UNIT- IV

(1) What is mean by backtracking? How does it help to solve problems.

<u>Backtracking</u>

⇒ Backtracking is a systematic method for searching one or more solutions for a given problem.

⇒ Backtracking is a design technique for finding solutions for enumeration, decision and optimization problems.

⇒ Backtracking is a refinement of brute force technique and uses a modified DFS search.

⇒ Backtracking starts with an empty solution vector. then it adds components in a step-by-step fashion, by checking the node with bounding functions. If the node is not a promising node, then backtracking occurs.

(2) What are implicit and explicit constraints?

<u>Implicit constraints</u> ⇒ Implicit constraints are rules that limit the generation of processing of a solution vector that maximizes, minimizes or satisfies the criterian function expressed as a vector $(x_1, x_2, \cdots x_n)$.

<u>Explicit constraints</u> ⇒ Explicit constraints are rules that restrict a component of a solution vector say x_i from choosing a specific value from a set S.

(3) Define the following terms: live, E-node, dead node and state space tree.

<u>Answer state</u> ⇒ these are solution states where the path from the root to the leaf defines the solution of the problem.

<u>Live node</u> ⇒ A node that has been generated already but is yet to generate the children.

<u>E-node</u> ⇒ A node that is under consideration and is in the process of being generated.

<u>Dead node</u> ⇒ A node that is already explained and cannot be considered for further searches.

<u>State space tree</u> ⇒ It is an arrangement of all possible solutions in a tree-like fashion. this can be a binary tree whose children are fixed (fixed tuple tree) or a tree whose children vary (variable-tuple tree).

Node ⇒ Represents a partial solution from root to that node.

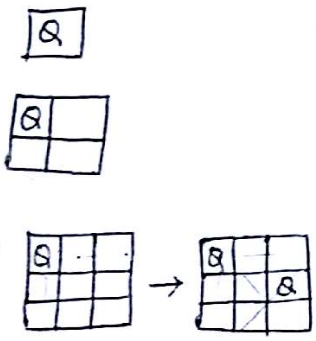
Edge ⇒ Transition from state.

(4) Do solutions exist for one, two, and three queen problems? Justify.

One-queen

Two-queen

Three-queen

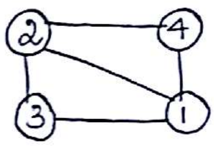


⇒ It can be observed no solution exists for one-, two- or three solu' queen problems for the board 1x1, 2x2 and 3x3.

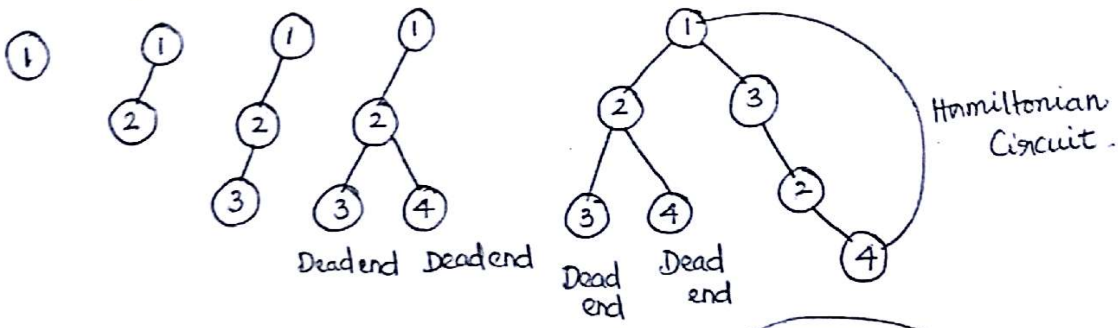
⇒ In this cases, the queens cannot be placed in <u>non-attacking</u> positions.

⇒ since the constraints for this problem is non-attacking placement of Queen one, two and three queen placement is not possible.

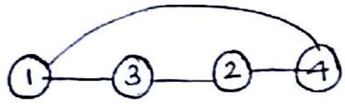
(5) What is Hamiltonian cycle. Consider the following graph and show that it has a hamiltonian cycle.



A Hamiltonian cycle of a graph starts from a vertex, visit all other vertices only once and returns back to the original vertex.



Hamiltonian cycle for the given graph is



1) How backtracking works on the NXN Queens problem and place the 4 Queens in 4X4 chessboard using backtracking with suitable algorithm.

Answer:-

The objective of this problem is to place N queen on an NXN chessboard such that no two queens are in an attacking position.

⇒ A 4-queen problem requires four queens to be placed in such a way that no two queens are in an attacking position.

⇒ The initial solution vector for the 4-queen problem can be denoted as (x_1, x_2, x_3, x_4).

Here every component vector x_i represents either a column where the placement of a queen is possible satisfying the constraints.

⇒ Let us denote col(i) as the column where a queen is located in the row i.

col(i) ensures that the column is chosen only once. The rows can be tested whether col(i) equals col(k).

⇒ In general, the promising node is stated as follows,

(1) col(i) = col(k) ⇒ shows that queens are in the same row in respective columns.

(2) col(i) − col(k) = i−k or col(i) − col(k) = k−i ⇒ whether the queens are in diagonal positions.

Informally,

Step 1: Start from the first column, check row by row, and place a queen.

Step 2: Move on to the next column and place the next queen.

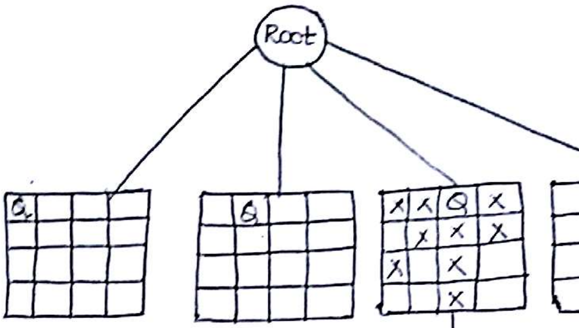
Step 3: Check if the placement of queen is safe.

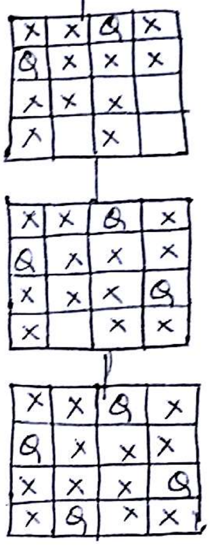
Step 4: If it is safe, print the solution; else remove the last queen and backtrack.

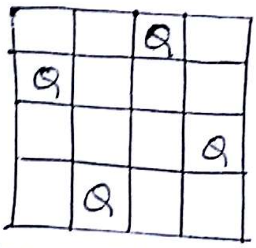
Step 5: If solution is not found; then report an error.

```
Algorithm queen(i)
I/p : Queen i
O/p : placement of queen i given by col(i)
Begin
    if promising(i) then
        if (i == n) then
            Print col[1].... col[n]
        end if
    else
        for j=1 to n do          // all column j
            col[i+1] = j
                queen[i+1];
        end for
    end if
End
```

```
Algorithm promising(i)
i/p: Queen i
o/p: Status about the feasibility of the
     placement of queen i as true or false
Begin
   flag=true
   for k=1 to i-1 do
      if (col[i] = col[k]) then
         if |col[i]-col[k]| = |i-k|
                or
            (col[i] == col[k]) then
                 flag=false
         End if
      End if
   End for
      return(flag)
End.
```







One possible solution to 4-Queen problem.

Promising function ⇒ Check whether the queens are placed in the same row or diagonal. If so, it sets the flag as false and returns it. However, if the queens are not in an attacking position, then the function returns flag as true.

Complexity Analysis:

Number of nodes that will be generated are

$$1+4+4^2+\dots+4^4 = \frac{4^{4+1}-1}{4-1} = \frac{4^5-1}{3}$$

Constraint that no two queens in an attacking position reduces these constraints to $1+4+4\times3+4\times3\times2+4\times3\times2\times1$ promising nodes.

∴ In general $1+n+n(n-1)+n(n-1)(n-2)+\dots+n!$ promising nodes are possible.

(2) Write an algorithm to determine the sum of subsets for a given sum and a set of numbers. Draw the tree representation to solve the subset problem given the number set as {5, 10, 15, 20, 25} with the sum = 30. Derive all the subsets.

Answer:

⇒ Sum of subsets is the variant of the knapsack problem. The sum of subsets extends this problem to that of checking whether it is possible to find subsets of items whose sum of weights equals W.

⇒ Formally, Given n positive items with weights $w_i = \{w_1, w_2, \ldots w_n\}$ and a positive integer W, the problem is about finding all the combinations of items whose sum of weights amounts to W.

⇒ The weights are usually in an ascending order of magnitude and unique. The solution of this problem is often expressed as a solution vector X, where the inclusion of an item is indicated by '1' and exclusion by '0'.

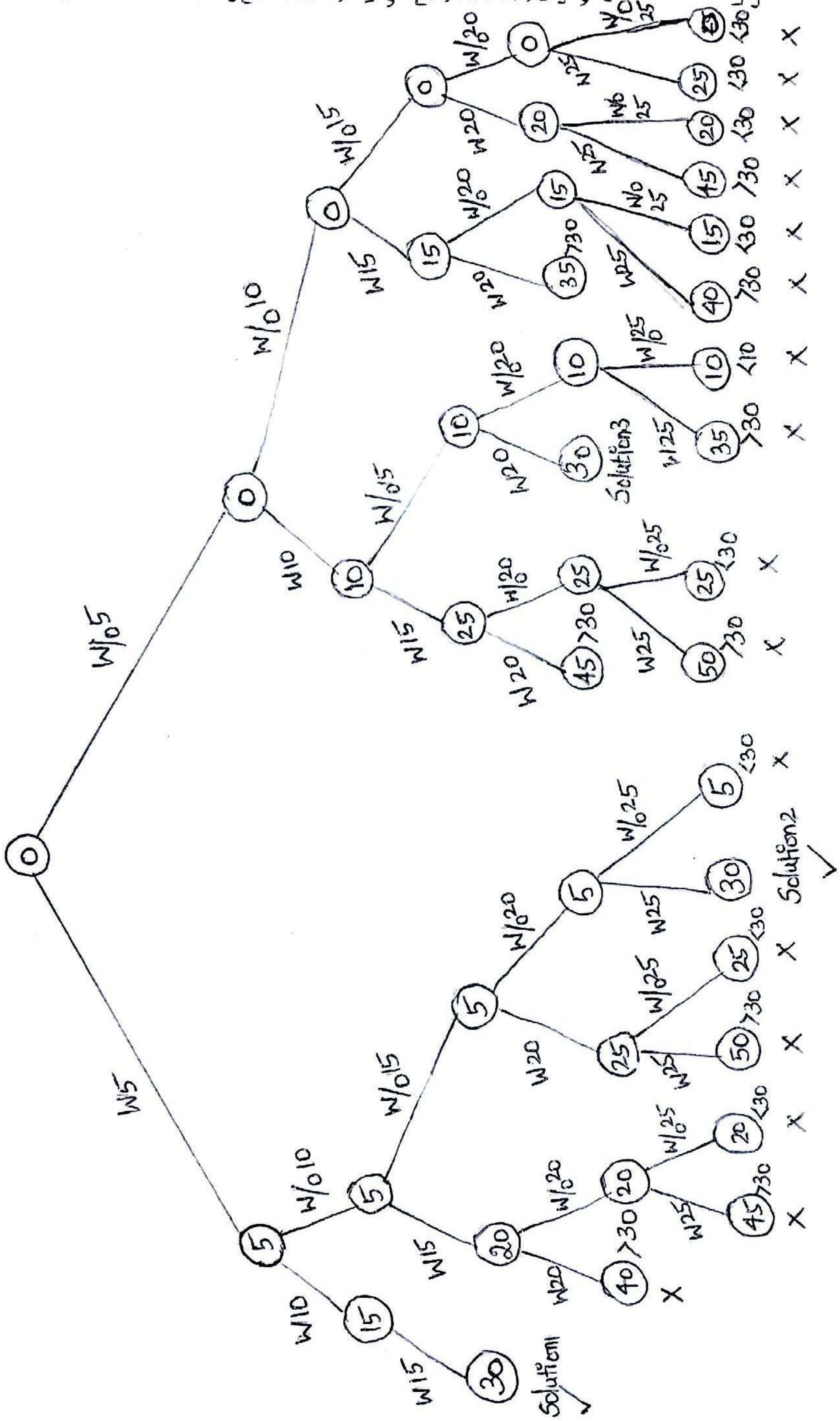
```
Algorithm Sum-of-subsets(i, weight, total, W)

I/p: Item i, weight - weight of an item i
     total - remaining weight,
O/p: Items whose sum equals W.

Begin  if promising-Sum-of-subsets(i)
          if (weight == W) then
              Print items x[1....i]
          end if
       else
          x[i+1] = 1
          Sum-of-subsets(i+1, weight+W_{i+1}, total-W_{i+1}, W)
             x[i+1] = 0
          Sum-of-subsets(i+1, weight, total-W_{i+1}, W)
       End if
     End

Algorithm  promising-Sum-of-subsets(i)
I/p: Item i, o/p: Status about the feasibility of including item
Begin  flag = true
       if ((weight+total ≥ W) and (weight == W) or (weight + W_{i+1} ≤ W)) then
          flag = false
       endif
          return flag
End
```

Pbm solution: Given, W = {5, 10, 15, 20, 25} and Sum m = 30.

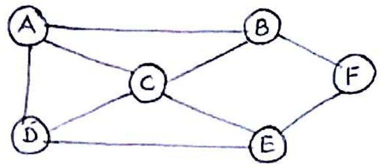
The different combinations to obtain m=30 are as follows,

Solution set = { [5, 10, 15] , [10, 20] , [25, 5] }

Solution vector = { [1, 1, 1, 0, 0] , [0, 1, 0, 1, 0] , [1, 0, 0, 0, 1] }



③ Write an algorithm to determine Hamiltonian cycle in a given graph using backtracking. For the following graph determine the Hamiltonian cycle.



Answer:

⇒ Hamiltonian cycle is a cycle that traverses all the vertices of a given graph G exactly once and ends at the starting vertex.

⇒ The input for a Hamiltonian circuit problem is an undirected graph.

Promising for Hamiltonian problem:

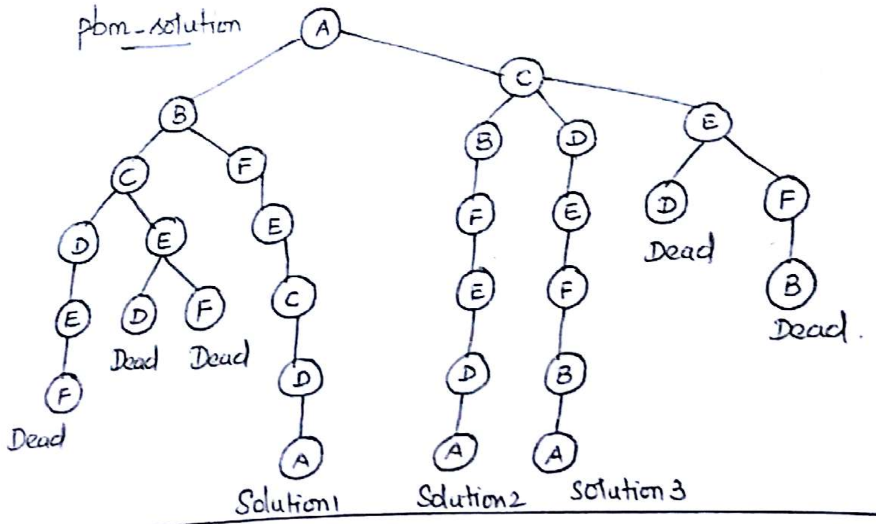
The constraints for a Hamiltonian problem are as follows,

1. The i^{th} vertex in the path must be adjacent to the $(i-1)^{th}$ vertex in any path.
2. The starting vertex and the $(n-1)^{th}$ vertex should be adjacent.
3. The i^{th} vertex cannot be one of the first $(i-1)$ vertices

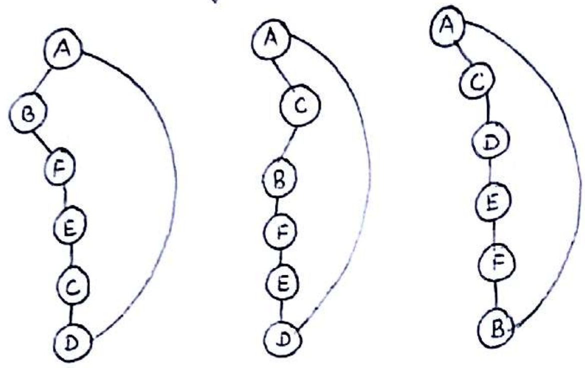
```
Algorithm Hamiltonian(i)
I/p: node i - initially the starting node.
O/p: Hamiltonian cycle if exists.

Begin
  if promising-Hamiltonian(i) then
     if (i==n-1) then
        Print V[0]....V[n-1]
     end if
  else  j=2
     while (j<=n) do
        V[i]=j
        Hamiltonian(i+1)
        j=j+1
     End while
  End if
End
```

```
Algorithm promising-Hamiltonian(i)
i/p : Node i
o/p : Status of visiting of node i

Begin  flag=true
   for j=1 to i-1 do
      if (vertices Vi and Vj are neighbours) then
           flag=false
      End if
   End for
   if (vertices Vi and Vi-1 are neighbours) then
           flag=true
   else
           flag=false
   End if
           Return (flag)
End
```

pbm-solution



∴ Hamitonian cycles are,



Complexity :

Number of nodes in the state Space tree

$$1+(n-1)+(n-1)^2+\dots+(n-1)^{n-1}$$

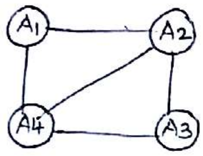
$$=\frac{(n-1)^n-1}{n-2}$$

Number of edges in the graph $= n$

cost of one edge $= c$

Total cost for all edges in the graph $= Cn = O(n)$.

(4) Explain in detail about Graph coloring algorithm and discuss minimum colour required to colour the following graph and draw State Space tree.



Solution:

⇒ Graph coloring problem assign 'M' colours to the vertices of a graph G such that the adjacent vertices of the graph G do not share the same colour.

⇒ The input for the algorithm is the adjacency matrix of a graph whose entries are 1 for the vertices that share an edge and 0 if no edge is shared between the vertices.

```
Algorithm Colouring(i)
i/p: node i
o/p: Colours of vertices of graph G,
     (ie) array colour[1....n]
Begin
  if promising-Colouring(i) then
    if(i==n) then
      Print colour[1....n]
    else
        j=1
        while( j<=n) do
            colour[i+1]=j
            Colour [j+1]
               j=j+1
        End while
    End if
  End if
End.
```

```
Algorithm promising-colouring(i)
  i/p: node i
  o/p: Status of coloring of node i
Begin
     flag=true
   for j=1 to i-1 do
     if (vertices i and j are neighbours) then
       if (colour(i)== colour(j)) then
           flag=false
       End if
     End if
   End for
     return (flag)
End
```

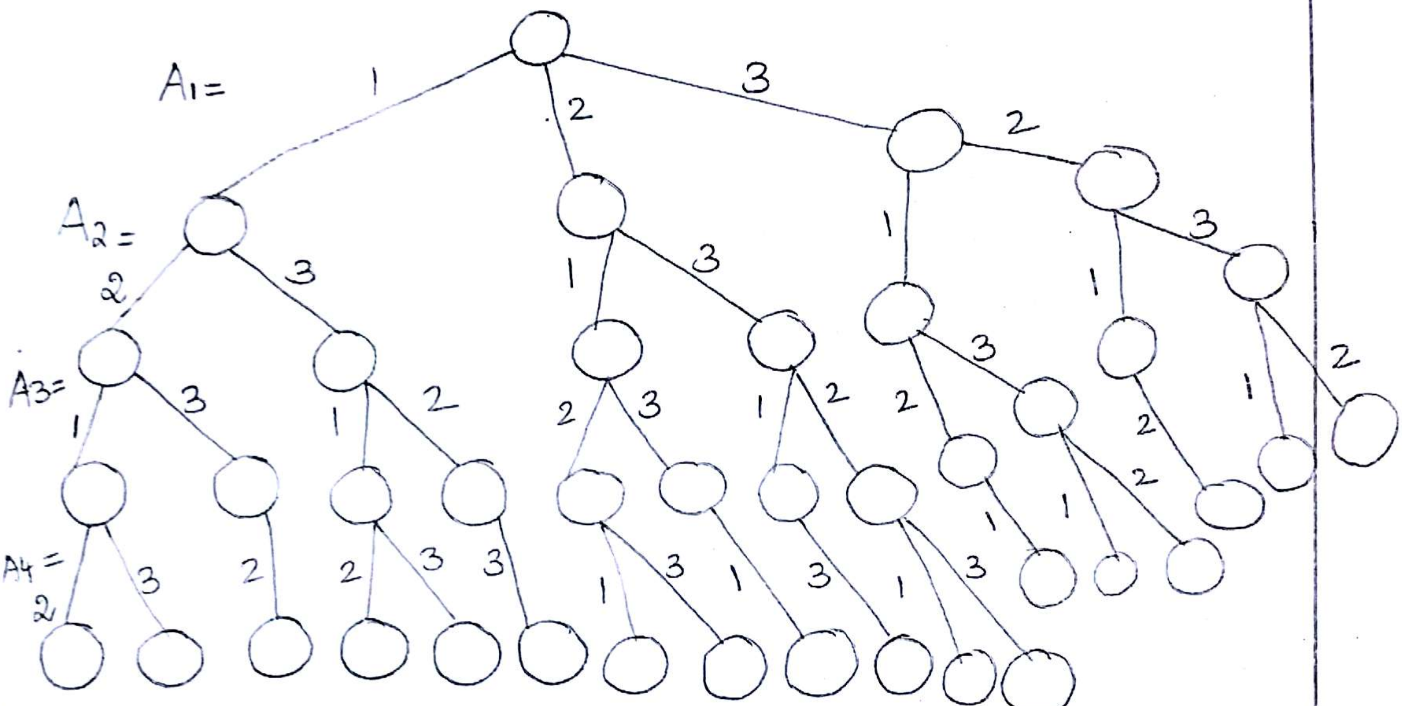
Pblm-Solution Given, G $V = \{A_1, A_2, A_3, A_4\}$

Assume, set of colours = {Red, Green, Blue}

Number of colours = 3.

Vertex	Red	Green	Blue	
A_1	✓	×	×	A_1 = Red
A_2	×	✓	×	A_2 = Green
A_3	✓	×	×	A_3 = Red
A_4	×	×	✓	A_4 = Blue

State space tree:



Complexity analysis:

Number of nodes in a state space tree $= 1 + M + M^2 + \cdots + M^n = \frac{M^{n+1} - 1}{M-1}$.

Time required for next value (colour) $= O(mn)$

Time required for m colouring $= \sum_{i=1}^{n} m^i n = \frac{n(m^{n+1} - 2)}{m-1}$

$= O(nm^n)$

Scanned by CamScanner

5. Write the procedure to generate permutation and generate the permutation for a) 123 b) ABCD c) 1234

Solution:

⇒ Permutation is the method of obtaining all possible arrangements of N items.

⇒ For N elements N! permutations are possible.

```
Algorithm permutate(i)
i/p: Input item A with n element A[1...n]
o/p: List of permutation.
Begin  if(i==n) then
          Display A[1...n]
       else
          for j=1 to n do
             A[j] ↔ A[i]
             Permutate(i+1)
             A[j] ↔ A[i]
          End for
       End if
End.
```

Complexity:

If $n=1$, no permutations are required.

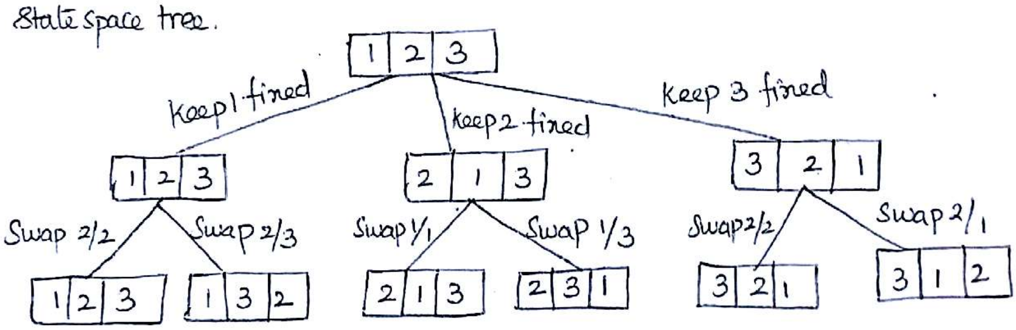
$n \geq 2$, the recurrence equation for permutation is,

$$T(n) = \begin{cases} 0 & \text{if } n=0 \\ nt_{n-1} & \text{for } n \geq 2. \end{cases}$$

∴ Solving this equation, complexity of the algorithm is $\Theta(n\,n!)$.

Prom-Solution: permutation of elements {1,2,3} where $N=3$, $3!=6$ permutations are possible. Let us fix the first value of a permutation, say 1; now, there are N-1 ways of arranging the second item and N-2 ways for the third item.

State space tree.



Possible permutation = {123, 132, 213, 231, 321, 312}