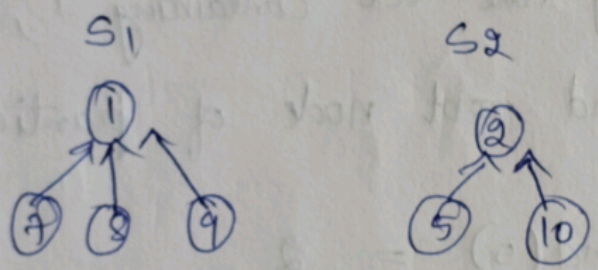
→ Parent of Root node can be represented as -1.

disjoint set operations :-

- Union
- find

Union :- If S_i and S_j are two disjoint sets, then their union is $S_i \cup S_j = \{x / x$ is in S_i or $S_j\}$

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



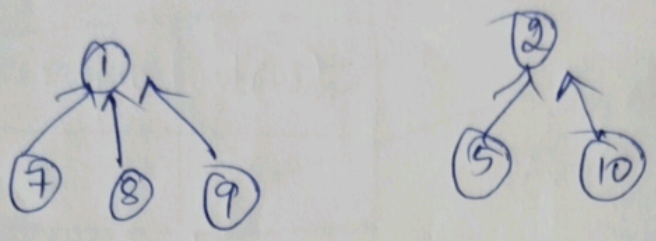
$S_1 \cup S_2 = \{1, 7, 8, 9, 2, 5, 10\}$

Tree representation of $S_1 \cup S_2$

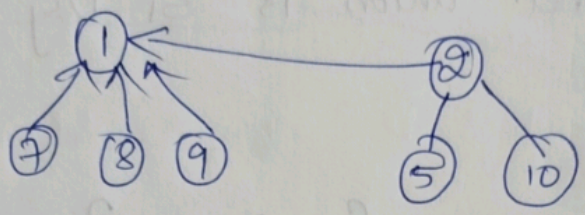
→ As $S_1 \cup S_2$ is one set, so we need to have one root node.

→ we any one root node from two sets can be made as child to another node

ex:- S_1 S_2



→ Union



$S_1 \cup S_2$

find (i) :- finding the set containing i;

used to find root node of particular node.

ex:- find(5) = 2

find(9) = 1

→ Initially we have to consider every element as a seperate element.

→ find P[i] value, parent of node i

→ for root node P[i] = -1

→ After performing union update P[i] values

Algorithms

- Simple union
- Simple find
- weighted union
- collapsing find

→ Simple Union :-

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

$i = 1,\ j = 2$

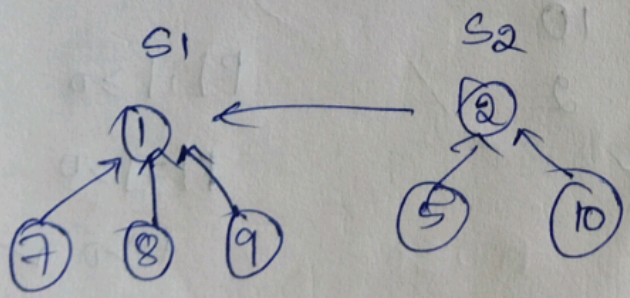
$P[1] = 2 \Rightarrow$ root

$i = 2,\ j = 1$

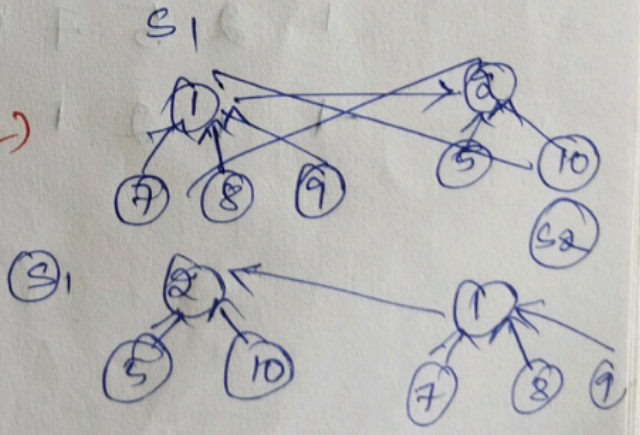
$P[2] = 1 \Rightarrow$ root

→ In simple union if two sets are there, for union make any one root node from two sets as root node.

→ No restrictions (or) constraints



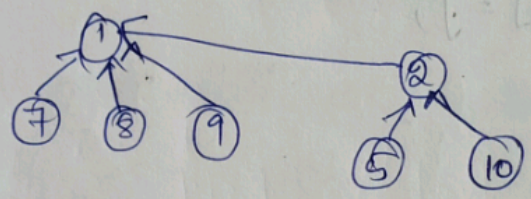
(or)

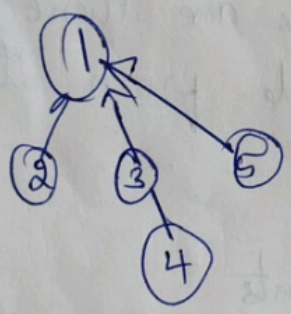


→ Simple find :-

```
Algorithm simplefind(i)
{
 while ( P[i] ≥ 0) do
 i := p[i];
 Return i;
}
```

example :-





1	2	5	7	8	9	10
-1	1	2	1	1	1	2

i = P[i]
i = -1
return i = -1

P[4] ≥ 0
3 > 0 ✓
i = P[i]
i = 3
P[3] ≥ 0
1 > 0 ✓

P[i] ≥ 0
P[3] > 0
1 > 0 ✓
i = P[i]
i = 1
return i
1 > 0
i = P[3] = 1

i = 4
P[4] = 3
3 > 0 ✓ i = 3
P[3] = 1
1 > 0 ✓
i = 1
P[i] = -1
-1 > 0 ✗

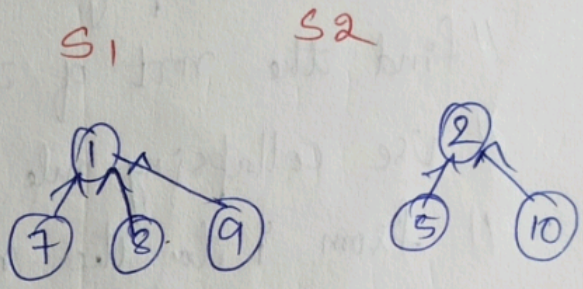
ex :- i = 5

P[5] ⇒ 2
2 > 0 ✓
i = P[i]
i = 2
while P[2] = 1
1 > 0 ✓
i = P[2]
i = 1
⇒ P[i] > 0
P[i] = -1
-1 > 0
comes out of loop
Return i

Result = 1

→ weighted Union :-

weighting rule for union(i, j) :- If the number of nodes in the tree with root i is less than the number in the tree with root j; then make j as the parent node.



Algorithm weightedUnion(i, j)

// union sets with roots i and j, i ≠ j

// weighted rule, P[i] = -count[i] & P[j] = -count[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;
  }
}
```

ex:- ~~i = 1, j = 2~~

P[1] = ~~-1~~ P[2] = -3

temp = -7

1	2	3	4	5	6
-4	1	1	1	-2	5

Collapsing Find :

collapsing rule : If j is a node on the path from i to its root and $p[i] \neq root[i]$, then set $p[j]$ to root.

```
Algorithm CollapsingFind(i)
// find the root of the tree containing element i.
// Use collapsing rule to collapse all nodes
// from i to the root.
{
  r := i;
  while (p[r] > 0) do r := p[r]; // finds the root.
  while (i ≠ r) do // collapse nodes from i to root r.
  {
  s := p[i] ; p[i] := r ; i := s;
  }
  return r;
}
```

while. $7 \neq 1$

$s = 6 \quad p[7] = 1 \quad i = 6$

$s = 4 \quad p[6] = 1 \quad i = 4$

$i = 7, \ r = 7$

$p[r], \ p[7] > 0$

$6 > 0$ ✓

$r = 6$

$p[6] > 0$ ✓

$4 > 0$ ✓

$r = 4$

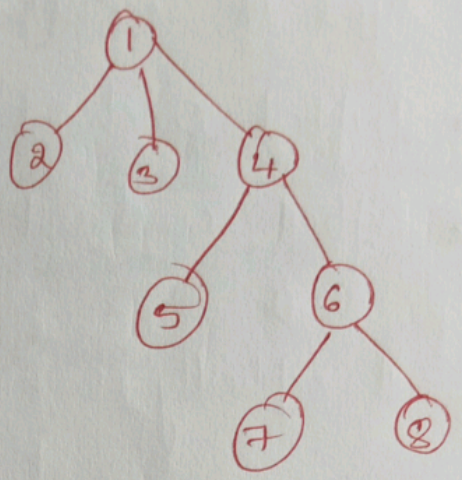
$p[4] = 1$

$1 > 0$ ✓

return r i.e, '1'

$p[i] = -1$

find(7) = 1



Simple find :

find(7) = 6

P(6) = 4

P(4) = 1

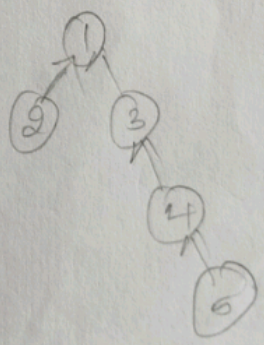
In Three steps we can find find(7)

⇒ 20 times find(7) i.e, it will take 20×3 i.e, 60 steps or units

In collapsing find ex:- find(7)

→ first find the parent of 7 i.e, '6', then it will collapse node '6'

→ next again parent of 6 i.e, then it will collapse node '4'.



ex :-

s = 6

P[7] = r = 1

i = 6 i ≠ r

s = P[i] = P[6] = 4

P[6] = 1

i = 4 4 ≠ 1

s = P[4] = 1

P[4] = r = 1 i = s ⇒ 1 = 1, return r

Backtracking :- [General Method]

It is used to solve problems in which a sequence of objects is choosen from a specified set so that the sequence satisfy some criteria.

When to use backtracking :-

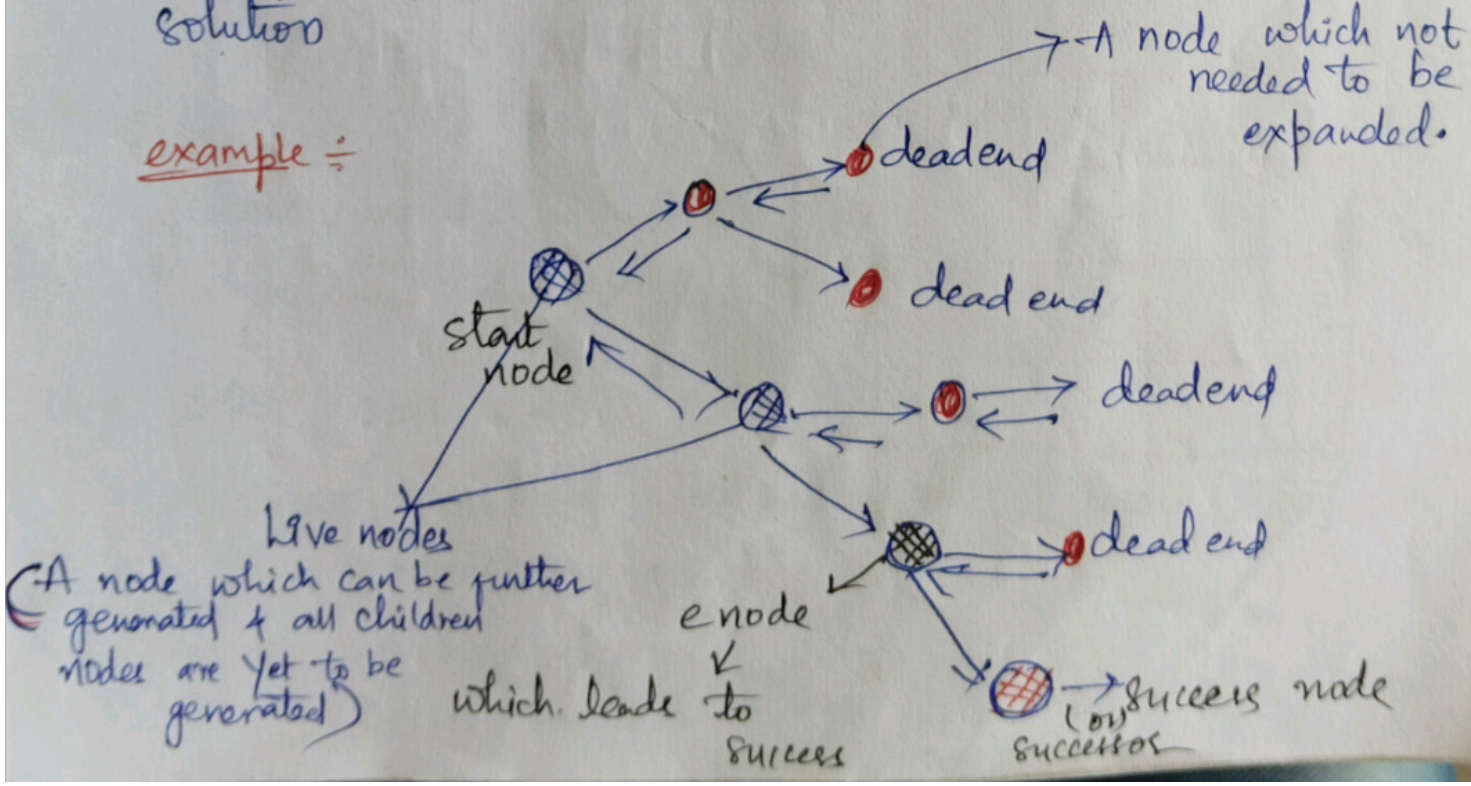
If no of choices are provided, a decision need to be made from available choices where sufficient information is not avaible on best choice.

each decision leads to new set of choices. to get optimal solution.

How it works ?

Backtracking is a systematic method of trying out various sequences of decisions, until we find optimal solution

example :-



2 types of constraints in backtracking

→ Implicit constraint : These are the rules that specifies how each element in the tuple should be related

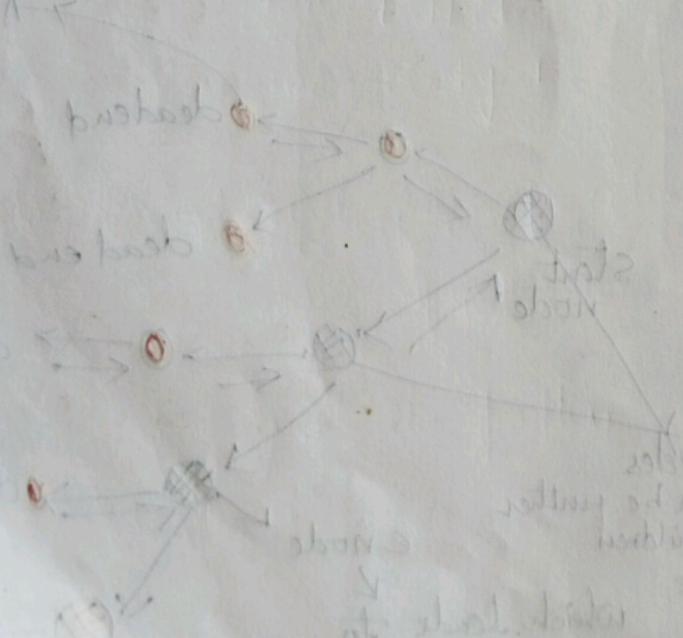
→ explicit constraint

↓

it explicitly says that rules that restrict each element to be choosen from a set

Applications :-

- n- queen problem
- sum of subsets
- graph coloring
- Hamilton

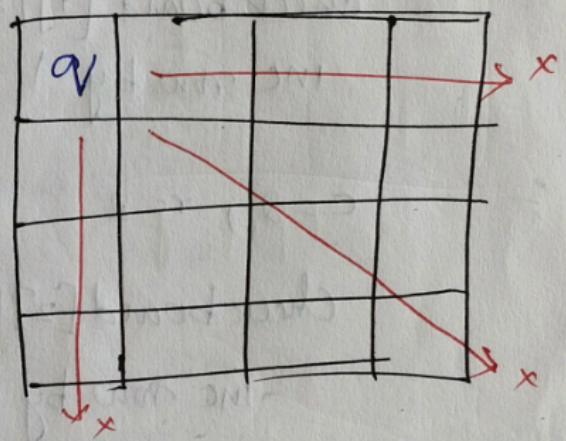


N- Queen Problems

N-Queens problem is to place n-queens in such a manner on an n×n chessboard so that no queens attack each other by being in the same row, column or diagonal.

* Note :- No two queens are placed in same row, same column, same diagonal.

example :-



4 queen problem

4×4 chess board.

step 0 :- c = 0, r = 0

Check, board[0][0] = safe

increment column by 1

step 1.0 :- c = 1, r = 0

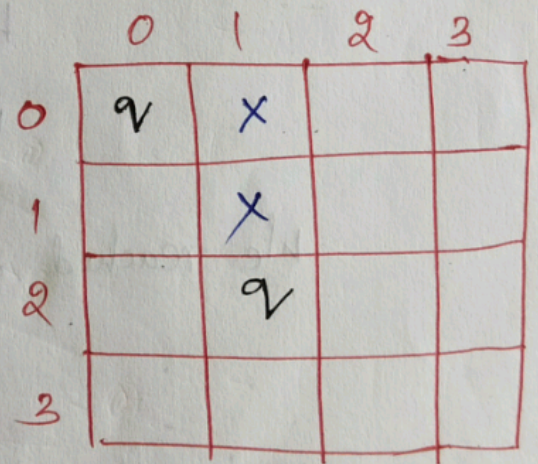
check board[0][1] = not safe

increment row by 1

step 1.1 :- c = 1, r = 1

check board[1][1] = not safe

increment row



board [r][c]

step 1.2 :- c=1, r=2

board[2][1] = safe

increment column by 1

step 2.0 :- c=2, r=0

Check board [0][2] not safe

inc row by 1

step 2.1 :- c=2, r=1

check board [1][2] = not safe

inc row by 1

step 2.2 :- c=2, r=2

Check board [2][2] = not safe

inc row by 1

step 2.3 :- c=2, r=3

board[3][2] = not safe

inc row by 1

We reached last row of column 2. There is no safe position.

Again backtrack the problem.

Note :- while performing backtracking, remove last placed queen

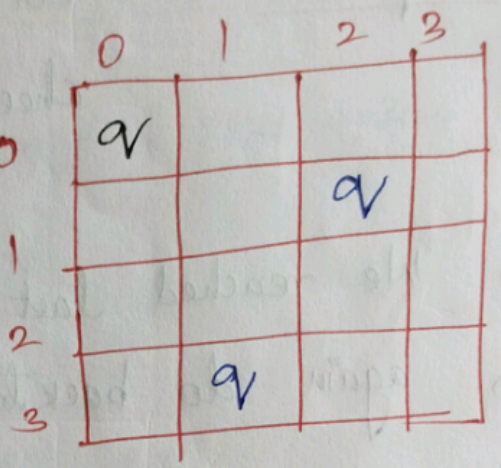
i.e, board[2][1] removed

new row to be checked = 2+1 = 3 i.e, inc row by 1

step 1.3 :- col=1, row=3

board[3][1] = safe

inc column by 1



step 2:0 :- col=2, row=0

Chek board[0][2] = not safe

inc row by 1

step 2:1 :- col=2, row=1

check board[1][2] = safe.

increment column by 1

step 3:0 :- col=3, row=0

Check board[0][3] = not safe

inc row

step 3:1 :- col=3, row=1

check board[1][3] = not safe

inc row

step 3.2 :- col = 3, row = 2

check board[2][3] = not safe

inc row by 1

step 3.3 :- col = 3, row = 3

check board[3][3] = not safe

inc

→ We reached last row & last colounmn

→ again do backtracking

→ Remove last placed queen.

i.e, board[1][2] → remove

inc row by 1.

i.e, col = 2, row = 1

board[1][2] ≠ not safe

inc row by 1

col = 2, row = 3

board[3][2] ≠ not safe

inc row

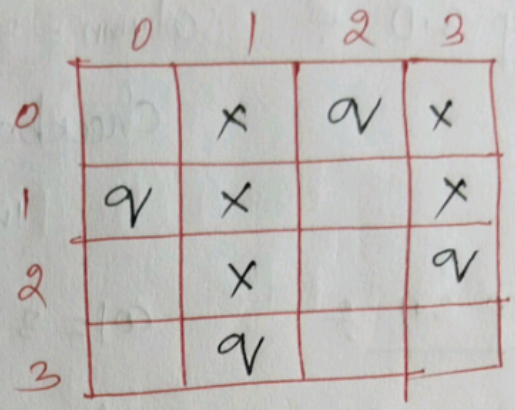
→ again backtracking

→ Remove last item placed i.e, board[3][2] X remove

→ inc row by 1

inc color row we reach last row

→ again backhack
→ remove board[0][0]
→ increment row by 1



step 0.1 col=0, row=1
check board[0][1] = safe, place queen
inc column

step 1.0 col=1, row=0
Check board[0][1] = not safe
inc row

step 1.1 col=1, row=1
Checkboard[1][1] = not safe
inc row

step 1.2 col=1, row=2
checkboard[2][1] = not safe
inc row

step 1.3 col=1, row=3
Checkboard[3][1] = safe, Place queen
inc column

step 2.0 col=2, row=0
Check board[0][2] = safe, place queen
inc column

step 3.0 :- column = 3, row = 0

checkboard [0][3] = not safe

inc row

step 3.1 :- col = 3, row = 1

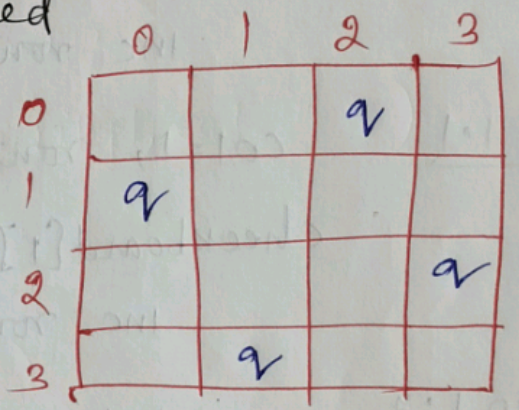
checkboard [1][3] = not safe

inc row

step 3.2 :- col = 3, row = 2

checkboard [2][3] = safe

Finally 4 queens were placed



Matrix representation :-

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

```
def is_safe(board, row, col, n):
    for c in range(col-1, -1):
        if board[row][c] == 1:
            return false

    i = row
    j = col
    while i>=0 and j>=0
        if board[i][j] == 1:
            Return false
        i -= 1
        j -= 1
    i = row
    j = col
    while i<n and j>=0
        if board[][j] == 1:
            return false
        i += 1
        j -= 1
    return True

def nQueens(board, col, n):
    if col >= n:
        return True
    for i in range(n):
        if is_safe(board, i, col, n):
            board[i][col] = 1
```

```
            if nQueens(board, col+1, n):
                return True
            board[i][col] = 0
    return False

n = int(input("Enter size of board: "))
board = [[0 for j in range(n)] for i in range(n)]
if nqueens(board, 0, n) == True:
    Print(board, 0, n
    for i in range(n):
        for j in range(n):
            Print(board[i][j], end=' ')
        Print()
else:
    Print("Not possible")
```

```
Algorithm NQueens(k, n)
{
  for i := 1 to n do
  {
    if place(k, i) then
    {
      x[k] := i;
      if (k = n) then write (x[1:n])
      else
      NQueens(k+1, n);
    }
  }
}

Algorithm place(k, i)
{
    for j := 1 to k-1 do
      if ((x[j] = i // in the same column
        or (Abs(x[j] - i) = Abs(j - k))) // or in the same diagonal
      then return false;
    return True;
}
```

Sum of subsets :-

- The problem is to find a subset of a given set $S = \{s_1, s_2, \ldots s_n\}$ of n +ve integers whose sum is equal to a given +ve integer 'd'.
- It is convinient to solve problem if elements are in increasing order i.e, $s_1 \leq s_2 \leq s_3 \leq s_4 \ldots s_n$
- Each subset need not be of fixed size.

example :-

$$S = \{1, 3, 4, 5\}$$

from this set we have to select a subset whose total is 8. and also the elements should be in increasing order.

sol 1 :- $\{1, 3, 4\}$

sol 2 :- $\{3, 5\}$

} Sum of subsets

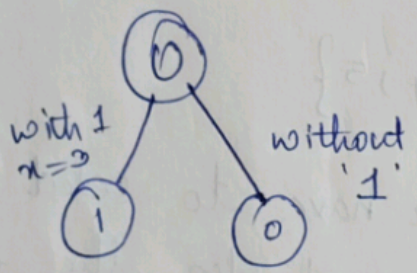
Space state tree :- It is a tree representing all the possible states (solutions or nonsolutions) of the problem from the root as an initial state to the leaf as a terminal state.

State space tree

→ Initially there are no elements in the tree, you have to start with zero.

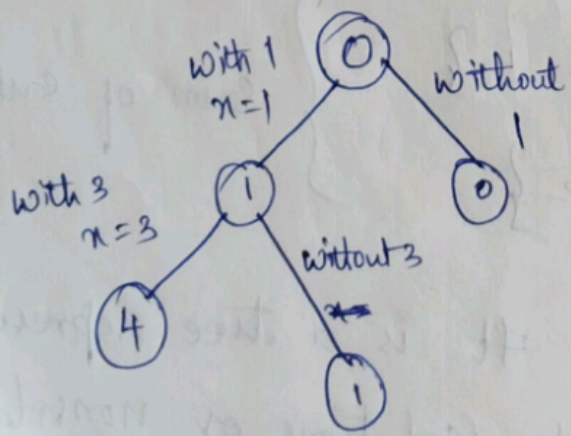
→ first element to be inserted is "1".

Subset of 0's

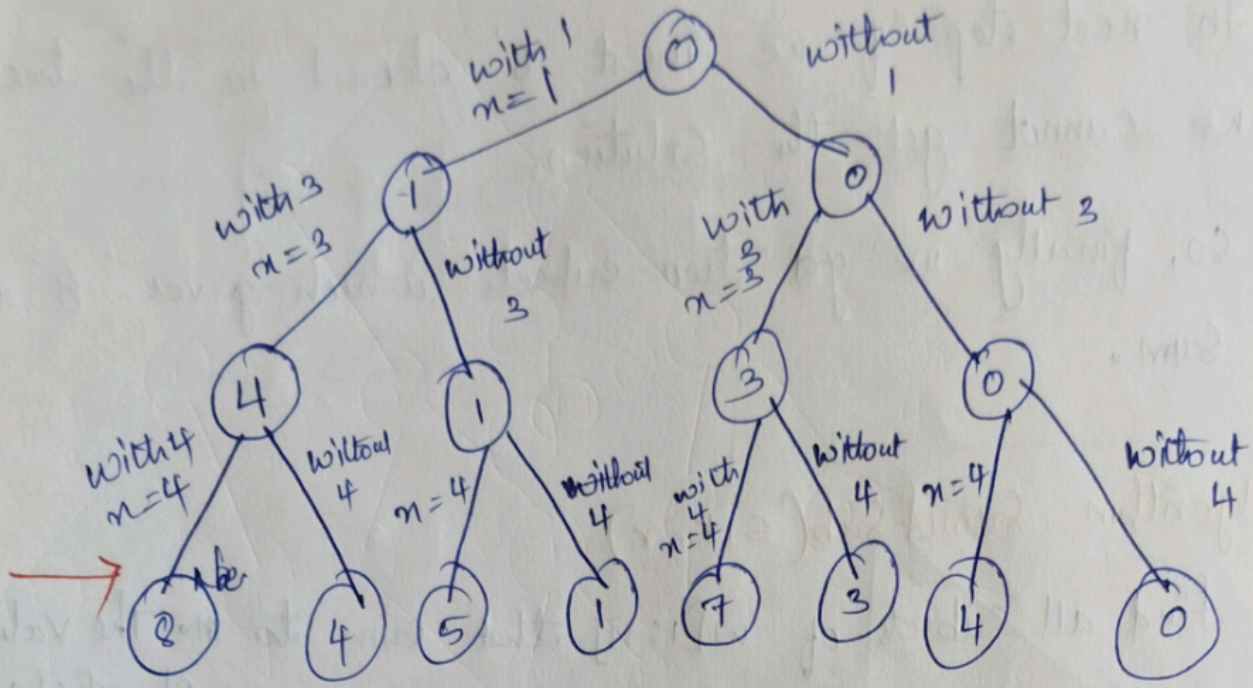


→ Next element inserted is '3', Here when with '3' we have to add node

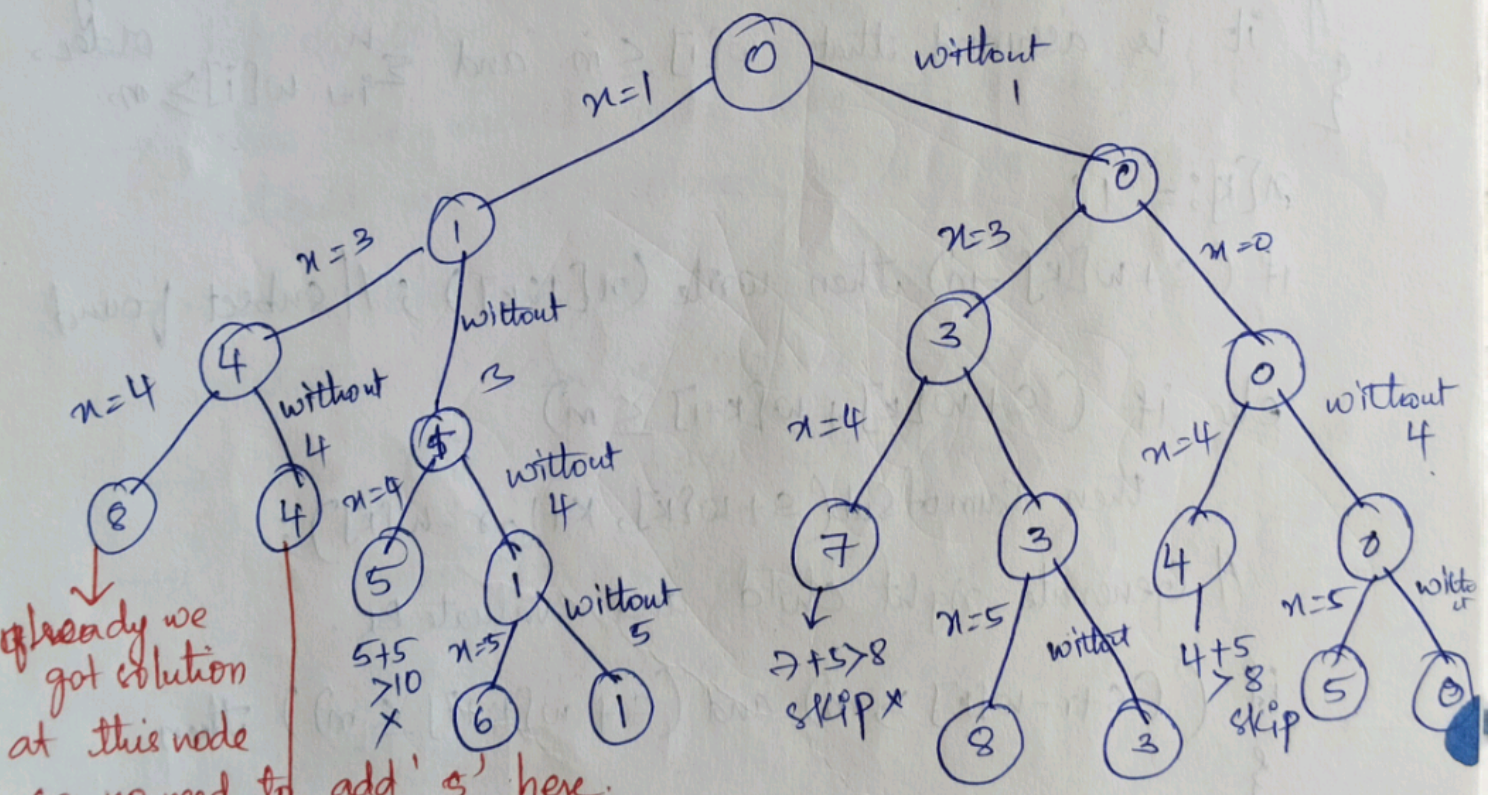
if it is without '3' only '1' need to be copied



→ Next element to be inserted is 4'.



→ Next step : while adding '5'.



already we got solution at this node so no need to add '5' here.

At this node if we add 5, 4+5>8, skip this node

→ In next step if we insert '6' element in the tree we cannot get the solution.

→ So, finally we got two subsets which gives '8' as sum.

Algorithm SumofSub(s, k, r)

// find all subsets of w[1:n] that sum to m. The values of x[j];

// 1 ≤ j < k, have already been determined, s = $\sum_{j=1}^{k-1} w[j] * x[j]$

// and r = $\sum_{j=k}^{n} w[j]$. The w[j]'s are in nondecreasing order.

// it is assumed that w[1] ≤ m and $\sum_{i=1}^{n} w[i] \geq m$.

{

x[k] := 1;

if (s + w[k] = m) then write (x[1:k]); // subset found

else if (s + w[k] + w[k+1] ≤ m)

then SumofSub(s + w[k], k+1, r - w[k]);

// Generate right child and evaluate Bk.

if ((s + r - w[k] ≥ m) and (s + w[k+1] ≤ m)) then

{

x[k] := 0;

SumofSub(s, k+1, r - w[k]);

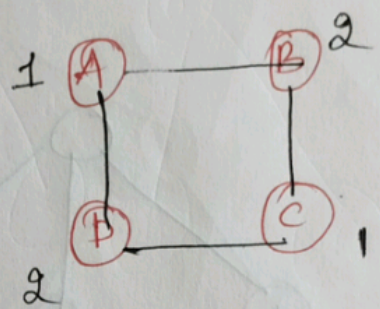
}

}

Graph Colouring :- Also called as m-way coloring problem

All the vertices of graph should be colored in such a way that no two adjacent vertices should have same colors.

ex :-



$G = (V, E)$

Let m = 3 — three colors are there

1, 2, 3 are colors.

→ let us color 'A' vertex with color '1'.

→ Now we should make sure that adjacent vertices should not be colored with same color i.e, B & D

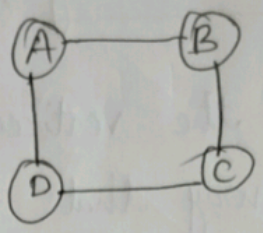
→ color 'B' with 2

→ Vertex 'c' can be colored with 1 or 3

→ Like this we have to find all possible solutions.

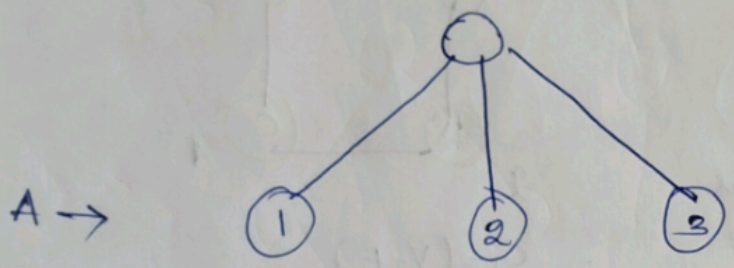
→ In order to find out the all possible solutions to graph coloring, a state space tree is constructed.

Let us construct state space tree

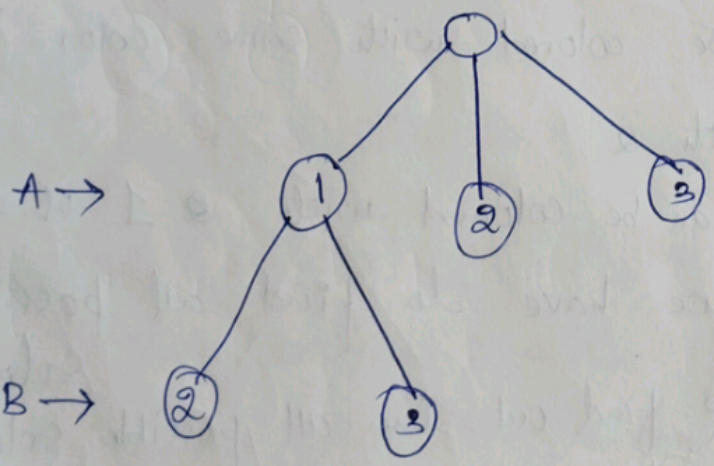


m = 3

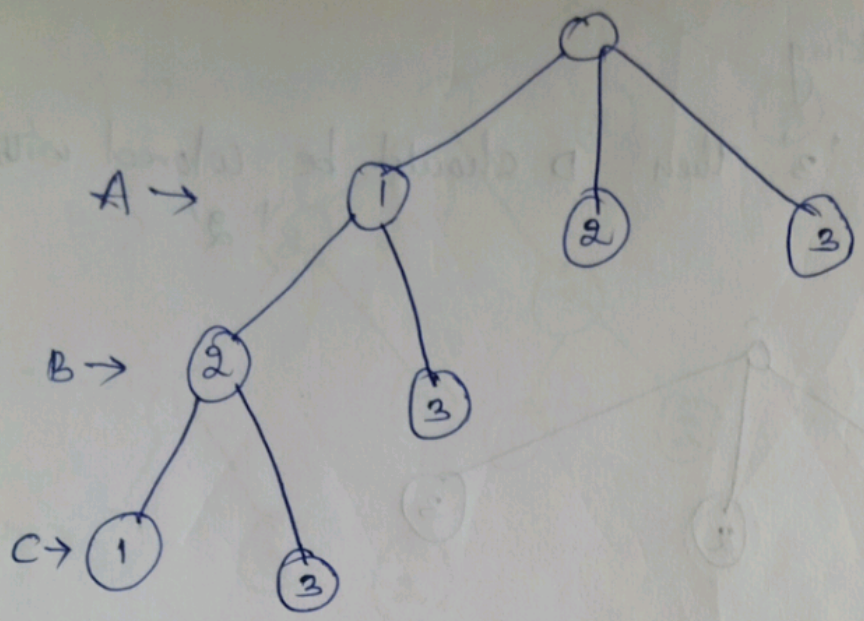
colors – 1, 2, 3



→ Let us Assume A is colored with '1'.



→ Now 'c' should be colored – 2 possibilities are there either '1' or '3' color is used

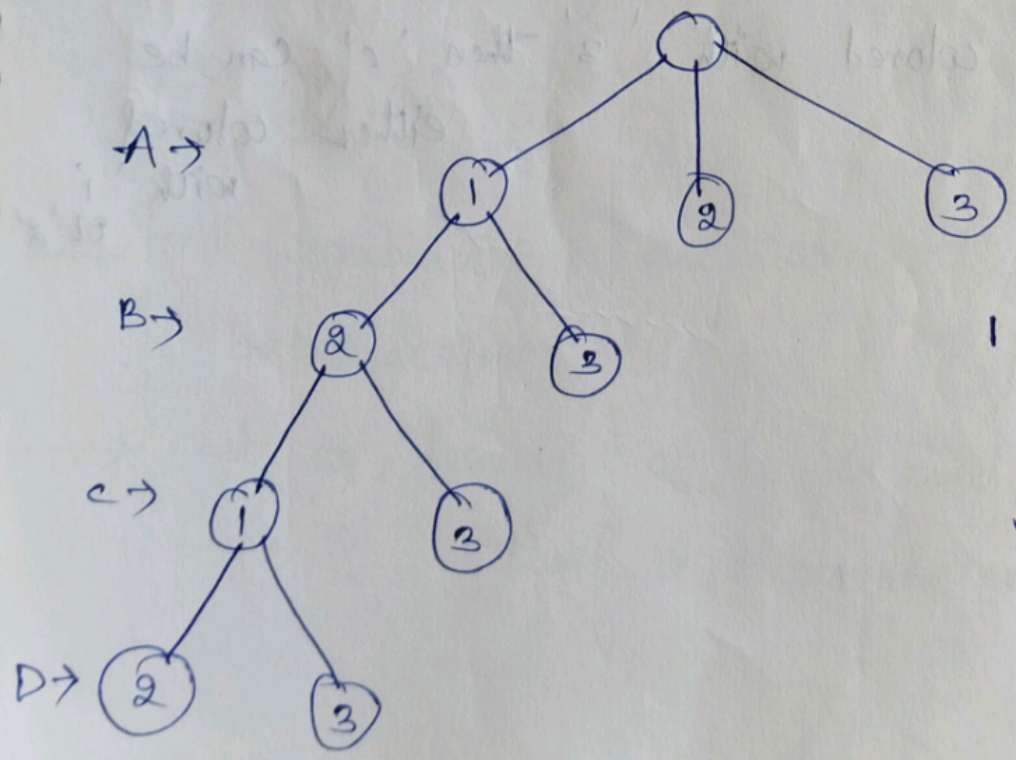


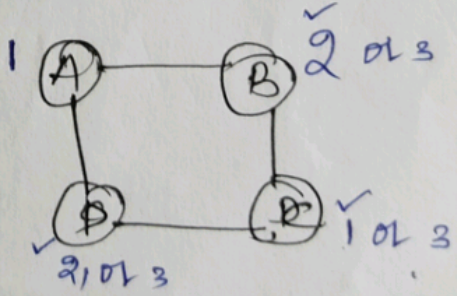
let us assume 'c' is colored with '1', adj vertices of c are D and B

'B'.

Now 'D' should be colored – adj vertices of 'D' are 'A' and 'c' → A and c are colored with '1'

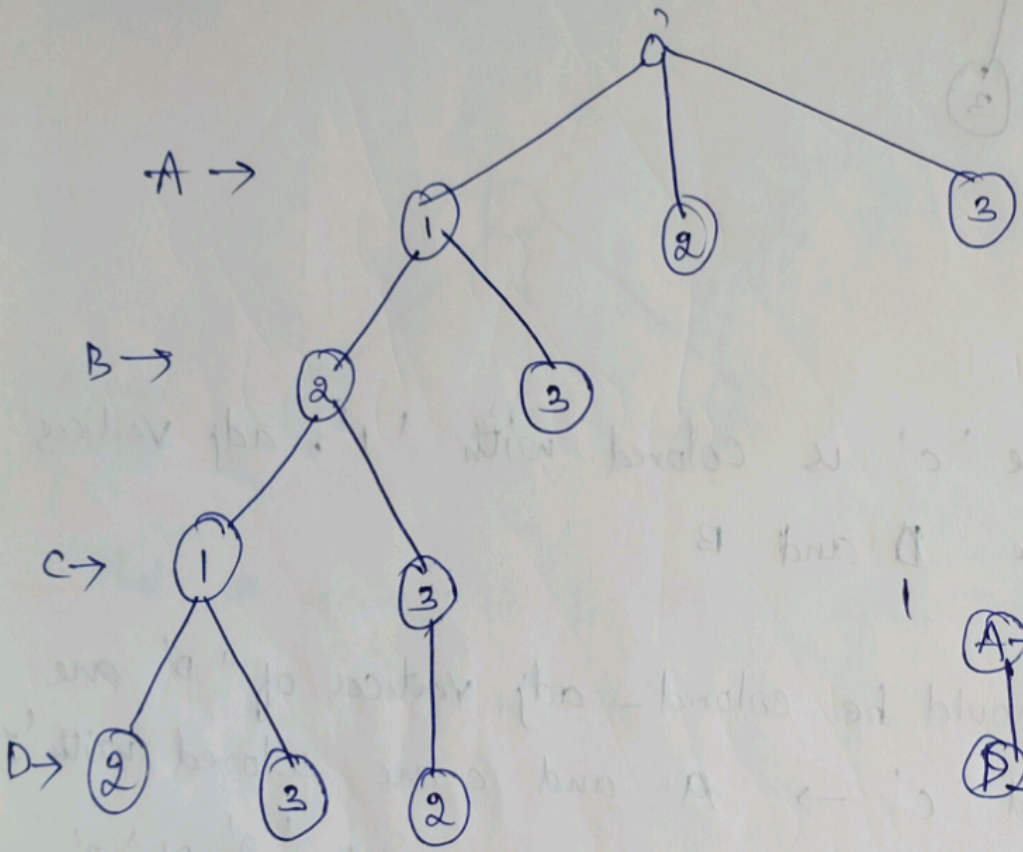
so 'D' should be colored either with '2' or '3'

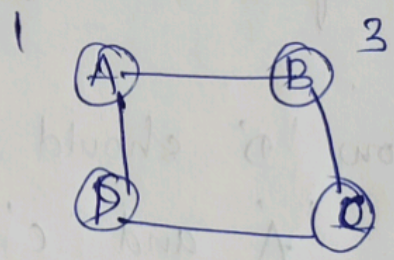




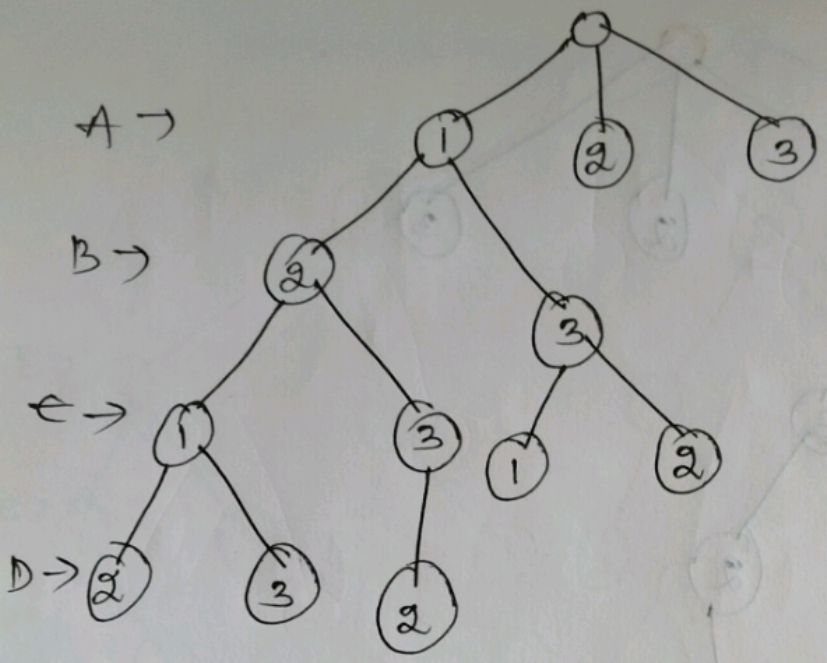
→ Now do backtracking

→ If 'c' is colored '3' then D should be colored with '2'

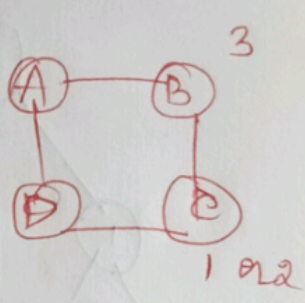




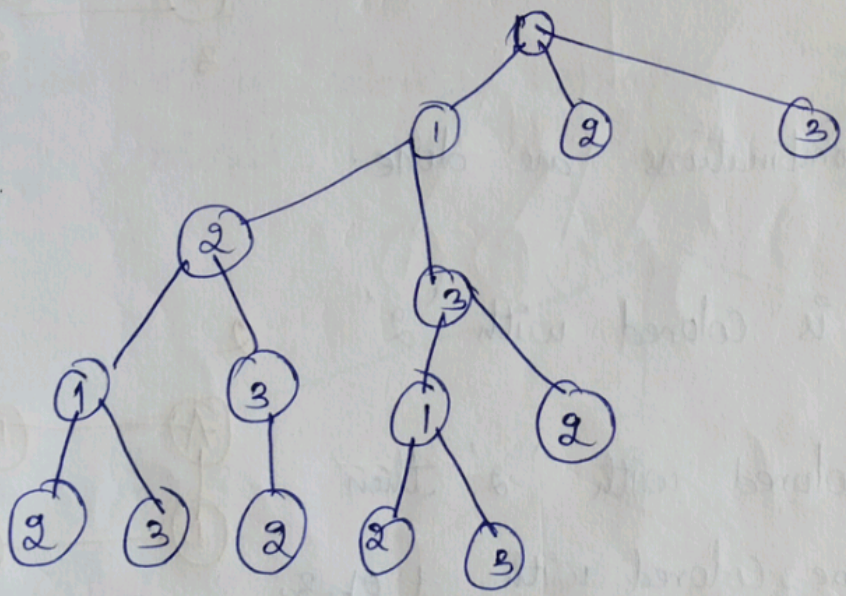
→ If 'B' is colored with '3' then 'c' can be either colored with '1' or '2'



i.e,

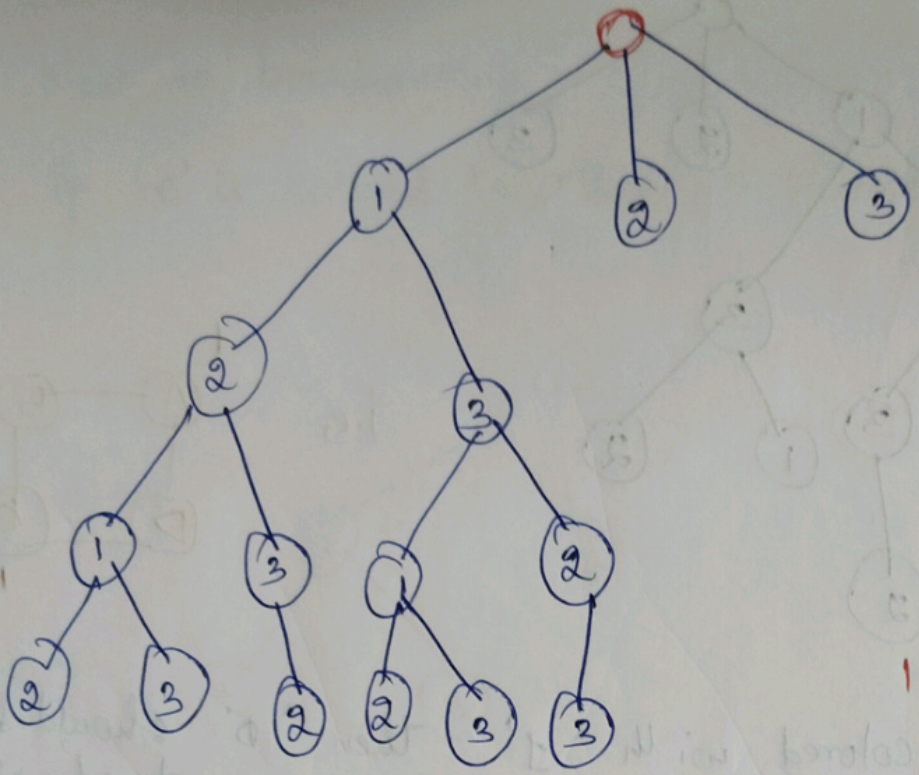


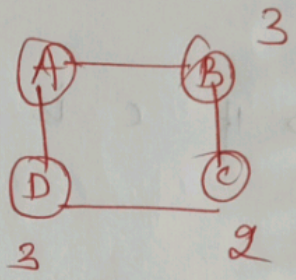
→ If c is colored with '1' then 'D' should be colored with '2' or '3'



→ All combinations are over then do the backtracking.

→ Let us assume 'c' is colored with '2' then 'D' should be colored with '3'



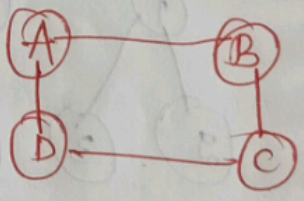


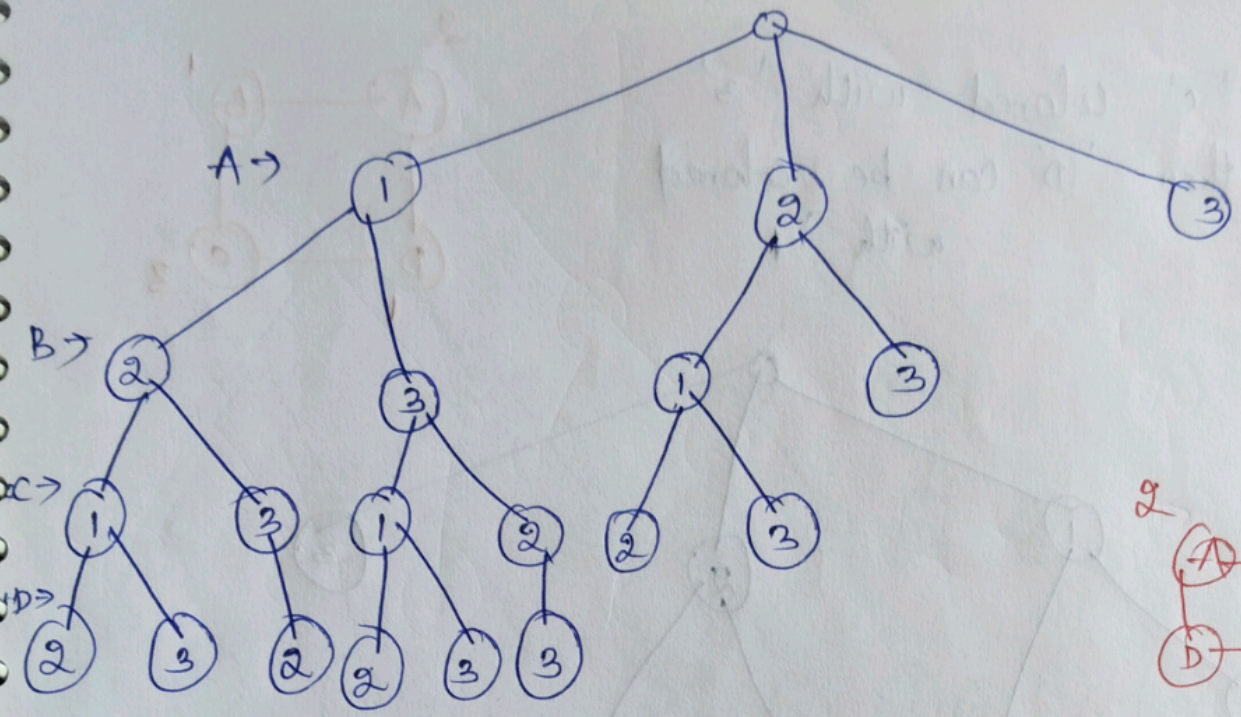
→ All possible combinations are done.

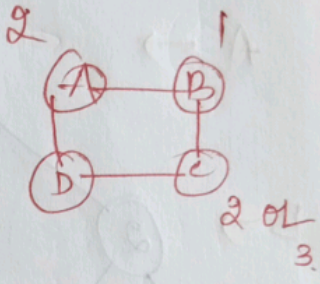
→ Backtrack

→ Assume 'A' is colored with '2'

→ If 'A' is colored with '2' then 'B' can be colored with 1 or 3.

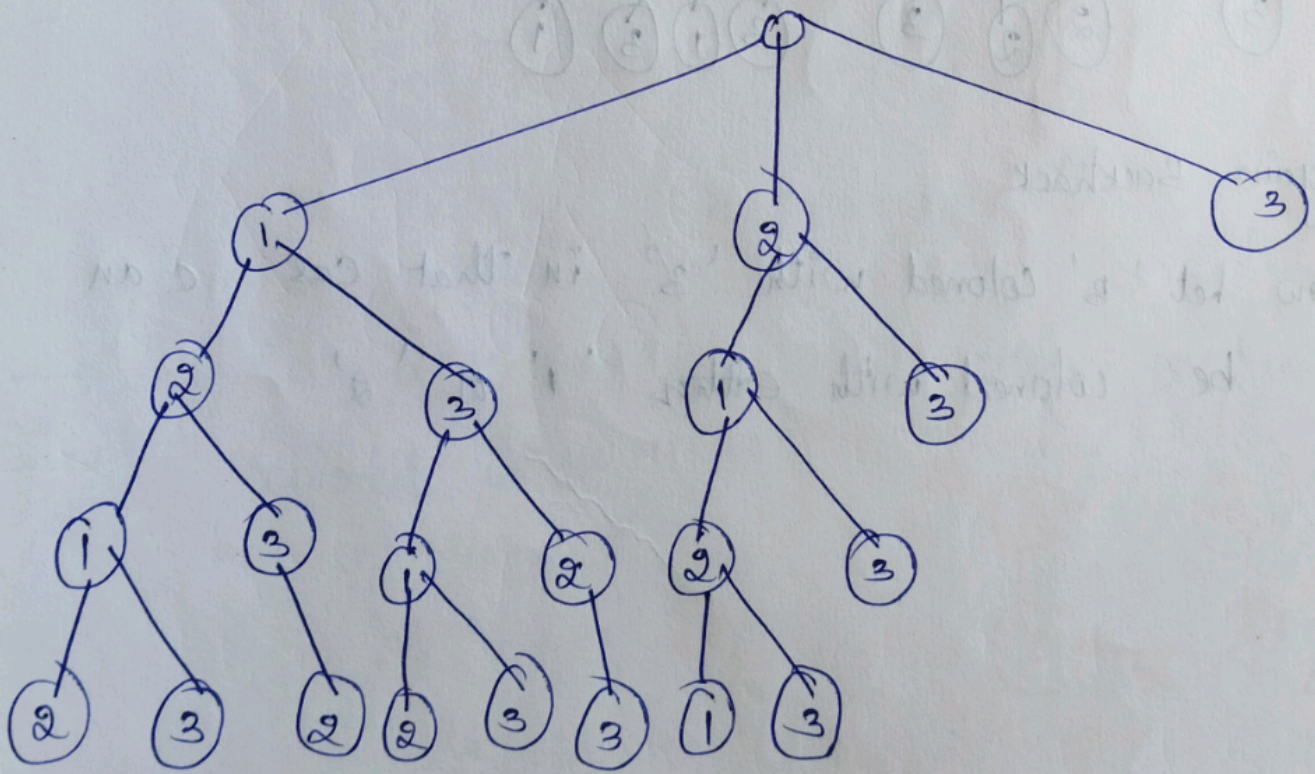






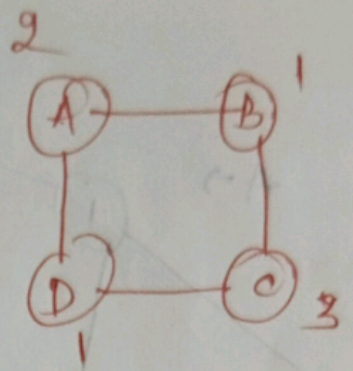
→ Assume B is colored with '1'

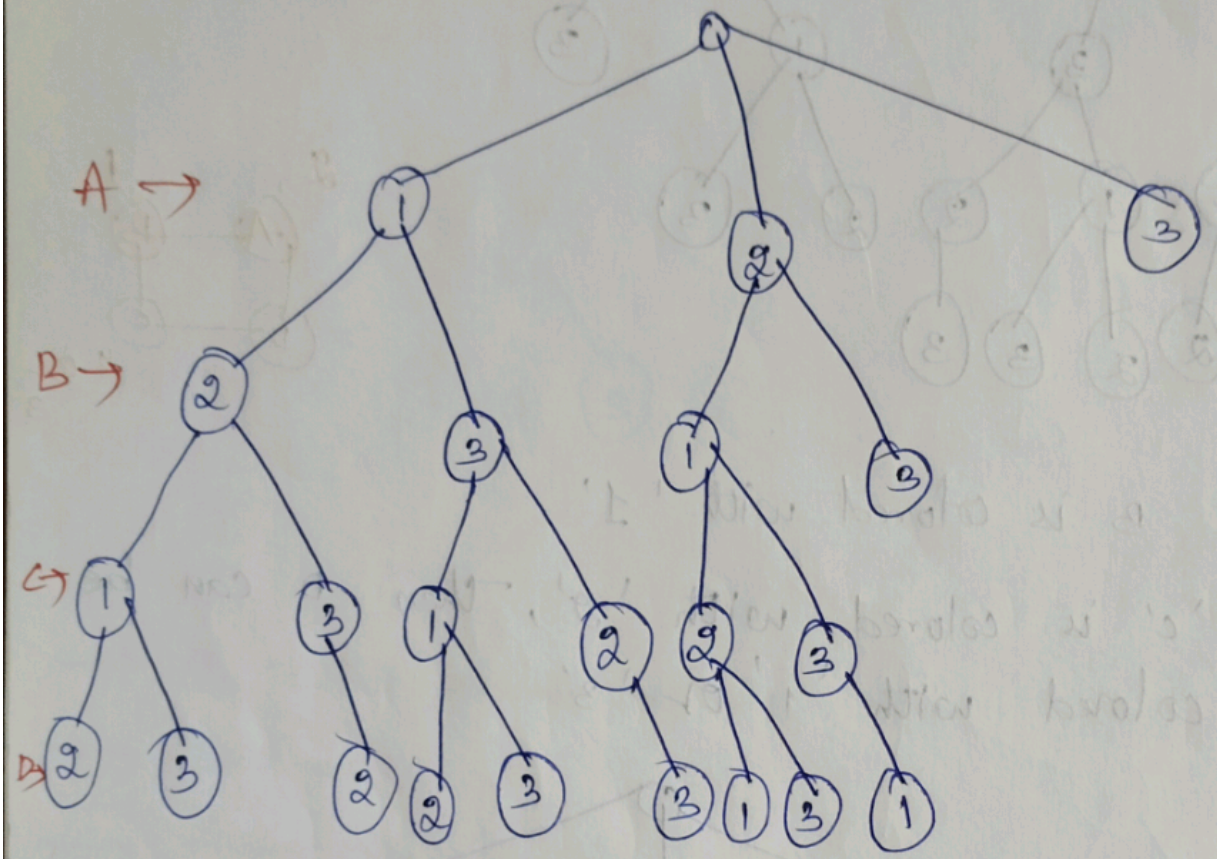
→ let 'c' is colored with '2', then D can be colored with '1' or '3'



All possible combinations are done → backtrack.

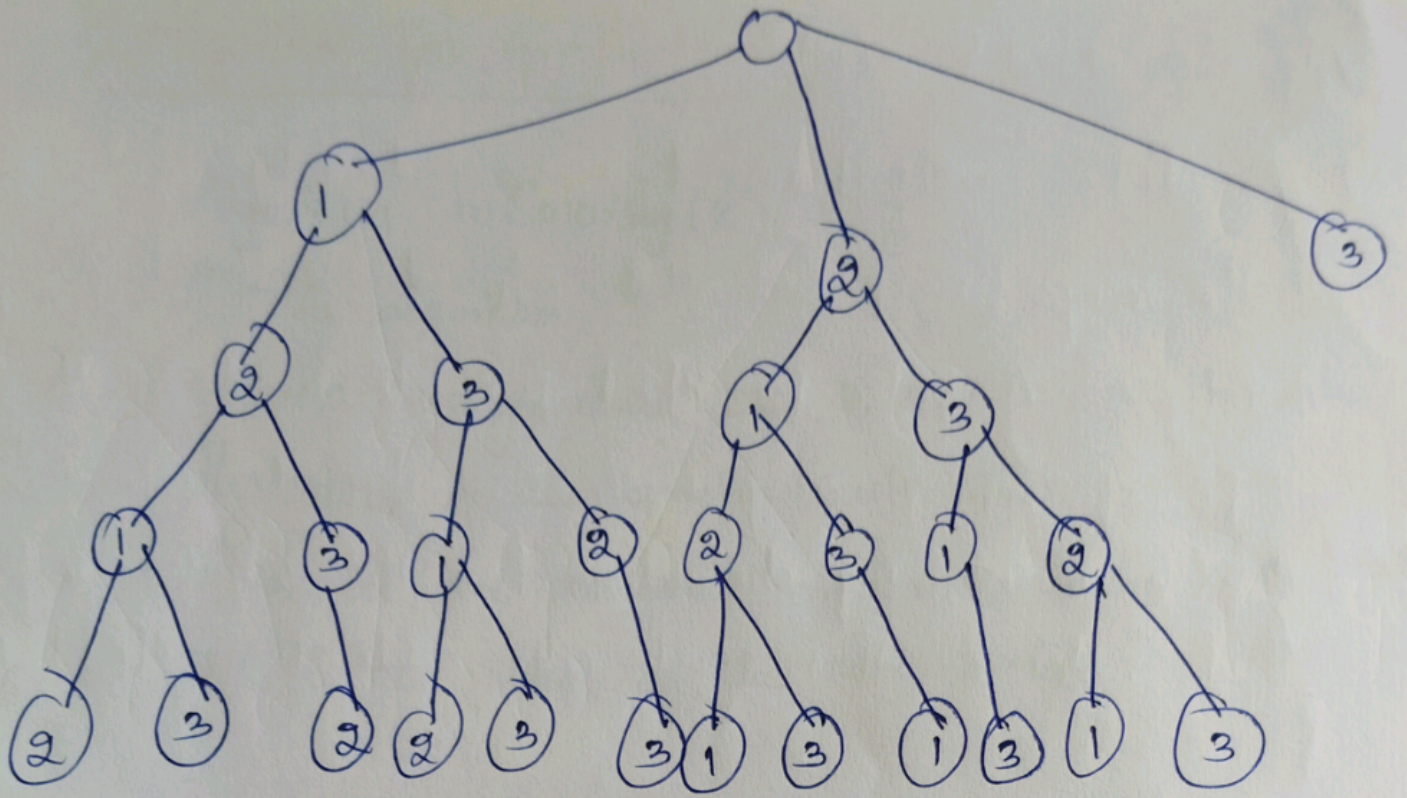
Let 'c' colored with '3'
then 'D' can be colored
with '1'



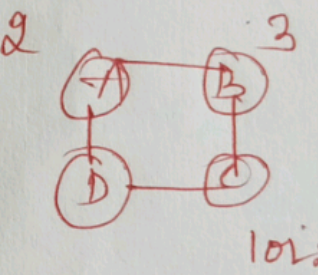


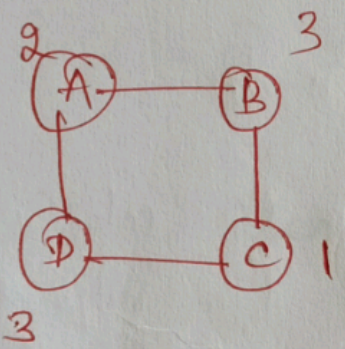
Again Backtrack

Now let 'B' colored with '3' in that case C can be colored with either '1' or '2'

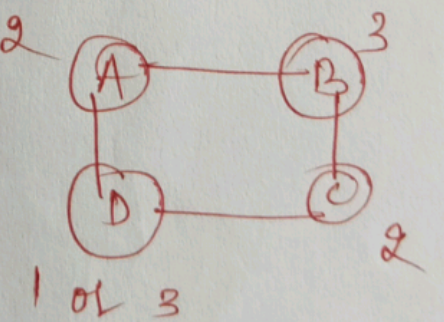


→ If 'c' is 1, then D is colored with





→ If 'c' is colored with (2) then D can be colored with



Algorithm for Graph coloring :

```
Algorithm mColoring(K)
// This algorithm
// All assignments of 1, 2, ..., m to the
// Vertices of the graph such that adjacent
// Vertices are assigned distinct integers are printed.
// K is the index of the next vertex to color.
{
  repeat
  { // Generate all legal assignments for x[K].
    NextValue(K); // Assign to x[K] a legal color.
    if (x[K] = 0) then return; // No new color possible
    if (K = n) then // At most m colors have been
                    // used to color the n vertices
      write (x[1:n]);
    else mColoring(K+1);
  } until(false);
}
```

```
Algorithm NextValue(k)
{ // x[1],....x[k-1] have been assigned integer values in
  // the range [1,m] such that adjacent vertices have distinct
  // integers. A value for x[k] is determined in the range
  // [0,m]. x[k] is assigned the next highest numbered color
  // while maintaining distinctness from the adjacent vertices
  // of vertex k. If no such color exists, then x[k] is 0.
  {
    repeat
    {
      x[k] := (x[k]+1) mod (m+1); // next highest color
      if (x[k]=0) then return; // All colors have been used
      for j := 1 to n do
      {
        // check if this color is
        // distinct from adjacent colors.
        if ((G[k,j] ≠ 0) and (x[k] = x[j]))
          // if (k,j) is and edge and if adj
          // vertices have the same color.
          then break;
      }
      if (j = n+1) then return; // New color found
    } until (false); // otherwise try to find another color.
  }
```