mood-book

UNIT-II

DISJOINT SETS

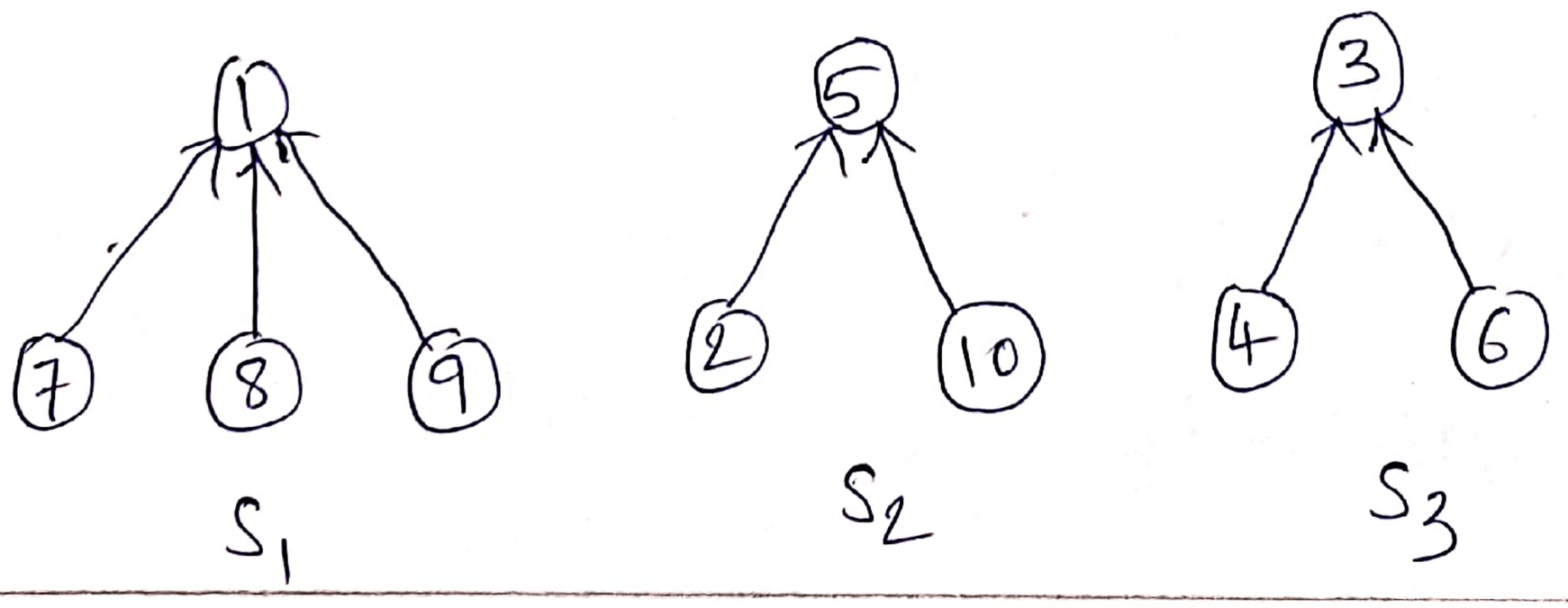
We shall assume that the elements of the sets are the numbers 1, 2, 3, ..., n. These numbers might, in practice, be indices into a symbol table in which the names of the elements are stored.

We assume that the sets being represented are pairwise disjoint (that is, if S_i and S_j, $i \neq j$ are two sets, then there is no element that is in both S_i and S_j $S_i \cap S_j = \phi$).

For example, when n=10, the elements can be partitioned into three disjoint sets

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

one possible tree representation of sets



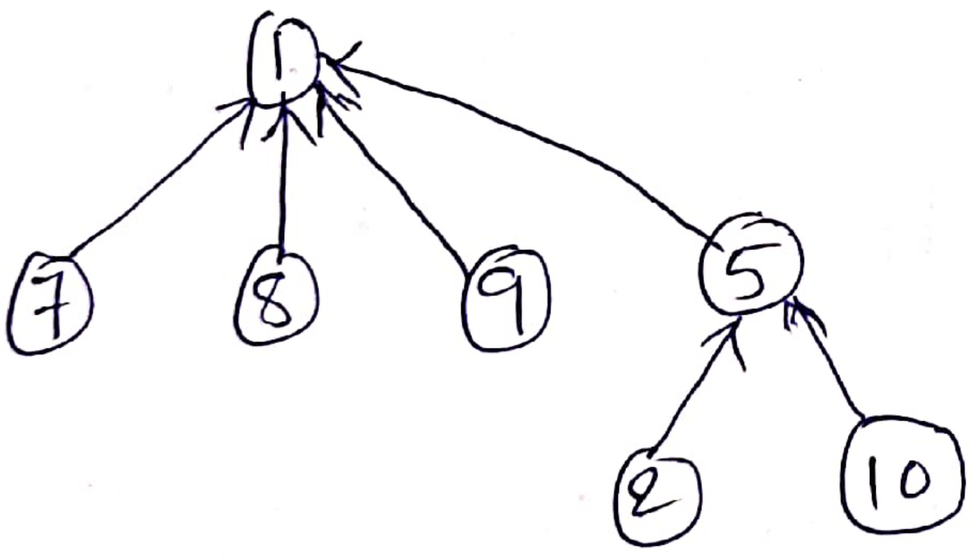
S_1 S_2 S_3

We have linked the nodes from the children to the parent.
Any element in the set can be placed at root, the remaining elements will be added as child nodes.

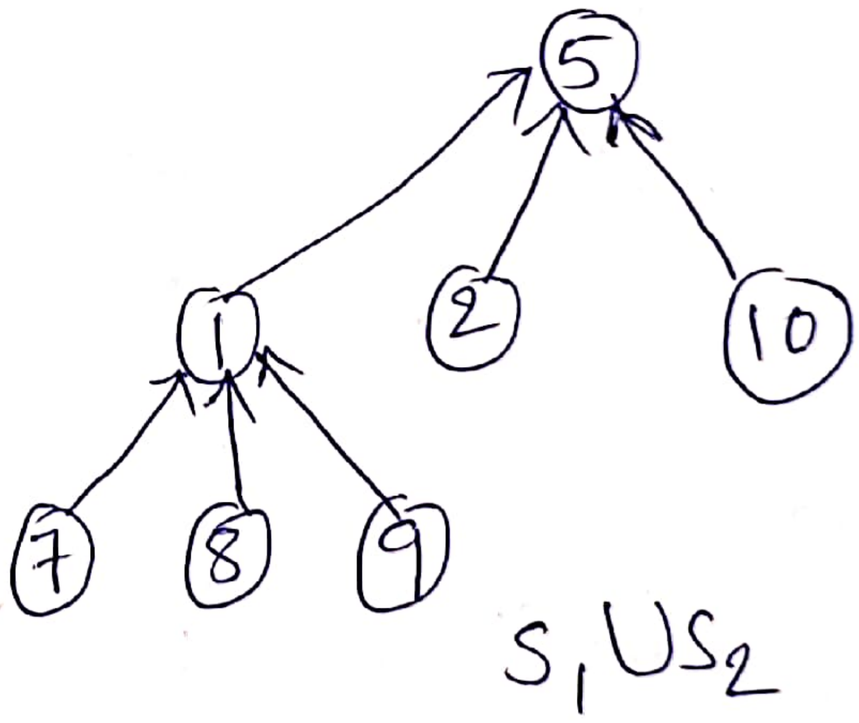
operations on the sets

1) Disjoint set Union If S_i and S_j are two disjoint sets, then their union
$S_i \cup S_j$ = all elements x such that x is in S_i or S_j.

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



or



$S_1 \cup S_2$

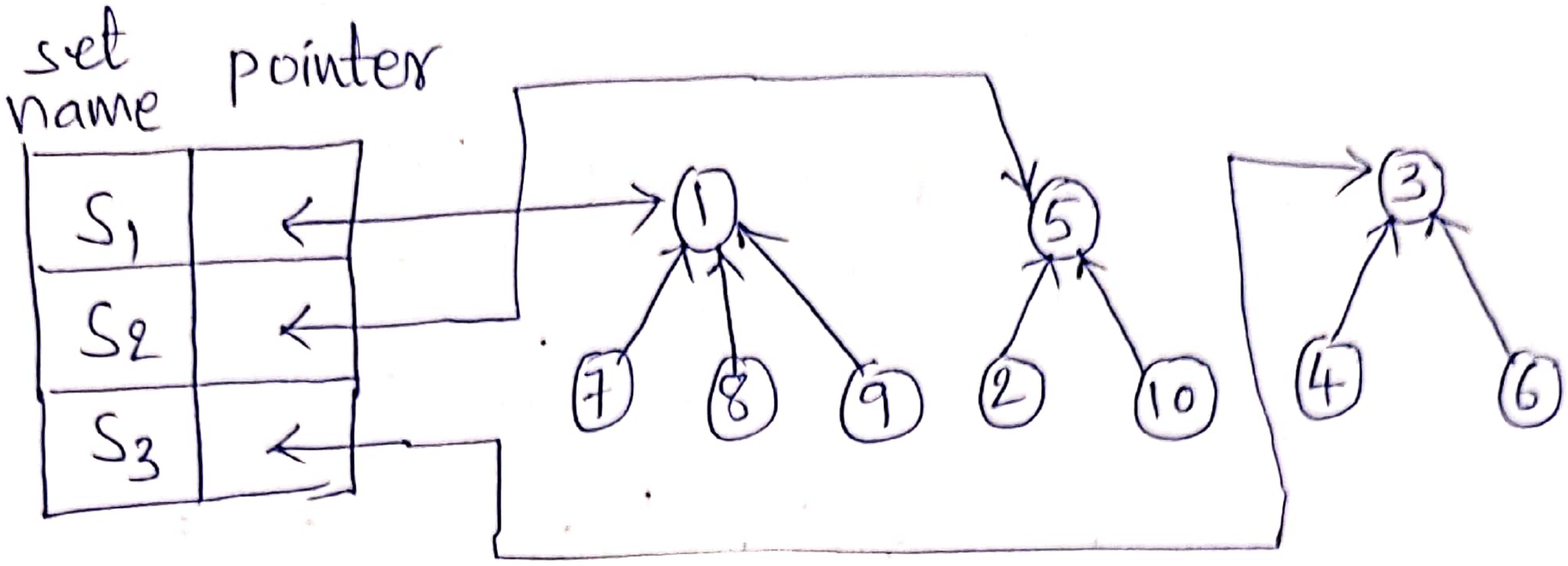
$S_1 \cup S_2$

2) Find(i) Given the element i, find the set containing i.

Thus Find(8) $\rightarrow S_1$ 8 is in set S_1

Find(4) $\rightarrow S_3$ 4 is in set S_3.

[setname, Pointer] table representation of sets



With each set name, we keep a pointer to the root of the tree representing that set. In addition, each root has a pointer to the set name. Then to determine which set an element is currently in, we follow parent links to the root of its tree. and use the pointer to the set name.

We ignore the set names and identify sets just by the roots of the trees representing them.

To find $S_i \cup S_j$, unite the trees with roots FindPointer(S_i) and FindPointer(S_j).

FindPointer() is a function that takes a set name and determines the root of the tree that represents it (that set).

Array representation of sets

P[1:n], n is the max no. of elements.

P[i] → parent of i, it represents the tree node which is parent of i

Root nodes have a parent of -1.

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

→ Algorithm SimpleUnion(i,j)

{

P[i] := j; O(1) One step.

}

i - root of first tree

j - root of second tree

First tree becomes a subtree of second.

→ Algorithm SimpleFind(i)

{

while(P[i] >= 0)

do

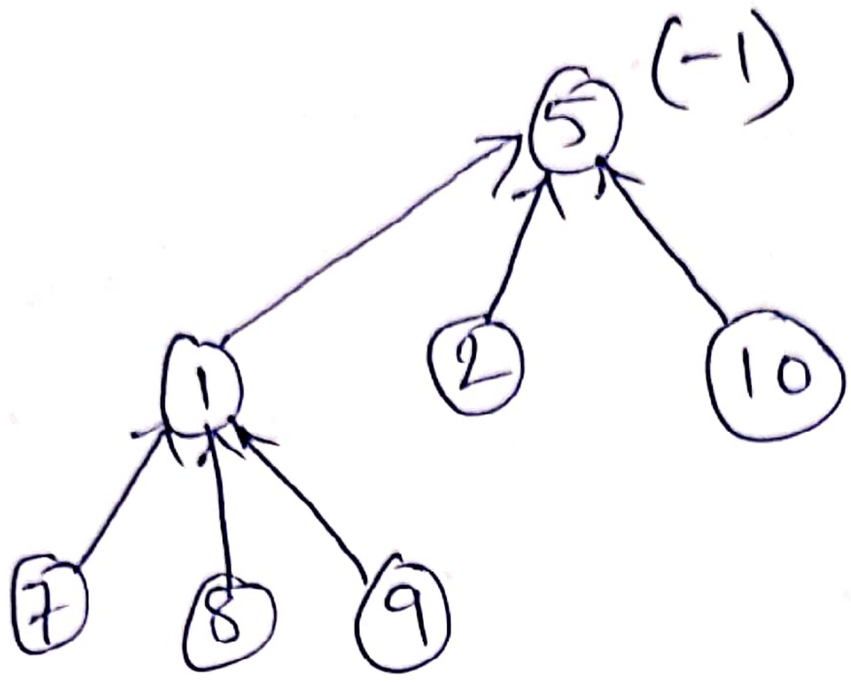
{ i := P[r];

}

return i;

}

start at node i, follow the parent pointers until we reach a node (root) with parent value -1.



Find(8)

i	P[i]
8	1
1	5
5	-1

∴ Find(8) → 5

→ Although these two algorithms are very easy to state, their performance characteristics are not very good.

Example Given $S_i = \{i\}$, $1 \le i \le n$

$S_1 = \{1\}$, $S_2 = \{2\}$, ... $S_n = \{n\}$.

Then the initial configuration consists of a forest (collection of trees) with n nodes and $P[i] = 0$, $1 \le i \le n$

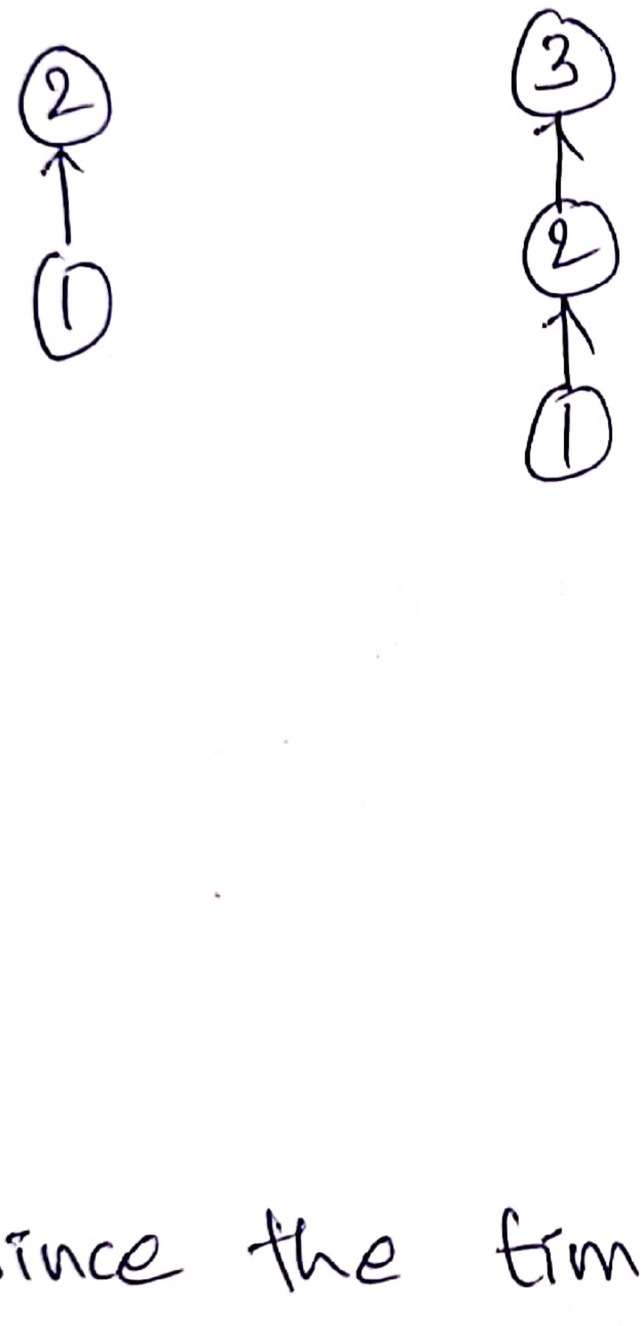
Now let us process the following sequence of union-find operations.

union(1,2), union(2,3), union(3,4), ... union(n-1,n).

Find(1), Find(2), ... Find(n-1), Find(n).

→ The sequence results in the following degenerate tree (a tree in which each parent has only one child node).

Union(1,2) union(2,3) . . . union(n-1,n)



→ Since the time taken for a union is constant O(1), the n-1 union operations can be processed in time O(n-1) = O(n).

→ Each find operation requires following a sequence of parent pointers from the element to be found to the root.

The time required to process a find for an element at level k of a tree is O(k). [we need to follow k pointer links upward to reach root].

The total time needed to process the n find operations is

Find(1), Find(2), Find(3), ... Find(n-1), Find(n).

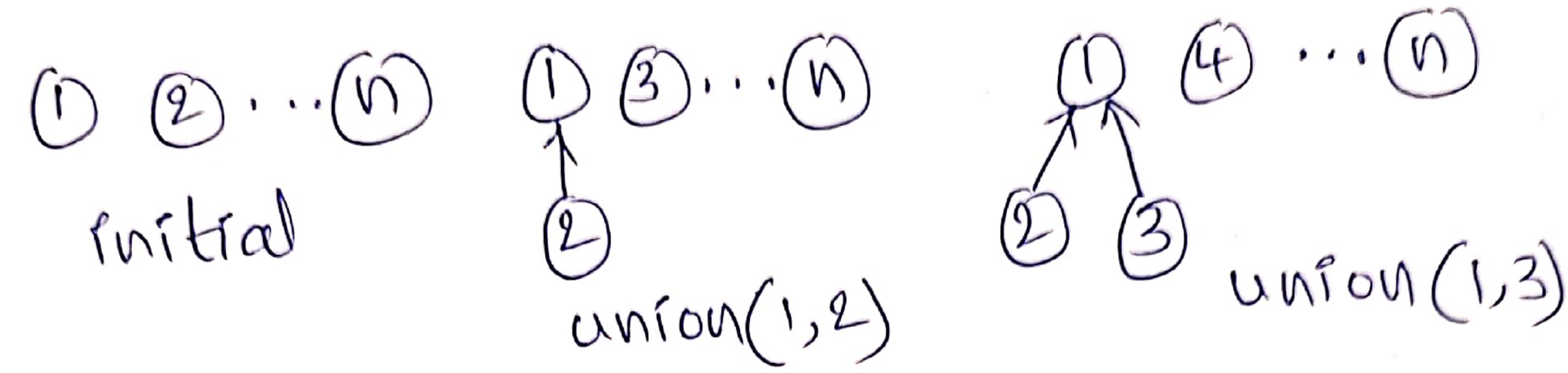
$$n + (n-1) + (n-2) + \cdots + 2 + 1.$$

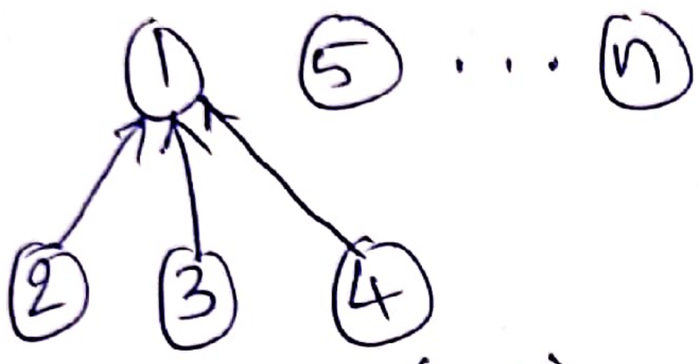
$$= O\left(\sum_{k=1}^{n} k\right) = O\left(\frac{n(n+1)}{2}\right) = O\left(\frac{n^2}{2} + \frac{n}{2}\right) = O(n).$$

→ We can improve the performance of our union and find algorithms by avoiding the creation of degenerate trees. To accomplish this, we make use of a weighting rule for Union(i,j).

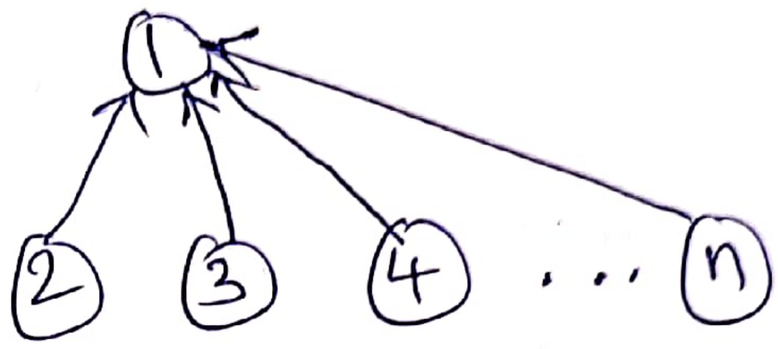
Definition [weighting rule for Union(i,j)]: If the number of nodes in the tree with root i is less than the number in the tree with root j, then make j the parent of i; otherwise make i the parent of j.

For the sequence of set unions given before we obtain the following non-degenerate trees





...



To know how many nodes there are in every tree, we maintain a count field in the root of every tree. If i is a root node, then count[i] equals the number of nodes in that tree.

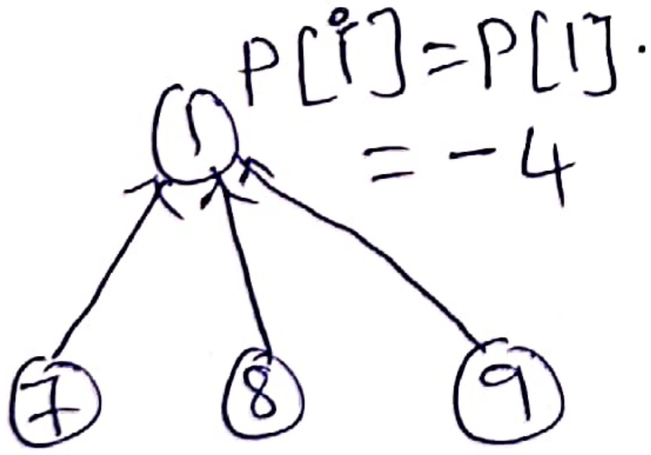
Since all nodes other than the roots of trees have a positive number in the p (parent) field, we can maintain the count in the p field of the roots as a negative number.

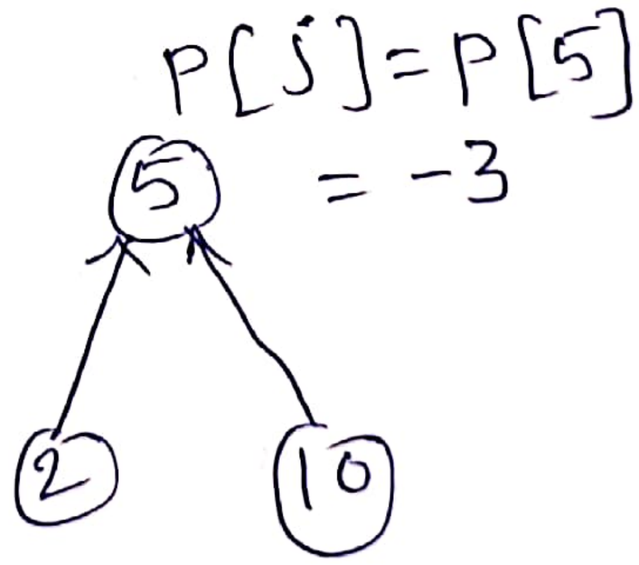
Using this convention, we obtain the following algorithm.

→ Algorithm WeightedUnion(i, j)
// Union the sets with roots i and j, i ≠ j
// using the weighting 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;  // j becomes root of resulting
      P[j] := temp;  // tree
   }
 else // if P[i] ≤ P[j]
 {  // j has fewer or equal nodes
    P[j] := i;  // i becomes root of resulting
                   tree.
    P[i] := temp;
 }
}
```

Ex

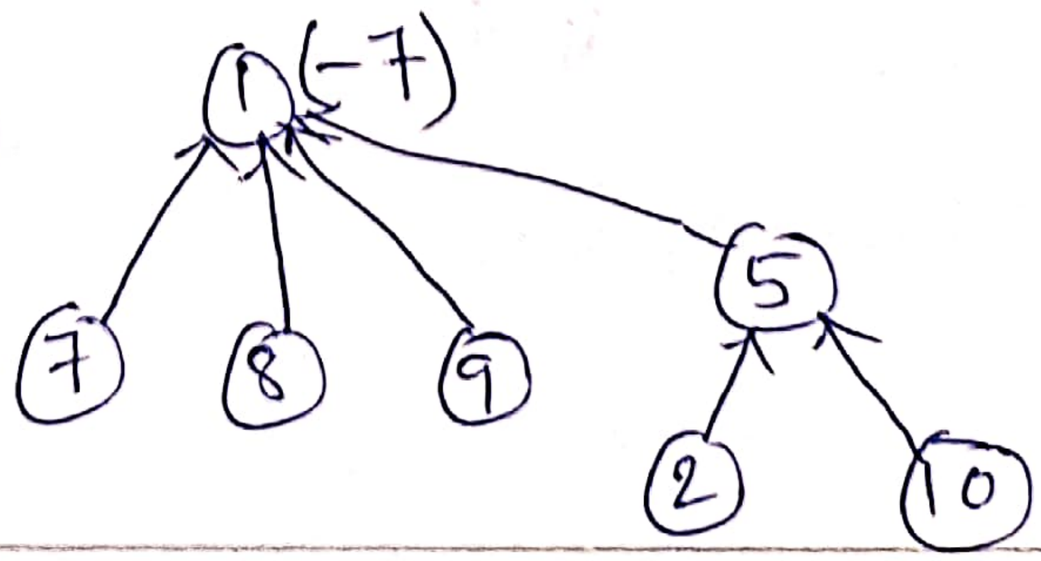




temp = -4 + (-3)
= -7.

P[1] ≤ P[5] 1 becomes root.

WeightedUnion(1,5)



Ex consider the behavior of weighted Union on the following sequence of unions starting from the initial configuration

P[i] = -count[i] = -1 , $1 \le i \le 8 = n$

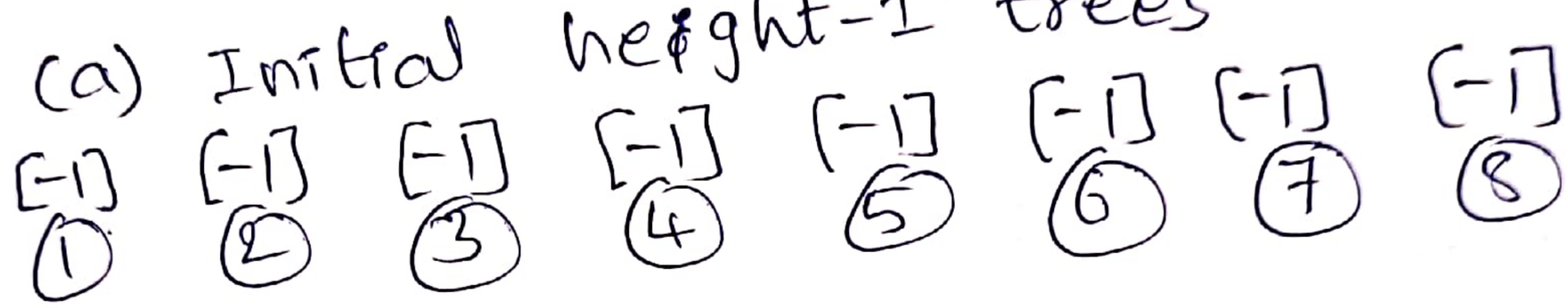
Union(1,2), Union(3,4), Union(5,6), Union(7,8)

Union(1,3), Union(5,7)

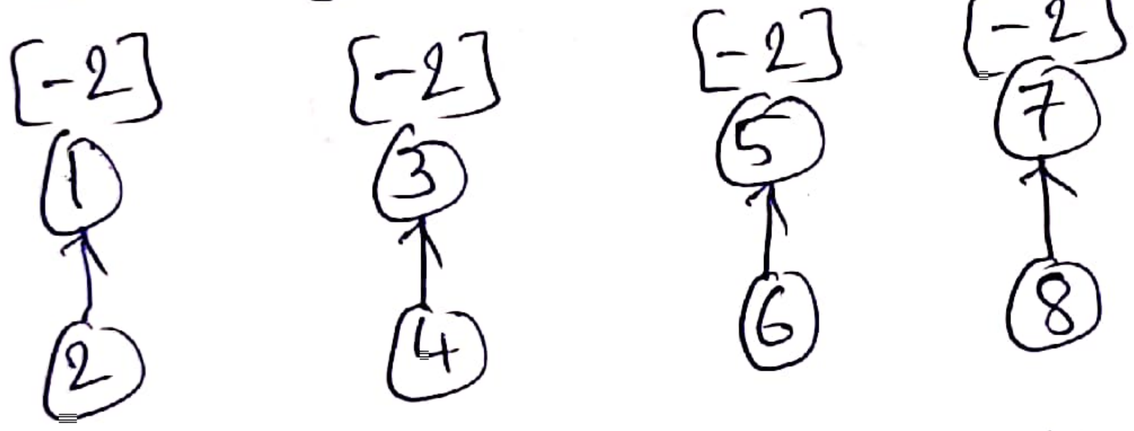
Union(1,5)

Following trees will be obtained.

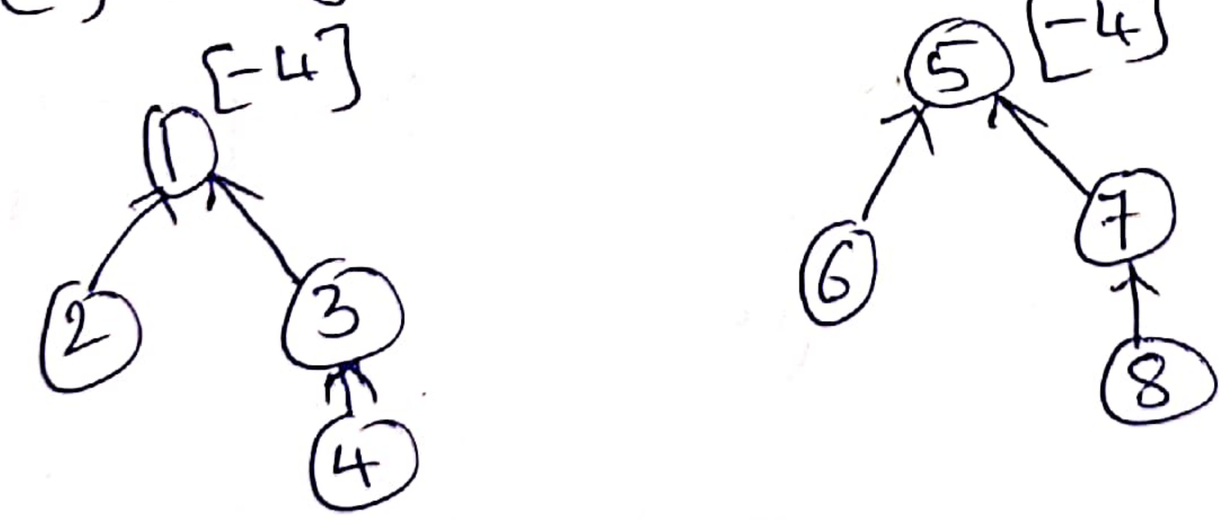
→ (a) Initial height-1 trees



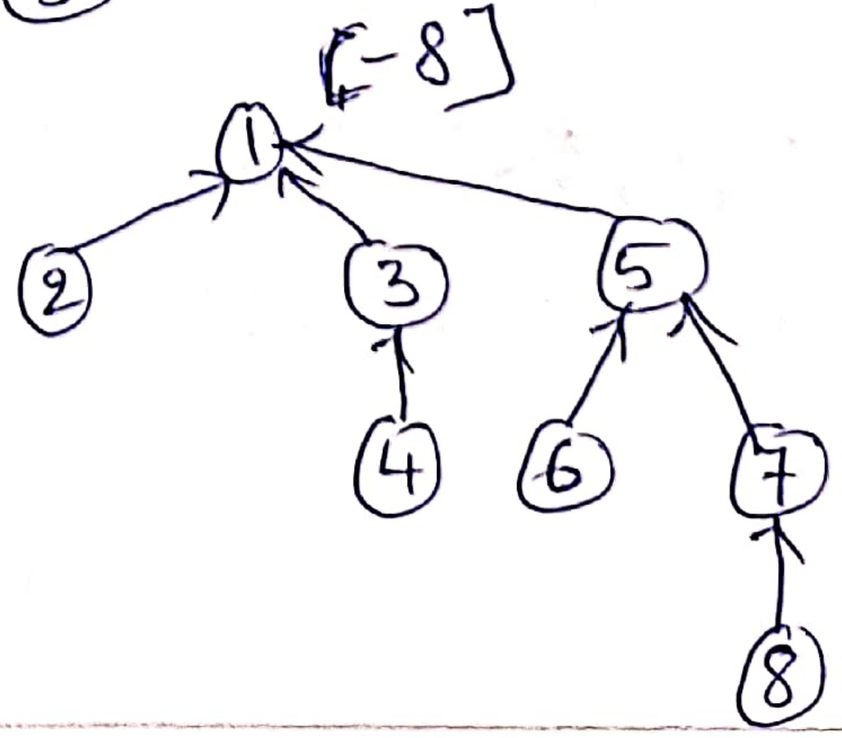
→ (b) Height-2 trees following Union(1,2), Union(3,4) Union(5,6), Union(7,8)



→ (c) Height-3 trees following Union(1,3), Union(5,7)



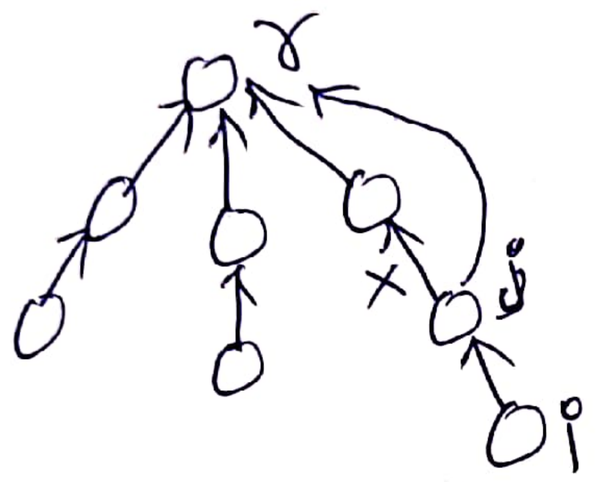
→ (d) Height-4 tree following Union(1,5)



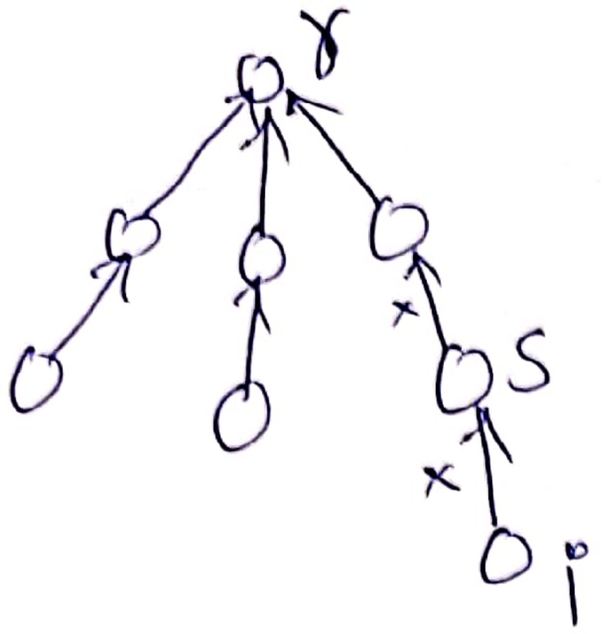
Collapsing Find

Further improvement is possible. This time the modification is made in the find algorithm by using collapsing rule.

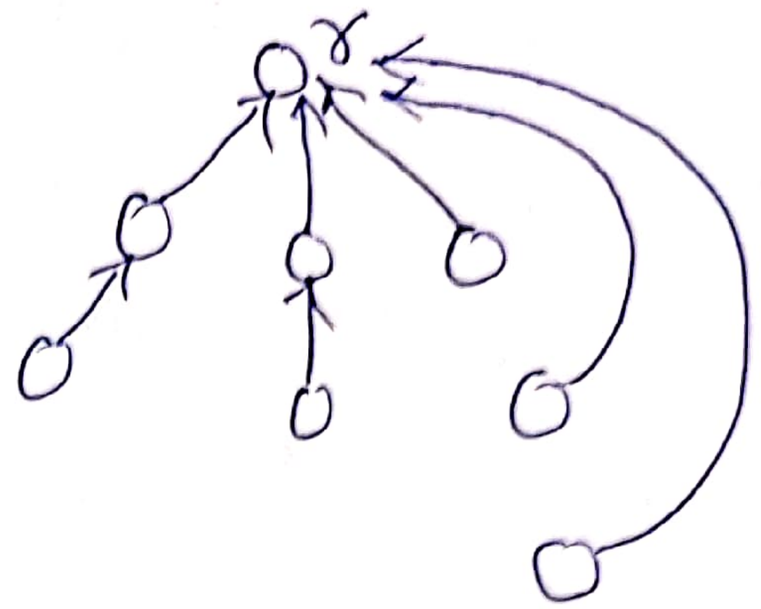
Definition [Collapsing rule] If j is a node on the path from i to its root and p[i] ≠ root[i], then set p[j] to root[i].




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



After
→
applying
CollapsingFind
rule



BACKTRACKING

In Backtracking we will use bounding function (criterion function), implicit and explicit constraints (conditions or rules).
The major advantage of backtracking method is, if a partial solution $(x_1, x_2, x_3, \cdots, x_i)$ can't lead to an optimal solution then $(x_{i+1}, \cdots x_n)$ solution may be ignored entirely.

At a particular step if we can't find a solution, we will backtrack (going back) to the previous step, undo (cancel) the solution found in previous step, and check whether there is any other alterna te choice that we have not tried but that might give a solution.

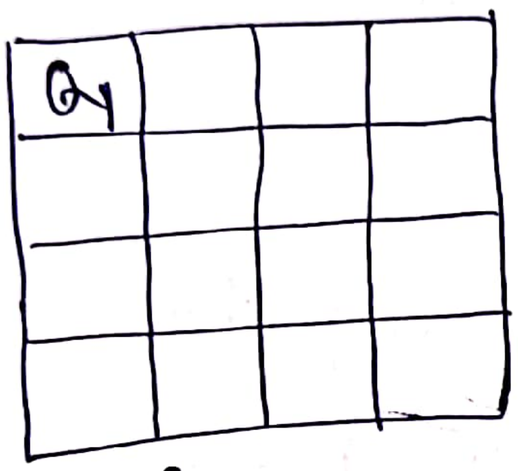
4-Queens problem

Consider a 4x4 chessboard. Let there are 4 queens. The objective is to place these 4 queens on 4x4 chessboard in such a way that no two queens should

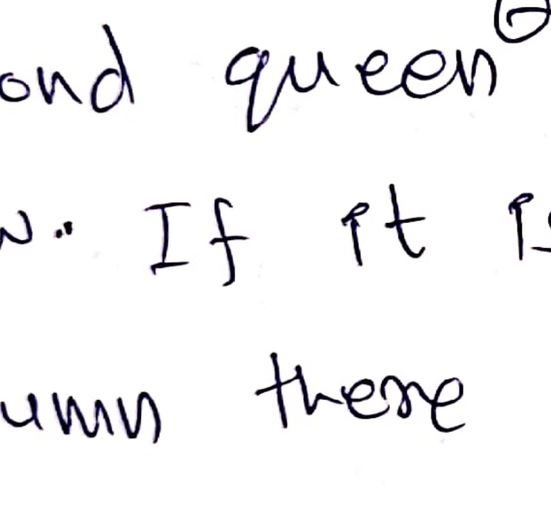
be placed in the same row, same column, or same diagonal position.

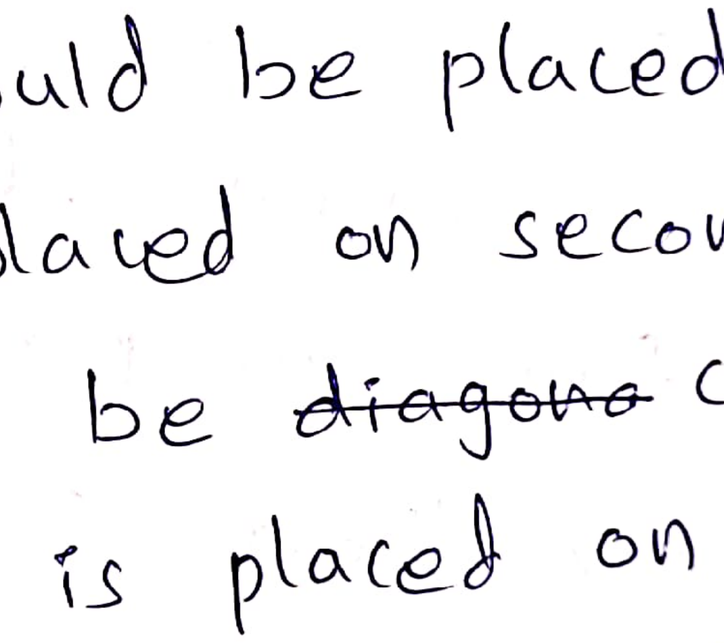
Let $\{x_1, x_2, x_3, x_4\}$ be the solution vector (one dimensional array), where x_i is column number on which the queen Q_i is placed.

First queen Q_1 is placed in first row and first column.



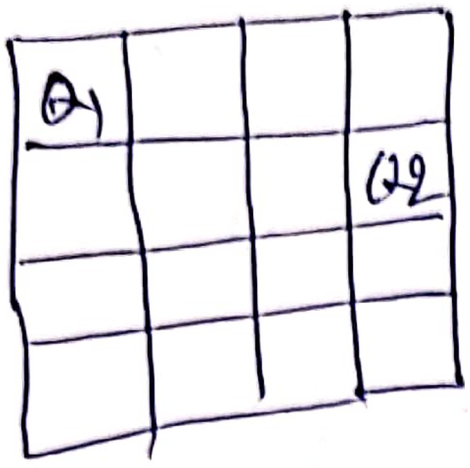
Second queen Q_2 should be placed on second row. If it is placed on second row, first column there will be ~~diagona~~ column attack from Q_1. If Q_2 is placed on second column both will be on same diagonal. So place Q_2 in third column

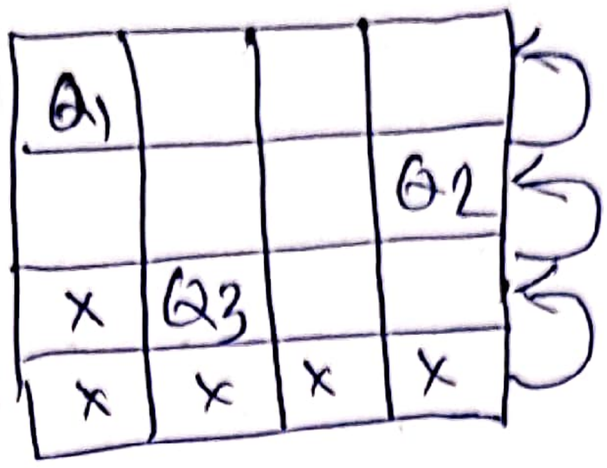




we are unable to place Q_3 in third row.

So backtrack go back to previous step and place Q_2 somewhere else.



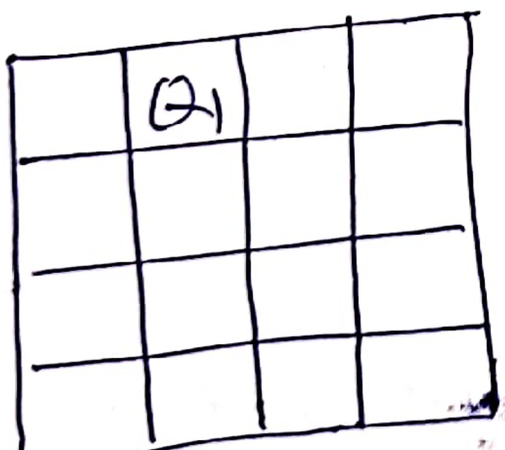


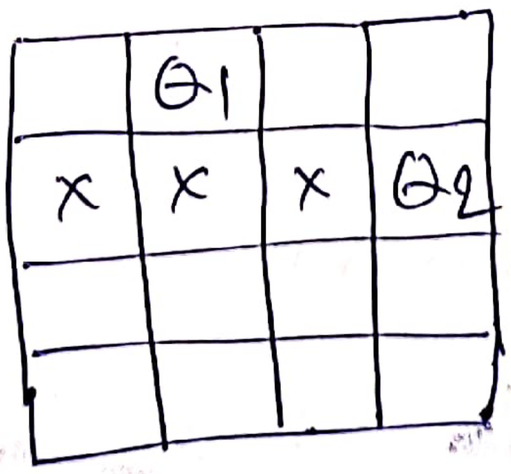
we are not able to find a nonattacking cell position for Q_4.

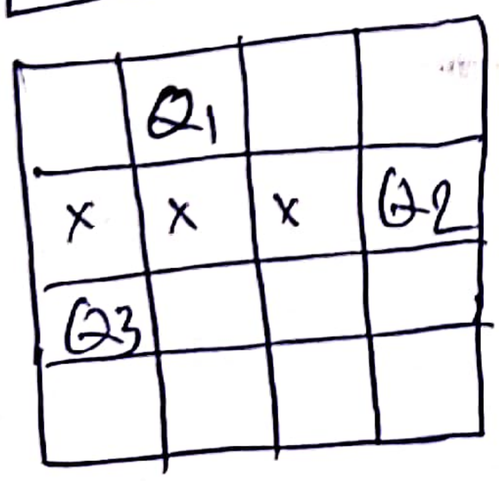
Backtrack to Q_3, but for Q_3 there is no other alternate place.

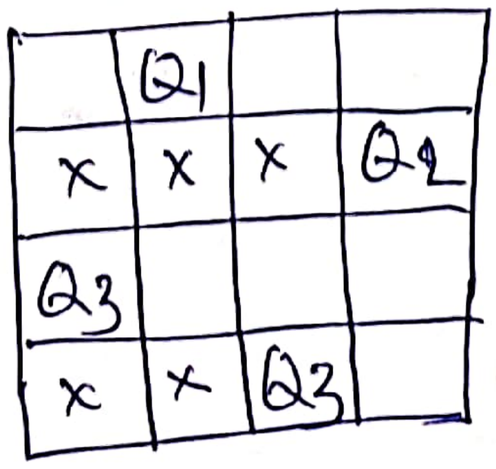
Then Backtrack to Q_2, but for Q_2 we have tried all the alternative places.

Then Backtrack to Q_1 and check whether there is any other alternate place. Q_1 can be placed on first row, second column.









solution 1

solution tuple

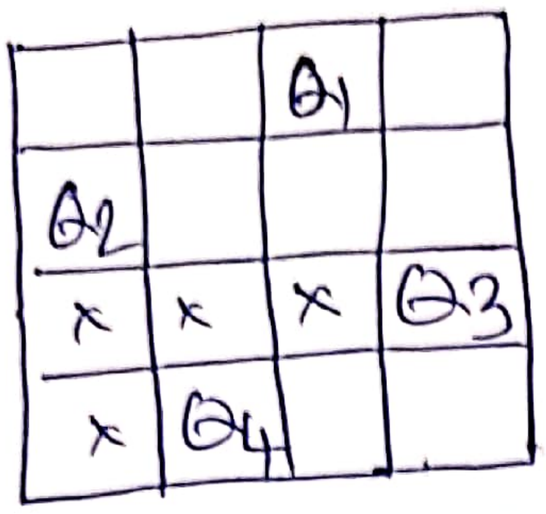
(x_1, x_2, x_3, x_4)

$(2, 4, 1, 3)$

solution vector or array

	1	2	3	4
x	2	4	1	3.

solution 2



solution tuple

(x_1, x_2, x_3, x_4)

$(3, i, 4, 2)$

solution vector (array)

	1	2	3	4
x	3	1	4	2

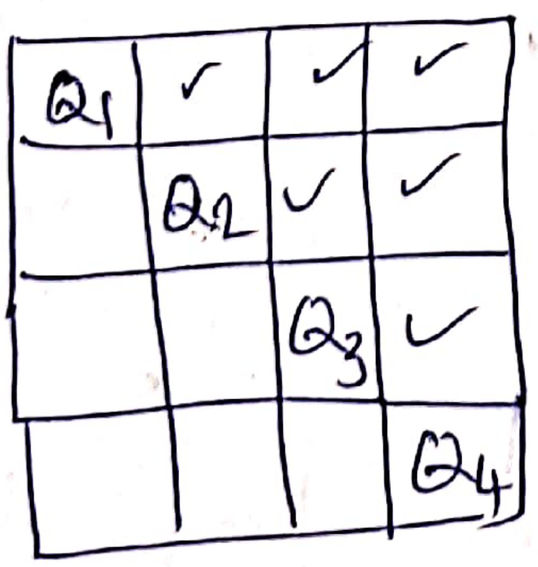
4-Queens problem has only two different (possible ways) solutions.

Basic Definitions of Backtracking

Explicit constraints are conditions that are explicitly stated and enforced during the search for a solution of a problem
— These are rules which restrict each x_i to take on values only from a given set.
Eg: For exam if for 4-Queens problem we take explicit constraints are queens must be placed on

different rows and

different columns.



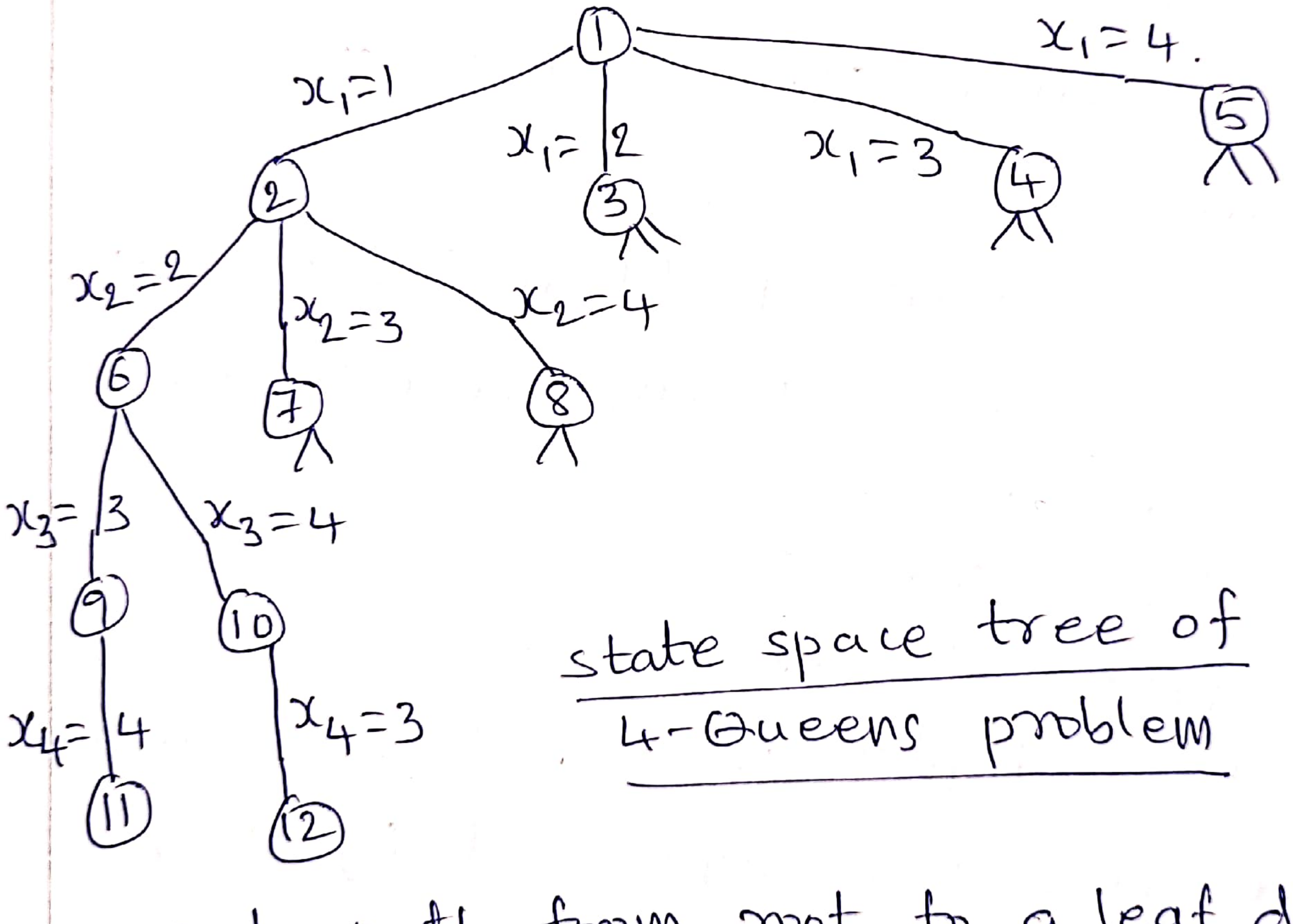
Four queens can be placed in

$4 \times 3 \times 2 \times 1 = 24 = 4!$ ways.

For 8-Queens problem 8 queens can be placed on 8×8 chess board in 8! different ways and for n-Queens problem in n! different ways.

$\therefore T(n) = O(n!)$.

solution space All tuples that satisfy the explicit constraints define a possible solution space for a problem



state space tree of 4-Queens problem

Each path from root to a leaf defines a tuple in the solution space

(x_1, x_2, x_3, x_4)

1st path (1, 2, 3, 4)

2nd path (1, 2, 4, 3)

24th path ( ).

$4 \times 3 \times 2 \times 1 = 24$.

Total 24 paths.

State space tree Tree representation of solution space is called state space tree

Bounding function (criterion function) It is a function $B(x_1, x_2, \ldots, x_n)$ which needs to be maximized or minimized or satisfied for a given problem.

Implicit constraints These are the rules which determine which of the tuples in the solution space satisfy criterion function.

Eg For 4-Queens problem, the Implicit constraints are queens must be placed on

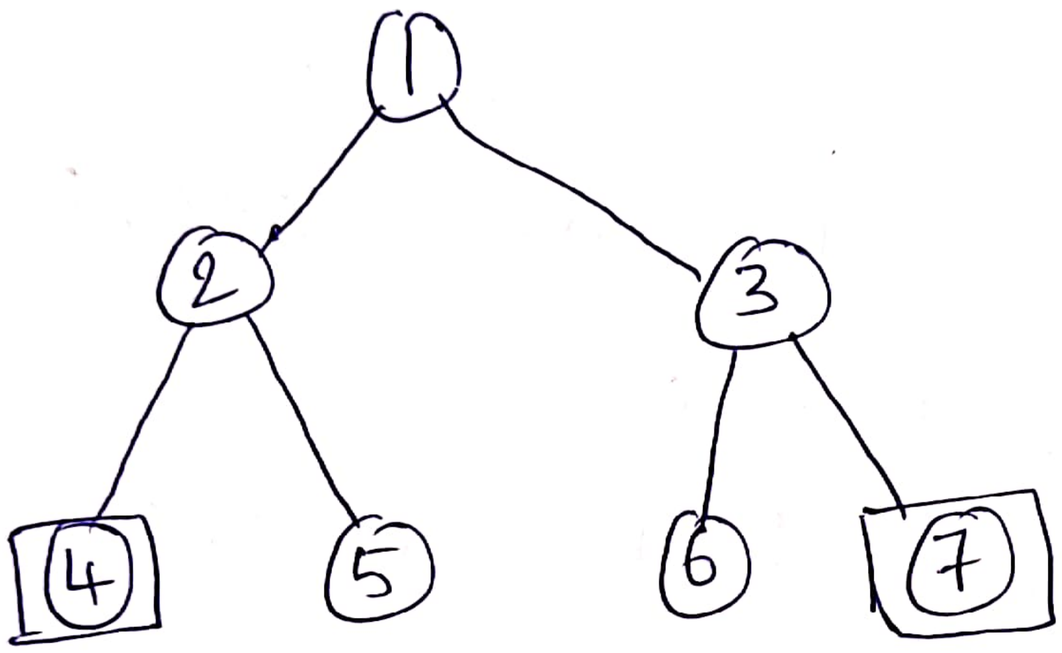
Different rows
Different columns
Different diagonals.

out of 24 tuples in the above solution space only two tuples satisfy these implicit constraints

(x_1, x_2, x_3, x_4)
$(2, 4, 1, 3)$
$(3, 1, 4, 2)$.

Problem state Each node in the state space tree is called a problem state or problem node.

solution states These are those problem states on the path from root to a leaf which define a tuple that satisfies Implicit constraints



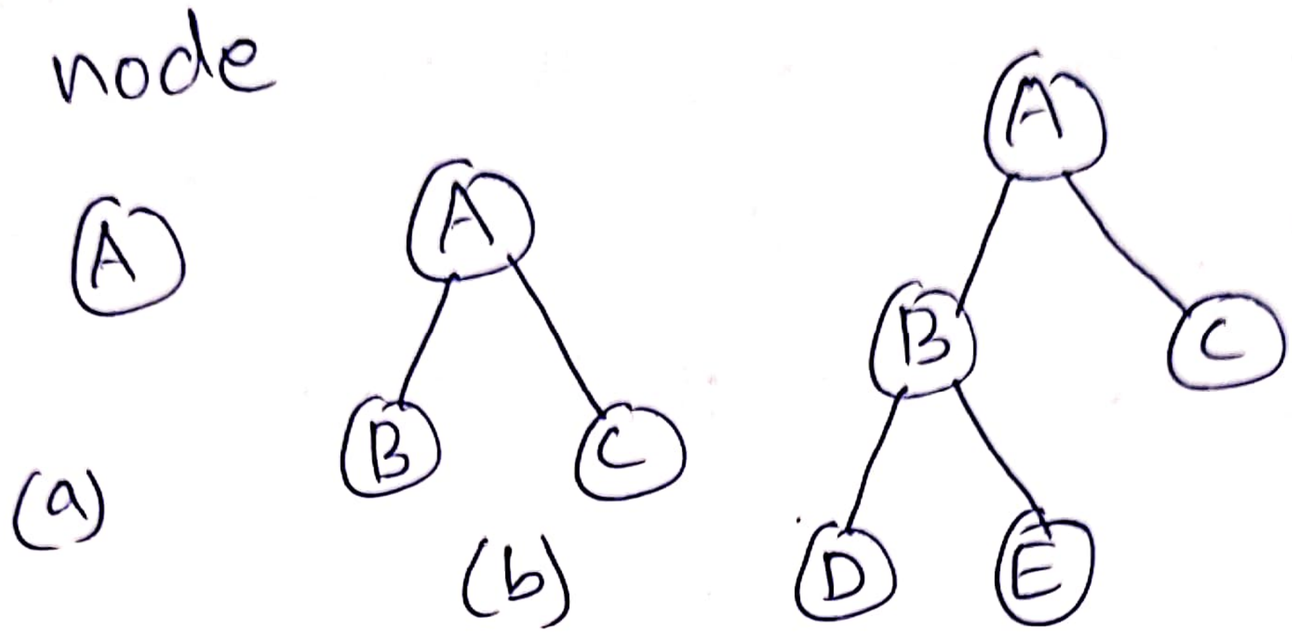
solution states are represented in the form of tuples. In above state space tree solution states are (1,2,4) and (1,3,7).

Answer states are those states s for which the path from the root to s defines a tuple that satisfies the Implicit constraints of the problem.

Answere states are represented with square nodes. In the above example answer nodes are [4] and [7].

Live node A node which has been generated, but all of whose child nodes have not yet been generated is called a live node

Eg



In Fig (a) A is a live node

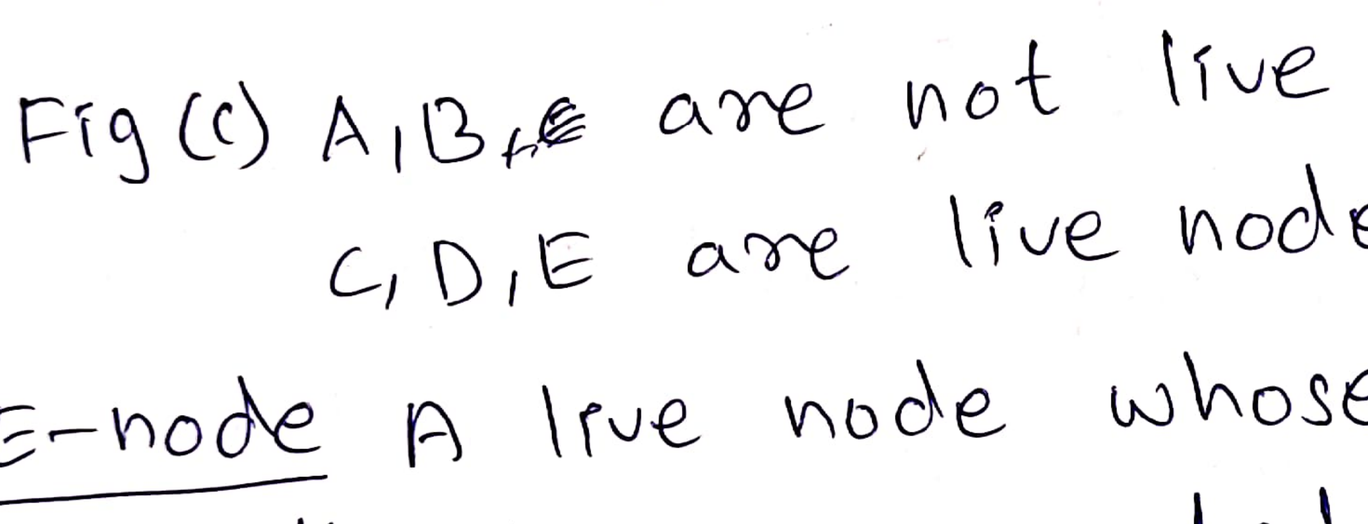
Fig (b) A is not a live node, B, C are live nodes

Fig (c) A, B are not live nodes

C, D, E are live nodes.

E-node A live node whose children are currently being generated is called an E-node [A node which is currently being expanded or explored].

Eg

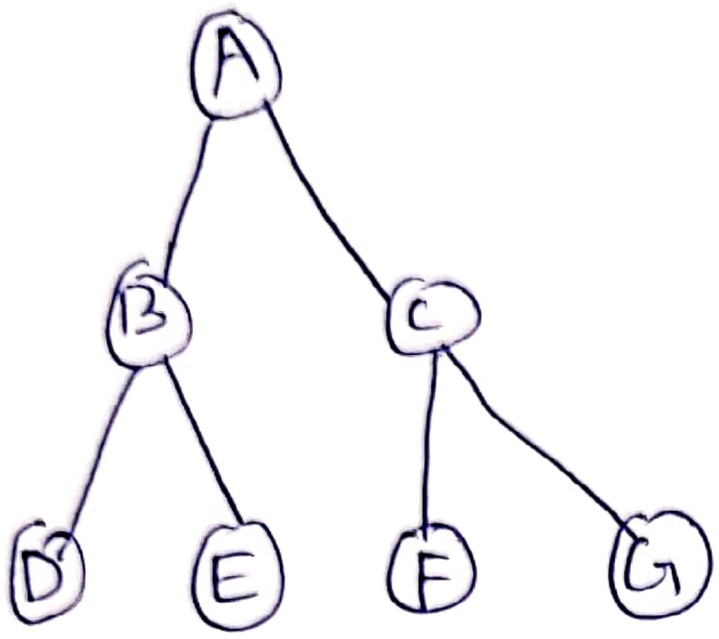


In (a) A is an E-node

(b) B is an E-node

(c) A is not an E-node

Dead node is a node that is either not to be expanded further or one for which all of its children have been generated.



A, B, C are dead nodes

D, E, F, G are live nodes.

General Method (control Abstraction for Backtracking)

Iterative

↑

Algorithm IBacktrack(n)

// This schema describes the backtracking
// process. All solutions are generated in
// x[1..n] and printed as soon as they are
// determined.

```
{
  k:=1;
while(k≠0) do
{
if(there remains an untried
      x[k] ∈ T(x[1], x[2], ..., x[k-1]) and
      B_k(x[1], ...x[k]) is true) then

  {
```

```
if (x[1], x[2], . . . , x[k] is a path to an answer
        node) then
{
    write (x[1:k]);
    return;
}
→ k:=k+1; // Go to next step.
else
  k:=k-1; // Backtrack to the previous step
} // end of while loop
} // end of algorithm
```

General Iterative Backtracking method

$T(x_1, x_2, \cdots, x_{k-1})$ be the set of all possible values for x_k

↓
Tuple

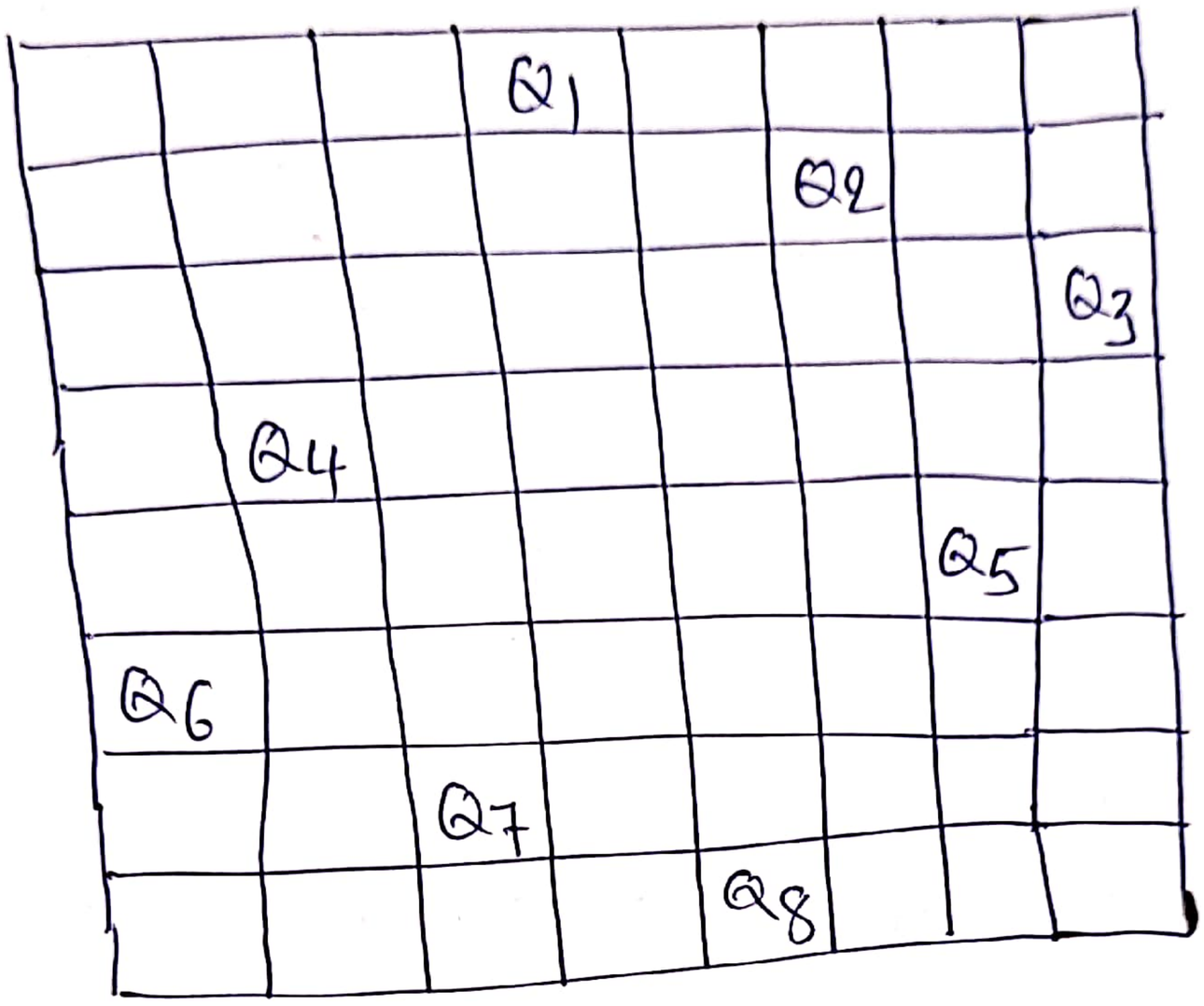
B_k is bounding function or criterion function.

Applications of Backtracking

n-queens problem
sum of subsets problem
Graph coloring problem.

8-Queens problem

8-Queens problem has 92 different solutions. The following is one of the solution.



The solution is

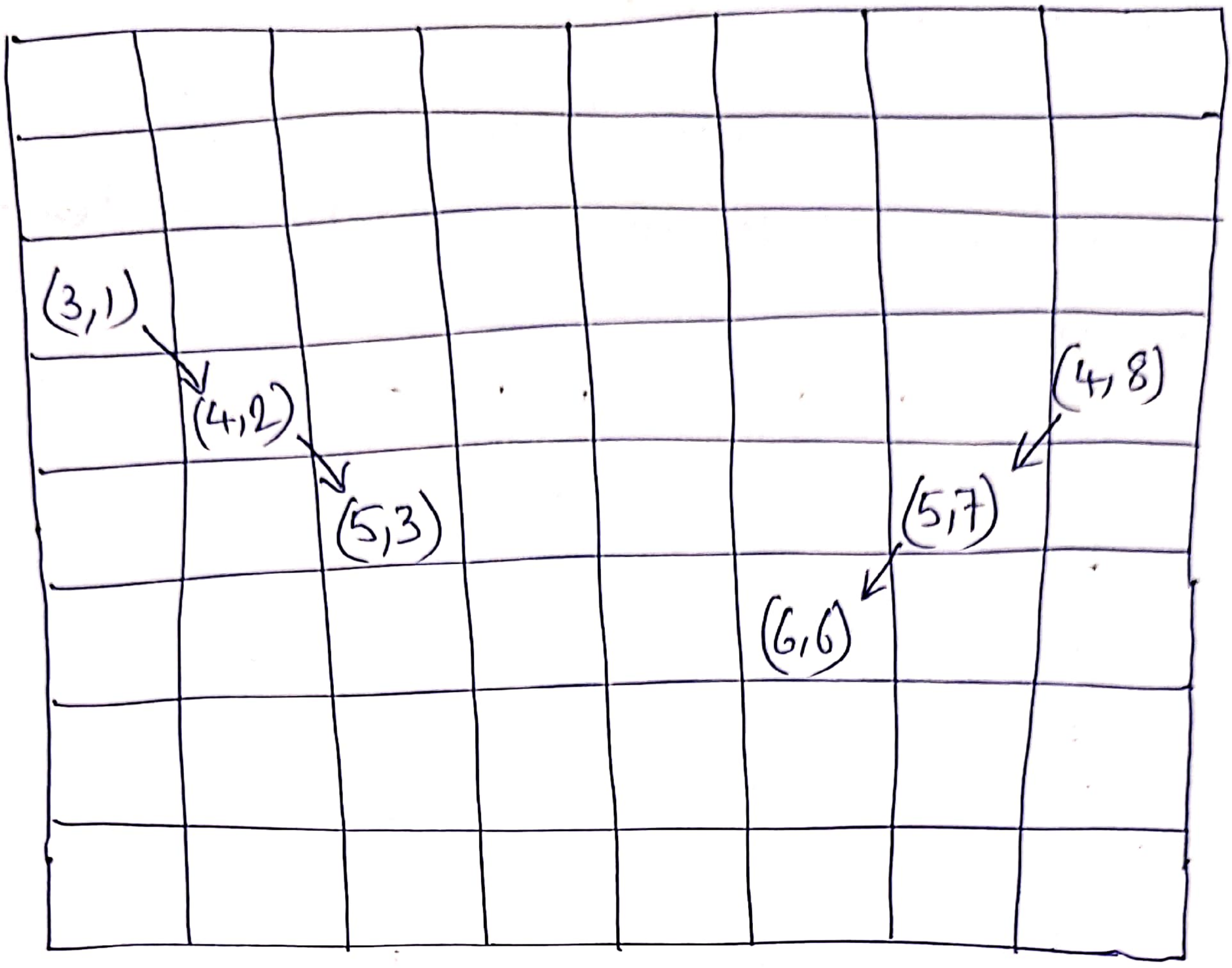
$(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) = (4, 6, 8, 2, 7, 1, 3, 5)$.

n-Queens problem

Let a[1:n, 1:n] be the two dimensional array representing the squares of chess board. Suppose two queens are placed at positions (i, j) and (k, l)

They will be on different row if $i \neq k$

They will be on different column if $j \neq l$.



Every element on the same diagonal that runs from the upper left to the lower right has the same row-colum value.

Eg (3,1), (4,2), (5,3)

$3-1 = 4-2 = 5-3 = 2$

→ Every element on the same diagonal that goes from upper right to the lower left has the same row+column value

Eg (4,8), (5,7), (6,6)

$4+8 = 5+7 = 6+6 = 12.$

suppose two queens are placed at positions coordinates (i, j) and (k, l)

Then by the above rules they are on the same dragonal only if

$$\boxed{i-j = k-l \quad \text{or} \quad i+j = k+l.}$$

The first equation implies

$$\boxed{j-l = i-k}$$

The second implies $\boxed{j-l = k-i}$

Therefore the two queens lie on different diagonals + iff [not on same diagonal]

$$\boxed{|j-l| \neq |i-k|}$$

All solutions to the n-Queens problem

```
Algorithm NQueens(k,n).
// Using backtracking, this procedure prints all
// possible placements of n queens on an
// nxn chessboard so that they are in
// nonattacking cell positions
{
   for l=1 to n do
      {
```

```
If Place(k,i) then // can Qk be placed on (k,i)?
{
   x[k]:=i; // Queen Qk is placed on (k,i)
   if(k=n) then write(x[1:n]); // All queens
                               // are placed
   else NQueens(k+1,n); // Go to next queen
   }
}
}
```

Time complexity is O(n).

---

can a next queen Q_k be placed in (k,i)?

```
Algorithm Place(k,i)
// This algorithm returns true if a queen
// Qk can be placed in kth row and ith
// column. otherwise it returns false. x[]
// is a global array whose first (k-1) values
// have been set. Abs(r) returns the
// absolute value of r.
{
for i:=1 to k-1 do
{
if ((x[i]=i  // Queen Qk and one of the
     or      // previous queens (Q1,Q2,..Qk-1)
Abs(x[i]-i)=  // are on the same column
   Abs(x[i]-i)=Abs(i-k)) // on the same diagonal.
```

$(i, x[i]), (k, i)$

```
if ((x[i]=k) or (Abs(x[i]-k) = Abs(i-k)))
 then return false; // Qk cannot be placed
                    // on (k,k)
}
else return true; // Qk can be placed on (k,k)
}
```

computing time of this algorithm since it has a for loop which runs

(k-1) times $O(k-1)$

Time complexity of n-Queens problem is

$O(n!)$

since we require the examination of at most n! cell position $[4!, 8!]$

SUM OF SUBSETS PROBLEM

Suppose we are given n distinct positive numbers (elements) usually called weights $W = (w_1, w_2, \ldots, w_n)$. in increasing order Then find out all subsets whose sum is M. This is called the sum of subsets problem.

solution vector $(x_1, x_2, \ldots x_n)$

$x_i = 0$ if w_i is not placed in the subset

$x_i = 1$ if w_i is placed in the subset.

Let us draw the state space tree to find the solution to sum of subsets.

All paths from root to a leaf node define a solution spare.

The left subtree of the root defines all subsets containing w_1, and the right subtree defines all subsets not containing object w_1.

the bounding function

$B_k(x_1, x_2, \ldots x_k)$ is true iff

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

and

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$$

and

$$w_1 \leq m$$

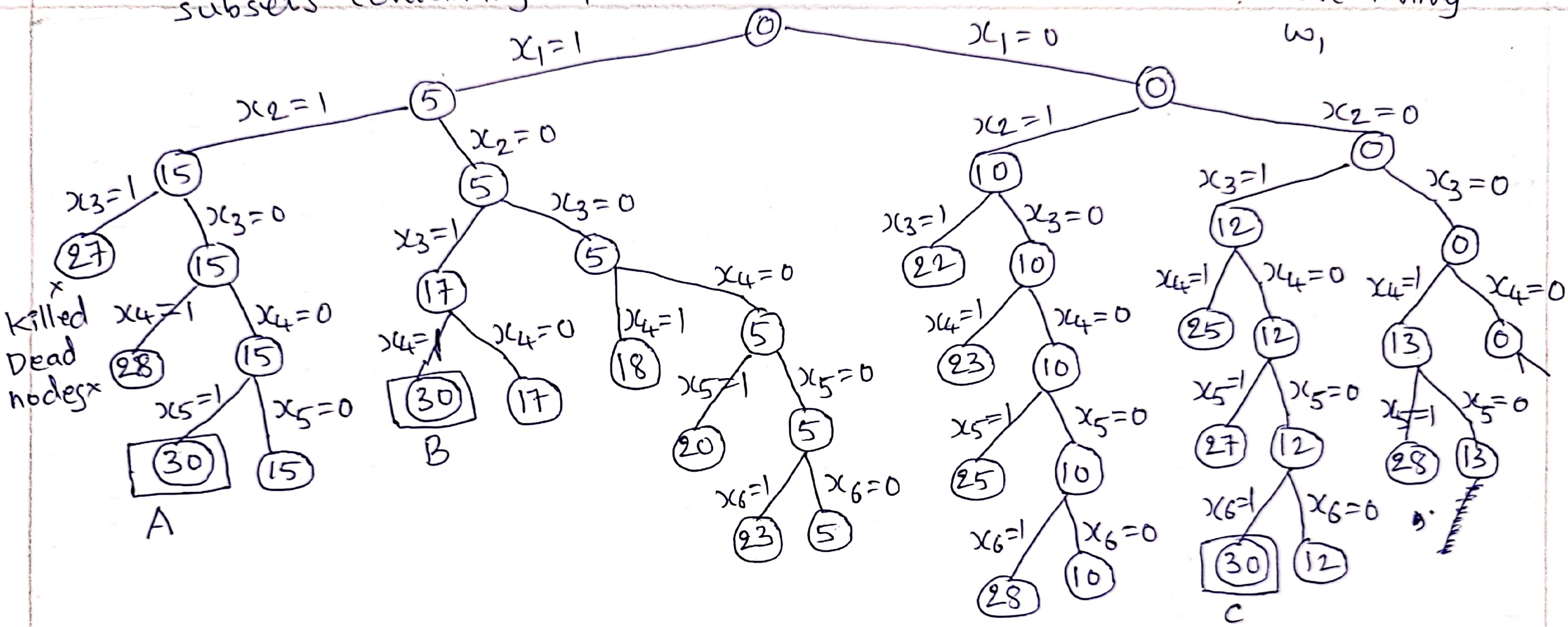
Feasible solution is a solution that satisfies bounding function.

Eg consider a set $S=\{5,10,12,13,15,18\}$ and $M=30$.

Find out all subsets whose sum $M=30$ using backtracking.

sol:- No. of objects $n=6$.

subset {empty}	sum(0)	Action (Initially subset is empty)
5	5	
5,10	15	
5,10,12	27	
5,10,12,13	40	sum exceeds M=30 Not a feasible solution
5,10,12,15	42	Not a feasible solution
5,10,12,18	45	Not a feasible solution Backtrack to element 10
5,10	15	
5,10,13	28	
5,10,13,15	43	Not a feasible solution
5,10,13,18	46	Not a feasible solution Backtrack to 10
5,10	15	
5,10,15	30	sum=30 solution1 obtained. found.
10,12		similarly find other subsets whose sum=30.



A, B, C are answer nodes

output solution tuples $(x_1, x_2, x_3, x_4, x_5, x_6)$

$(1,1,0,0,1,0) \rightarrow A$

$(1,0,1,1,0,0) \rightarrow B$

$(0,0,1,0,0,1) \rightarrow C.$

For n elements
Full state space tree contains
$2^n - 1$ leaf nodes or branches
$\therefore T(n) = O(2^n).$

State space tree for sum of subsets problem

For given set $S = \{5, 10, 12, 13, 15, 18\}$ and $M = 30$

$(w_1, w_2, w_3, w_4, w_5, w_6) = (5, 10, 12, 13, 15, 18)$ weights.

we have found three subsets

1) $(x_1, x_2, x_3, x_4, x_5, x_6) = (1, 1, 0, 0, 1, 0)$

sum of the weights

$= w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6$

$= 5 \times 1 + 10 \times 1 + 12 \times 0 + 13 \times 0 + 15 \times 1 + 18 \times 0$

$= 5 + 10 + 0 + 0 + 15 + 0 = \underline{30}$

2) $(x_1, x_2, x_3, x_4, x_5, x_6) = (1, 0, 1, 1, 0, 0)$

$= w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6$

$= 5 \times 1 + 10 \times 0 + 12 \times 1 + 13 \times 1 + 15 \times 0 + 18 \times 0$

$= 5 + 0 + 12 + 13 + 0 + 0 = \underline{30}$

3) $(x_1, x_2, x_3, x_4, x_5, x_6) = (0, 0, 1, 0, 0, 1)$

$= w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6$

$= 5 \times 0 + 10 \times 0 + 12 \times 1 + 13 \times 0 + 15 \times 0 + 18 \times 1$

$= 0 + 0 + 12 + 0 + 0 + 18 = 30.$

Recursive backtracking algorithm for sum of subsets problem

```
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 = Σ_{j=1}^{k-1} w[j]*x[j]  and
// r = Σ_{j=k}^{n} w[j]. The w[j]'s are in nondecreasing
// (increasing) order. It is assumed that w[1] ≤ m
// and Σ_{i=1}^{m} w[i] ≥ m
{
// Generate left child. Note: s+w[k] ≤ m since B_{k-1} is true
x[k]:=1;
if (s+w[k]=m) then write(x[1:k]); //subset found.
// There is no recursive call here as w[j]>0, 1≤j<n
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]);
}
}
```

For set of n elements, the no. of subsets $= 2^n$ $\therefore T(n) = O(2^n)$.

GRAPH COLORING PROBLEM

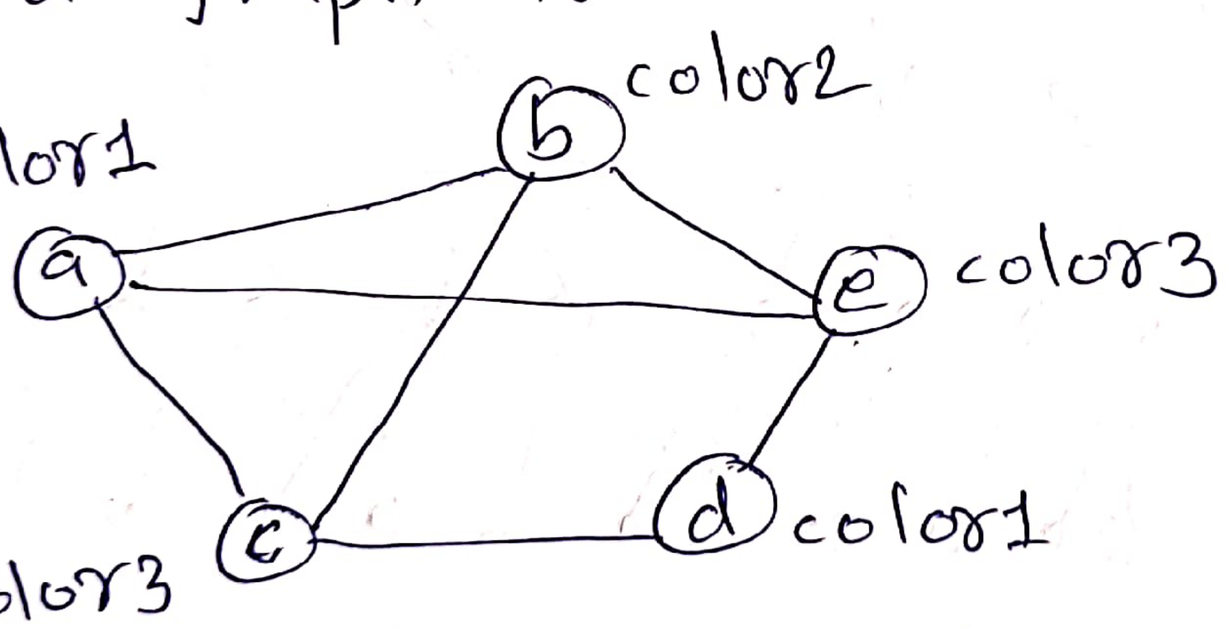
Let G be a graph and m be a given positive integer. Graph coloring is a problem of coloring each vertex in graph in such a way that no two adjacent vertices have (get) same color. and yet m colors are (should) be used.

This problem is also called m-coloring problem.

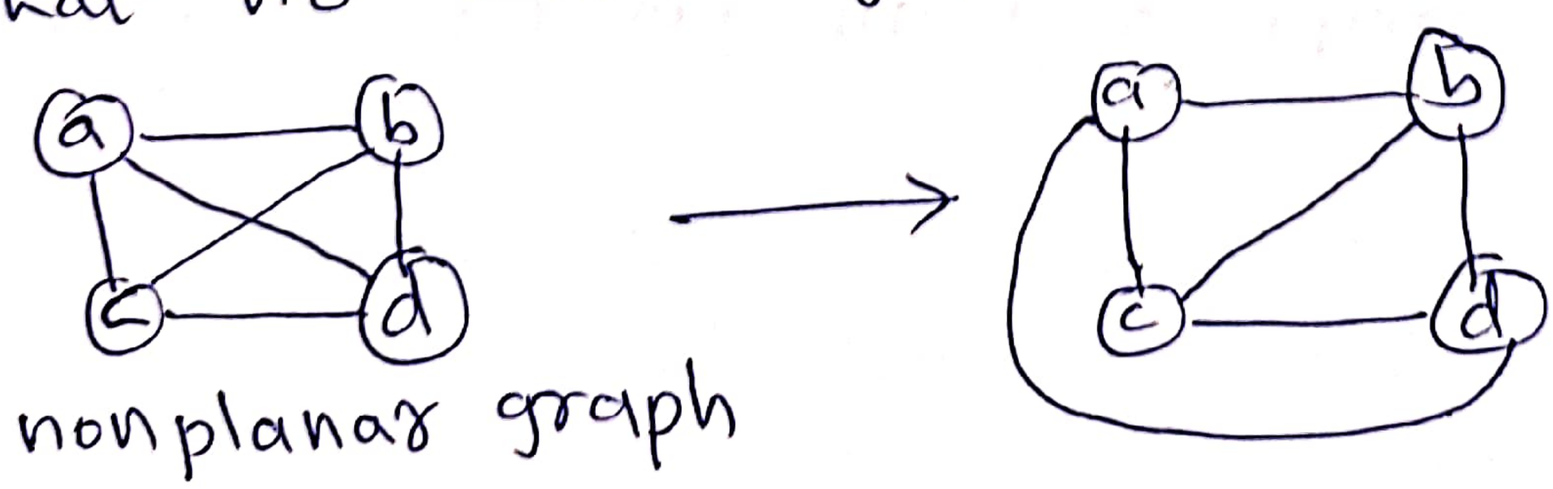
If the degree of a given graph is d, then it can be colored with d+1 colors.

The minimum number of colors required to color a graph is called its chromatic number.

Eg



→ A graph is said to planar graph iff it can be drawn on a plane in such a way that no two edges cross each other

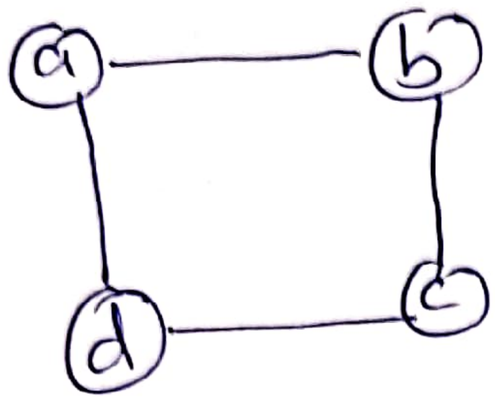


nonplanar graph

planar graph.

The chromatic number of any planar graph is 4.

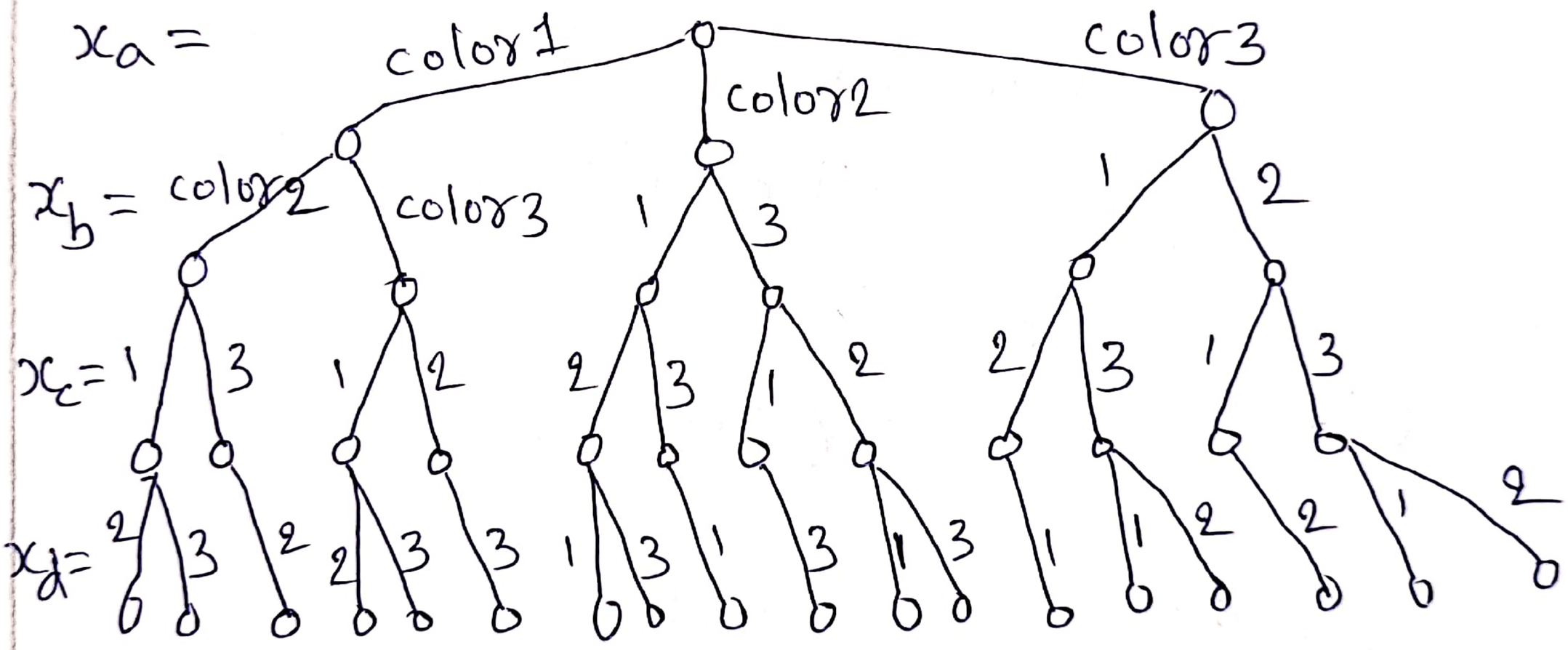
Eg Generate all possible 3-colorings for the following 4-node graph by drawing a state space tree



sol:- Given three colors. Let us represent these colors with color1, color2, color3 or 1,2,3.

x_a = color that is used to color vertex a

similarly x_b, x_c, x_d.



18-different solutions (colorings).

A 4-node graph and all possible 3-colorings

Finding all m-colorings of a graph

```
Algorithm mColoring(k)
// This algorithm was formed using the recursive
// backtracking schema. The graph is represented
// by its boolean adjacency matrix G[1:n,1:n].
// All assignments of 1,2,...,m to the vertices of
// the graph such that adjacent vertices are
// assigned distinct (color) integers are printed.
// k is the index of the next vertex to color.
{
 repeat
 {// Generate all legal (valid) color 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]); // terminate algorithm
   else
    mColoring(k+1);
 } until(false);
}
```

Algorithm to generate a next color

```
Algorithm NextValue(k)
// x[1], x[2], ..., 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 number
//ed 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 color have been
                           //used. No distinct color
                           // possible

   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 an edge and if adjacent
   //vertices have the same color.
     then break;  // j is adjacent to k and
}                 //has the same color.
```

```
if (j = n+1) then return; // new color found.
}
until (false); // otherwise try to find another
               // color.
}
```

→ Time complexity of m-coloring problem to color an n-node graph is $O(m^n)$. since at each of the n vertices we have to make a selection of one color from m colors.

$= node1 \times node2 \times \cdots \times node\ n$

$= (m\ choices) \times (m\ choices) \times \cdots \times (m\ choices)$ n times

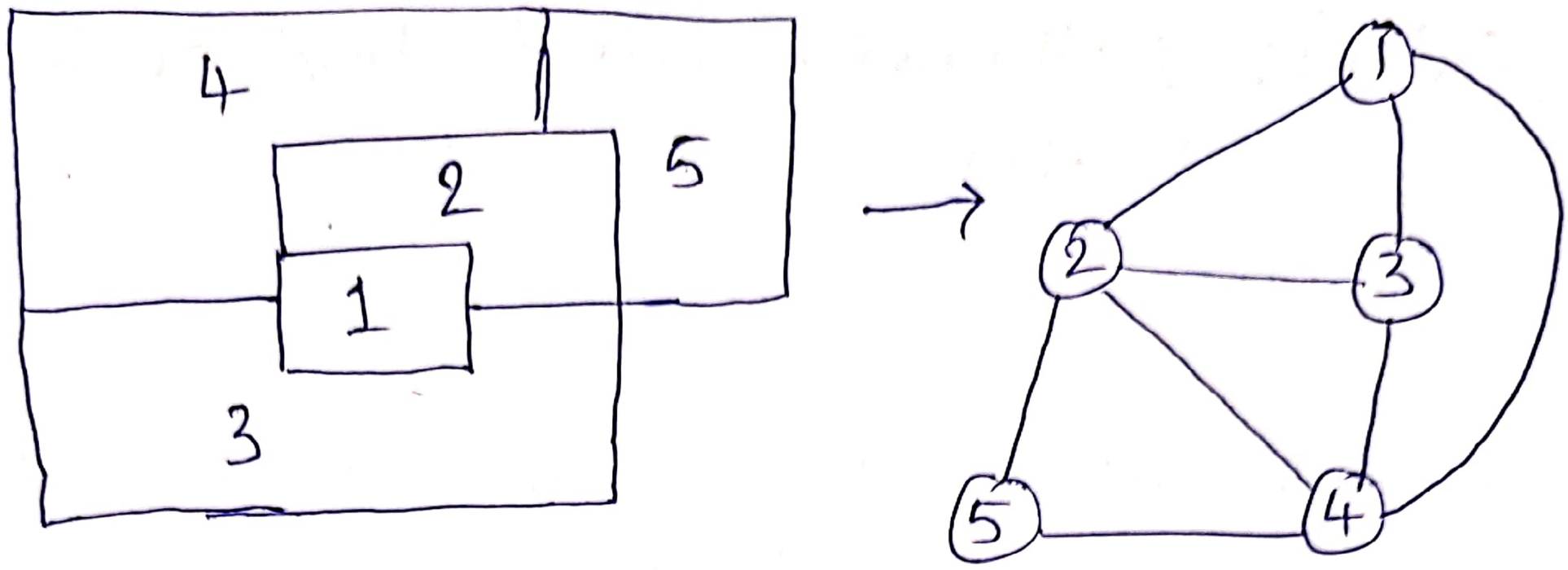
$= m^n$

$\underline{T(m,n) = O(m^n)} = T(m\text{coloring})$

→ Since the graph is represented in the form of boolean adjacency matrix (2-D array) $G[1:n, 1:n]$. space complexity $= O(n^2)$.
$G[n][n]$

→ one of the applications of graph coloring is, it is used to color the different regions of a map.

A map and its planar graph representation



map with five regions and its corresponding planar graph representation

→ Each region of the map becomes a node, and if two regions are adjacent, then the corresponding nodes are joined by an edge.

Degree of this map (graph) is 4.

It requires minimun ~~four~~ five colors.