Disjoint sets:

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

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

These are disjoint sets, where there are no common elements.

. There are three types of disjoint sets Representations.

They are:

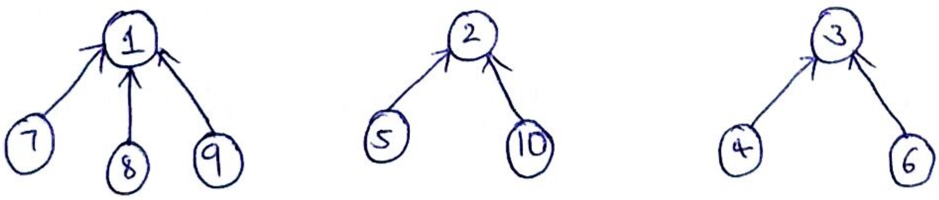
i) Tree representation:-

. A tree is used to represent each set and the root to name a set.

. Each node points upward to its parent.

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

Let, us take first element of every set take as root (or we can take other elements too).

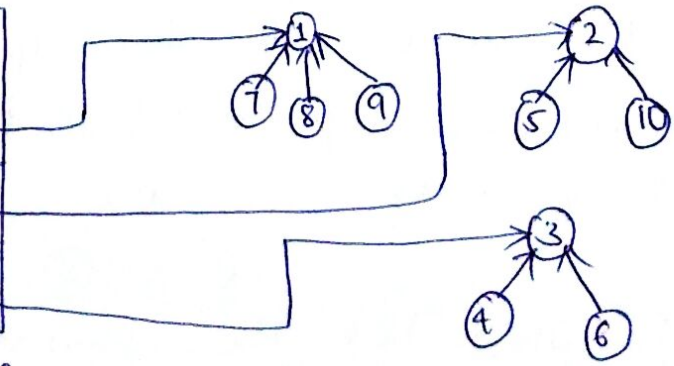


S1 S2 S3

ii) Data representation:

. Here, we are using the pointer which is used to address the each set by pointing the root node of each set to the pointer.

setname	pointer
S1	
S2	
S3	



iii) Array representation:-

. An array can be used to store parent of each element. Put the '-1' for the parent node.

Eg:- Take array of all the set as one array (1 to 10)

root root root parent of '4'

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

① Disjoint set operations:

• There are two important operations performed on disjoint sets are:

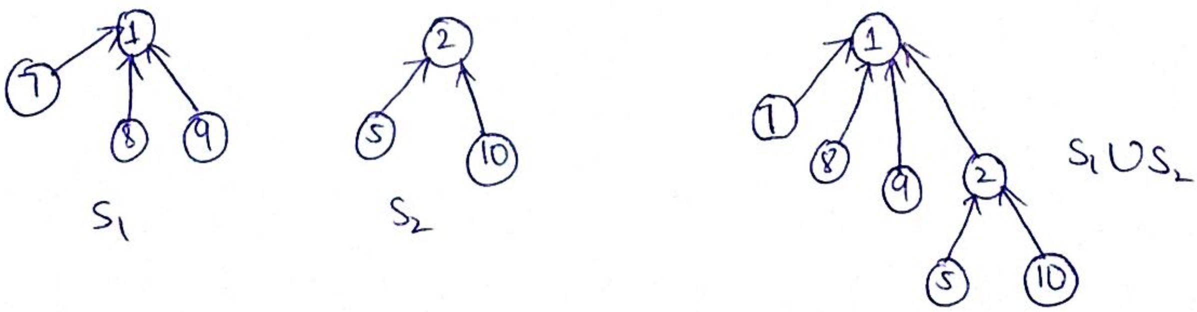
i) Union:

• If S_1 & S_2 are two disjoint sets, their unions $S_1 \cup S_2$ is a set of all elements 'x' such that 'x' is in either S_1 or S_2

• As the sets should be disjoint $S_1 \cup S_2$ replaces S_1 and S_2 which no longer exist.

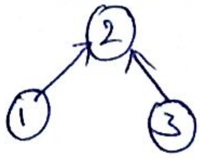
• Union is achieved by simply making of the tree as a subtree of other i.e to set parent field of one of the roots of the trees to other root.

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



ii) Find: Finding the element, with the help of its set which means the root node of respective set. Find(i).

Eg:- $S_1 = \{2, 1, 3\}$



Find(3) means in which set the element '3' is there, which means its root node is '2' i.e is in S_1.

(2) Union and find algorithms:-

Union:

- There are two types of union. They are: i) Simple Union
  ii) Weighted Union

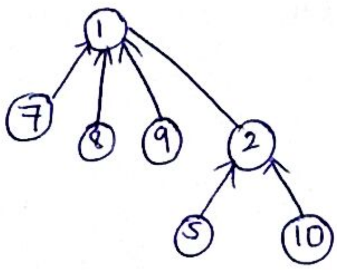
i) Simple Union:

Algorithm:

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

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

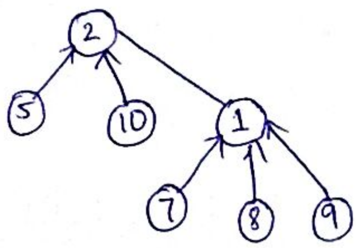
$S_1 \cup S_2$



Here, $i = 2$, $j = 1$

$P[2] = 1$ it means, parent of '2' is '1'.

$S_1 \cup S_2$



Here, $i = 1$, $j = 2$

$P[1] = 2$ it means, parent of '1' is '2'.

- Simple union leads to high time complexity in some cases.

ii) Weighted Union:

- Weighted union is a modified union algorithm with weighting rule. Widely used to analyze the time complexity of an algorithm in average case.
- Weighted union deals with making the smaller tree a subtree of the larger.
- If the no.of nodes in the tree with root 'i' is less than the no.of nodes in the tree with root j, then make 'j' the parent of 'i', otherwise make 'i' the parent of 'j'.
- Count the nodes can be placed as a negative number in the P[i]

value of the root 'i'.

Algorithm:

```
WeightedUnion(i, j)
// Union sets with roots 'i' & 'j', i≠j using the weighted rule.
// P[i]=-count[i] and P[j]=-count[j]
{
  temp: = P[i] + P[j];
  if (P[i] > P[j]) then
    { // i has fewer nodes
      P[i]:=j;
      P[j]: = temp;
    }
  else
    { // 'j' has fewer or equal nodes.
      P[j]: = i;
      P[i]: = temp;
    }
}
```

Eg: $i=1$, $j=2$, $P[1]=-4$, $P[2]=-3$

temp $= -4 -3 = -7$

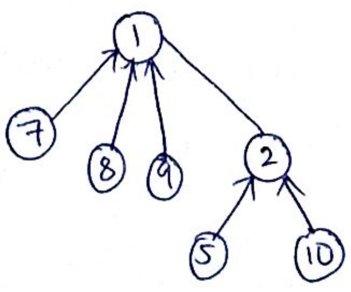
$P[1] > P[2]$ false

else

$P[2] := 1$ // $P[j] = i$, means parent of '2' is '1' means 'S2' has fewer nodes.

$P[1] := temp = -7$

$\therefore S_1 \cup S_2 \Rightarrow$



$\therefore S_2$ is subtree with fewer nodes.

Find:

- There are two types of find operations. They are: i) Simple find
  ii) Collapsing find.

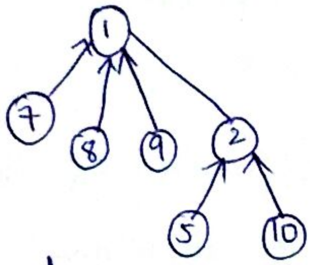
i) simple find:

```
Algorithm SimpleFind (i)
{ while(P[i] >= 0) do
```

{
i: = P[i];
}
return i;
}

. It is used to find the root noode of the given node.

Eg:



root node
∴ parent of '5' is '1'.

let i=5

P[5] = 2 ≥ 0

i: = P[5] = 2

i= 2 P[2] = 1 ≥ 0

i: = P[2] = 1

i=1 P[1] = -1 ≥ 0 false

∴ return i i.e '1'

. simple_find() leads to high time complexity in some cases.

ii) Collapsing Find:

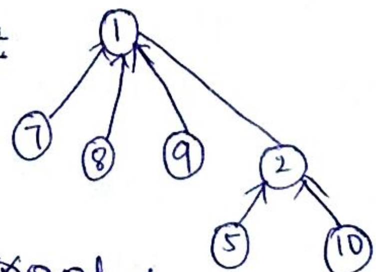
. CollapsingFind() is a modified version of SimpleFind()

. If 'j' is a node on the path from 'i' to its parent and p[i]! = root, then set 'j' to root.

Algorithm:

```
Algorithm CollapsingFind(i)
{
  r: = i;
  while (p[r] >0) do
  { r: = P[i];
  }
  while (i! = r) do
  {
    j: = P[i];
    P[i]: = r;
    i: = j;
  }
  return r;
}
```

Eg:



∴ parent of '5' is '1'
Other are collapsing in b/w them.

i=5

r: = 5

P[5] = 2 >0

r = P[5] = 2

P[2] = 1 >0

r: P[2] = 1

P[1] = -1 >0 false

i! = r ⇒ 5 ≠ 1

j = P[5] = 2

P[5] = 1

i: = 2

2 ≠ 1 ⇒ j = P[2] = 1

P[2] = 1

i: = 1

1 = 1 ⇒ return r = 1.

Backtracking:-

. The backtracking is a algorithmic-method to solve a problem with a additional way. It uses a recursive approach to explain the problems. We can say that the backtracking is needed to find all possible combination to solve an optimization problem.

① General method:-

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

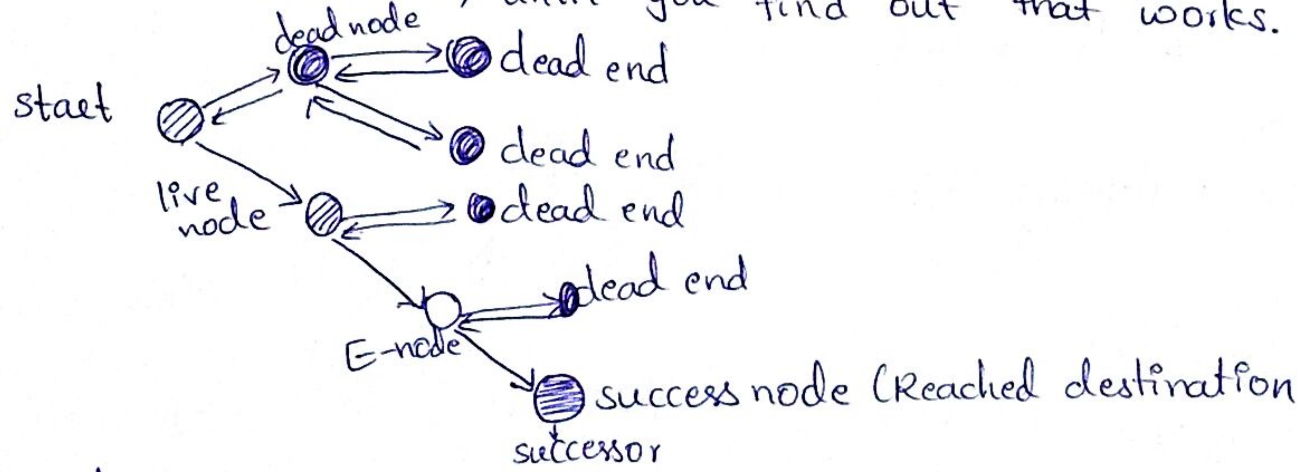
. The backtracking was used in following choices:

→ A sufficient information is not available on best choice.

→ Each decision leads to a new set of choices.

The backracking can be used as:

. Backtracking is a systematic method of trying out various sequences of decisions, until you find out that works.



Live node: A node which can be generated & all of its children have not yet being generated

Enode: Whose children is being generated & that becomes a successor

Dead node: A node which, cannot be expanded further.

→ Backtracking algorithm determines the $solu^n$ by systematically

Searching the soluⁿ space for the given problem. Backtracking is a depth-first search with any bounding function.

· All soluⁿ using backtracking is needed to satisfy a complex set of constraints. The constraints may be explicit or implicit.

→ Explicit constraint is ruled, which restrict each vector element to be chosen from the given set.

→ Implicit constraint is ruled, which determines which each of the tuples in the soluⁿ space, actually satisfy the criterion function.

② Applications:-

· There are different types of applications of backtracking.

→ N- Queen problem

→ Sum of subsets problems

→ Graph coloring and

→ Hamilton problem.

③ N- Queen's problem:-

Problem: N-Queen's problem is to place n-queens in such a manner on an nxn chessboard that no queens attack each other by being in the same row, same coloumn or same diagonal.

Algorithm:

```
Algorithm nqueen(k,n)
// this procedure prints all possible placement of nqueen on
        an n*n chessboard so that they are non-attacking.
{ for i=1 to n do
   { if place(k,i) then
      {
        x[k] = i;
        if(k==n) then write(x[1...n]);
        else
```

```
            nqueen(k+1, n);
      } } }
   }
Algorithm place(k, i)
  // return true if a queen can be placed in kth row ith column.
  // Else it return false.
  // x[] is a global array. abs(r) returns absolute value of r.
  {
    for j = 1 to k-1 do
    {
     if ((x[j] = i) // same column
       or (abs(x[j] - i) = abs(j-k)) // same diagonal
     } then return false;
    return true;
  }
```

Example: place the 4x4 queens. Let Queens be q_1, q_2, q_3 and q_4

<u>step-1</u>

$k = 1$ $n = 4$

	1	2	3	4
1	q_1			
2			$\cancel{q_2}$	q_2
3		q_3		
4				

$x[1] = 0$
$x[2] = 0$
$x[3] = 0$
$x[4] = 0$

nqueen$(\overset{1}{k}, \overset{4}{n})$

for $i = \underset{1}{1}$ to $\underset{1}{4}$

place$(\overset{1}{k}, \overset{1}{i})$

↳ for $j = \overset{1}{1}$ to $\overset{1-1}{k-1}$

$x[j] = i \Rightarrow x[1] = 1$

return true $\qquad 0 = 1$ (wrong)

$x[k] = i$

$\underline{x[1] = 1}$

nqueen$(k+1, n) \Rightarrow k = 2$

<u>step-2</u>

$k = 2$ $i = 1$ $\qquad i = \cancel{1}, \cancel{2}, \cancel{3}$

place$(2, 1)$

$j = 1$ to $2-1$

$x[1] = 1$

$1 = 1$ return false (same column)

$i = 2$ place$(2, 2)$

j = 1 to 2-1

x[1] = 2

1 = 2 (false).

abs(x[1] - 2) = abs(1 - 2)

abs(1 - 2) = abs(1-2) return false (same diagonal)

i = 3 place(2, 3) ⇒ j = 1 to 2-1

x[1] = 3

1 = 3 ✗

abs(x[1] - 3) = abs(1 - 2)

abs(1 - 3) = abs(1 - 2) ✗

return true

x[2] = 3

now, k = 2+1 = 3

<u>step-3</u>: k = 3, i = 1

place(3, 1) 3-1=2

j = 1 to k-1

x[1] = 1

1 = 1 false

i = 2 place(3, 2)

j = 1 to 3-1 ⇒ 2

x[1] = 2

1 = 2 false

i = 3

place(3, 3)

j = 1 to 3-1 ⇒ 2

x[1] = 3

1 = 3 false

i = 4 place(3, 4)

j = 1 to 3-1 ⇒ 2

x[1] = 4

1 = 4 false Backtracking to the 'q_2'

<u>step-4</u>: k = 2, i = 4

place(2, 4) ⇒ x[1] = 4

1 = 4 false

abs(x[1] - 4) = abs(1 - 2)

abs(1 - 4) = abs(1 - 2) false

return true

step-5: $k = 3$, $i = 1$

place(3, 1)

$j = 1$ to $3-1$

$x[j] = i \Rightarrow x[1] = 1$

$1 = 1$ return false

$i = 2$

place (3, 2)

$x[j] = i \Rightarrow x[1] = 2$ ✗

$1 = 2$

It can be placed

$x[k] = i$

$x[3] = 2$

$k = 3+1 = 4$

step-6: $k = 4$, $i = 1$

can't placed, Now increment $i = 1+1 = 2$

$k = 4$, $i = 2$

can't placed, it is attacking, now $i = 3$

$k = 4$, $i = 3$

also attacking

$k = 4$, $i = 4$

also attacking.

Now, backtrack to 'q_3'

step-7: $k = 3$, $i = 3$ It is attacking.

$i = 4$ It is also attacking.

Now, apply backtracking to 'q_2'

step-8: 'q_2' can't increment, so now backtrack to 'q_1'

$k = 1$, $i = 2$

place (1, 2) $j = 1$ to $1-1$

$x[1] = 2$

$1 = 2$ return true

$x[1] = 2$

now $k = 1+1 = 2$

	1	2	3	4
1		q_1		
2	X	X	X	q_2
3	q_3			
4	X	X	q_4	

step-9: $k = 2$, $i = \cancel{1}, \cancel{2}, \cancel{3}, 4$

step-10: $k = 3$, $i = 1$ no attack.

step-11: $k = 4$, $i = \cancel{1}, \cancel{2}, 3$

∴ Final solution is

$= (2, 4, 1, 3)$

mirror images of each queen is also a $solu^n$

$= (3, 1, 4, 2)$

	1	2	3	4
1		q_1	...	
2				q_2
3	q_3			
4			q_4	

$\Rightarrow$

	1	2	3	4
1			q_1	
2	q_2			
3				q_3
4		q_4		

④ Sum of subsets problem :-

- The problem is to find a subset of a given set $S = \{S_1, S_2, \ldots, S_n\}$ of 'n' positive integers whose sum is equal to a given positive integer 'd'. $S_1 \leq S_2 \leq \ldots \leq S_n$
- Each set size need not be fixed.

Algorithm:

where $s = \sum_{i=1}^{k-1} w_i x_i$

$r = \sum_{i=k}^{n} w_i$

```
sumofsubset(s, k, r)
{
    x[k]=1;
    if(s+W[k]=m) then write(X[1..k]);
    else if (s+W[k]+W[k+1]<=m)
     then sumofsubset(s+W[k], k+1, r-W[k]);
      if ((s+r-W[k]>=m) and (s+W[k+1]<=m)) then
          {
             X[k]=0;
             sumofsubset(s, k+1, r-W[k]);
          }
}
```

Example: $S = \{1, 3, 4, 5\}$ and $d (= m) = 8$ then

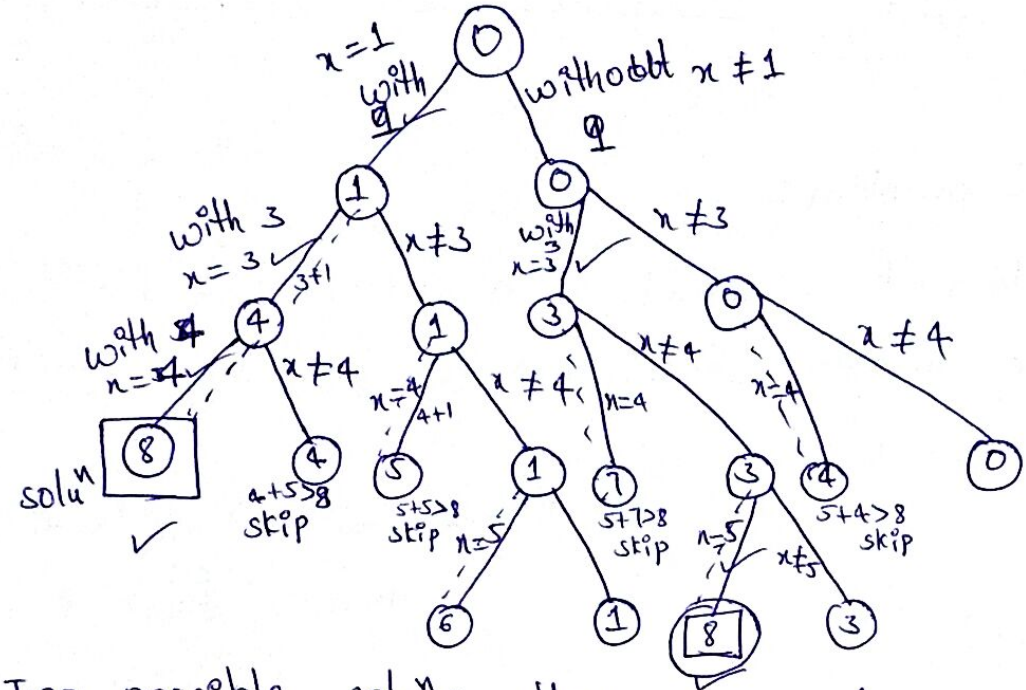
$S = \{1, 3, 4\}$ $= 1+3+4 = 8$ — $solu^n$ 1

$S = \{3, 5\}$ $= 3+5 = 8$ — $solu^n$ 2

- By using state space tree, we can find the solution for this sum of subsets problem.

· Initially start with '0' node.

· Next, take with '1' as next node (1st element of the set) and without '1' as copy (same as parent).



∴ Two possible solu^n's they are {1, 3, 4} and {3, 5}

⑤ Graph Coloring:

m-colorability decision problem:

· Let 'G' be a graph and 'm' be a given +ve integer. Determining whether the nodes of 'G' can be colored in such a way that no two adjacent nodes have the same color yet only 'm' colors are used is called M-colorability decision problem.

m-colorability optimization problem:

· m-colorability optimization problem asks the smallest integer 'm' for which the graph 'G' be colored. 'm' is called chromatic number of the graph.

m-coloring ():

· To determine all the different ways in which a given graph can be colored using at most 'm' colors.

. Suppose we represent a graph by its boolean adjacency matri

$G[1:n, 1:n]$ (row, column); The colors are represented by the integers 1, 2, 3, .., m

. The solution are given by the n-tuple (x1, ..., xn) where 'xi' is the color of node 'i'.

. Function mcoloring() is begin by first assigning the graph to its adjacency matrix, setting the array x[] to zero and involving stateme-nt mcoloring(1);

<u>Algorithm:</u>

⇒ Algorithm mcoloring(k)

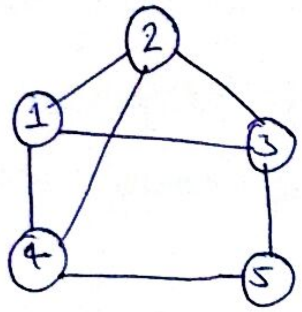
```
{ repeat
  { // Generate all legal assignment for x[k]
  nextvalue(k);
  if (x[k]=0) then return; // no color possible
  if (k = n) then // at most 'm' color have been used
    write (x[1...n]);
    else
  mcoloring(k+1);
  }
  until(false);
}
```

⇒ Algorithm nextvalue (k)

→ This distinct algorithm find the legal color for the kth vertex of graph.

```
{ 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
  { if ((G[k,j] ≠0) and (x[k] = x[j]))
       // adjacent vertices have the same color
    } then break;
  if (j = n+1) then return; // new color found
  } until (false); // otherwise try to find another color.
}
```

Eg:-



Adjacency matrix $\Rightarrow$

$$G = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1. & 0 & 1 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 & 1 & 0 \\ 3 & 1 & 1 & 0 & 0 & 1 \\ 4 & 1 & 1 & 0 & 0 & 1 \\ 5 & 0 & 0 & 1 & 1 & 0 \end{array}$$

Red blue Green

$m = 3 \Rightarrow \{R, B, G\}$
$(1, 2, 3)$

$x[1]$	$x[2]$	$x[3]$	$x[4]$	$x[5]$
$\cancel{0}$ 1	$\cancel{0}$ 1	0	0	0

initially $x[i] =$ (above), $k=1$

$\Rightarrow$ mcoloring(1) $\rightarrow$ Nextvalue[1] $\rightarrow x[k] = (x[k]+1) \bmod (m+1)$

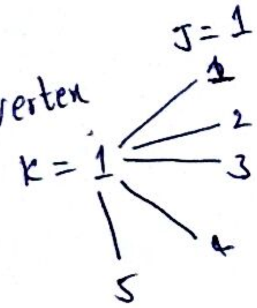
$x[1] = (0+1) \bmod (3+1)$

$= 1 \bmod 4$

$x[1] = 1$

$J = 1$ to $n = 5$

vertex



$G[1,1] \neq 0$ break

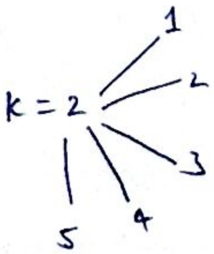
$j = n+1 = 5+1 = 6$

false No same color means we can color

mcloring$(k+1) \Rightarrow k = 1+1 = 2$

$\Rightarrow$ mcoloring(2) $\rightarrow$ NextValue[2]

$x[2] = (0+1) \bmod 4 = 1$



$G[2,1] \neq 0$ true and $x[k] = x[j] \Rightarrow x[2] = x[1]$

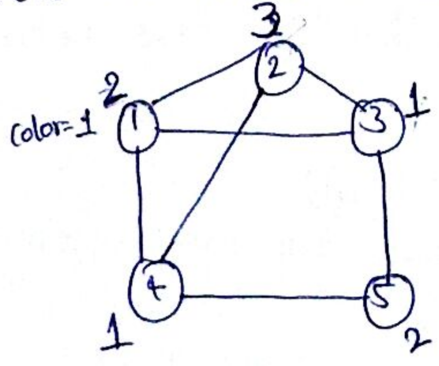
$1 = 1$

$\therefore$ It has same color over the adjacent vertices.

$G[2,2] \neq 0$ so color with '2' color to the vertex '2'.

$\therefore$ We follow this for all the vertices until all the vertices colored with given 'm' colors & with no adjacent vertices of same color.

i.e



~~chromatic number should be '4' becoz with '3' colors it is not possible to color it with no adjacent vertices with same color.~~