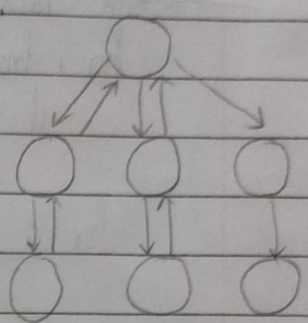
AA Mod 4

- **Backtracking** DFS

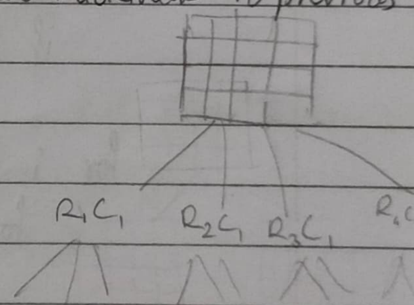
Finding solution incrementally by trying different options and undoing them if they lead to a dead end (the algo back tracks to previous decision point and explores a different path until a sol^n is found or all possibilities have been exhausted.

i) choose initial solution
ii) Explore all possible outcomes / extensions
iii) If an extension leads to soln, return it.
iv) if an extension doesn't lead to a soln, backtrack to previous soln. and try another
v) Repeat (ii) to (iv) untill all solns. are explored.



- **N-Queens**:-

In chess, queen can move straight in row, column, and diagonal. Here, we place the queens in such a way that they don't kill each other. Place queens successively in columns beginning from left and moving from top to bottom. If its not possible to place a queen in a column, we backtrack to previous column and move the queen down.



function solveNQueens(n): Main function
    result = [] initialize empty list
initialize board = empty 2D array of size n x n
call Utility fn. solveNQueensUtil(board, 0, n, result) for backtracking
    return result.
state of board | current column | size of board | list of solutions

function solveNQueensUtil(board, col, n, result): recursive backtracking
explores all possible combinations of queen placements
    if col == n: all queens have been placed
        result.append(copy(board)) copies board to result.
        return

Page No.
Date

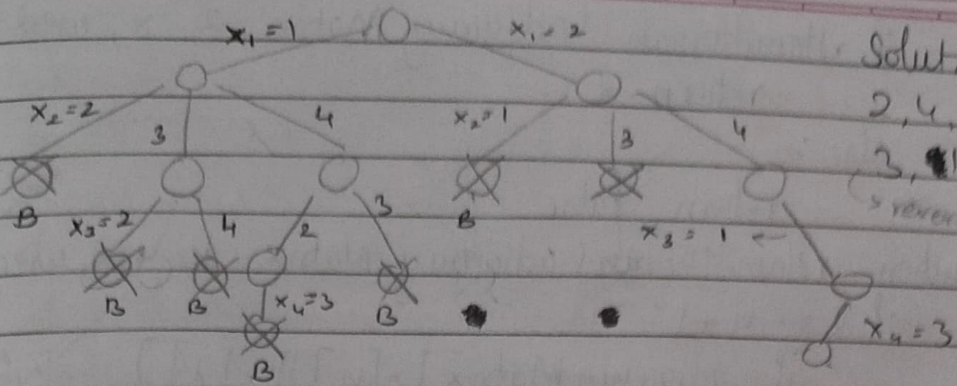
iterates from each row in current column (col)

```
for row from 0 to n-1:
    if isSafe (board, row, col, n):          checks if its safe to place a queen in that position
        board [row] [col] = 1                if its safe, it marks the position with 1
        solve NQueens Util (board, col+1, n, result)     calls itself recursively for next column
        board [row] [col] = 0                unmarks position '0' before backtracking
function isSafe (board, row, col, n):
    for i from 0 to col-1:
        if board [row] [i] == 1:
            return false
    for i, j in zip (range (row-1, -1, -1), range (col-1, -1, -1)):
        if board [i] [j] == 1
            return false
    for i, j in zip (range (row+1, n), range (col-1, -1, -1)):
        if board [i] [j] == 1:
            return false
    return true
```

This is mostly Python based IFs. Fixed version up ahead

Fixed:-

```
function isSafe (board, row, col, n):
    for i from 0 to col-1:                   check if queen is in same row
        if board [row] [i] == 1:
            return false                     conflict
    for i from row-1 down to 0:              check upper-left diagonal
        for j from col-1 down to 0:
            if board [i] [j] == 1:
                return false                 Conflict
    for i from row+1 to n-1:                 check lower-left diagonal.
        for j from col-1 down to 0:
            if board [i] [j] == 1:
                return false                 Conflict
    return true.                             none violated above
```



Solution
2, 4, 1, 3
3, 1, 4, 2
(reversed)

$T(n) = n\,T(n-1) + n$

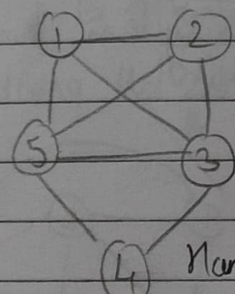
Time Complexity :-
Best : $O(n!)$: Worst : Avg
Space : $O(n)$

→ DFS

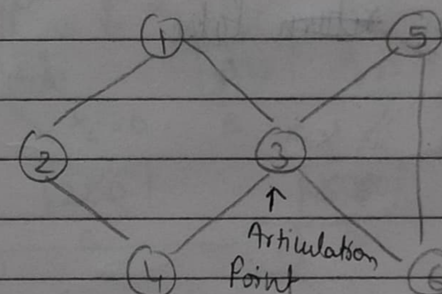
- <u>Hamiltonian Circuit</u> NP Complete Problem

Cycle that visits every vertex of G exactly once and returns to the starting vertex (G → graph G)

In this we have to show all the Hamiltonian paths.
If there's a path '1, 2, 3, 4, 1',
then we can't have other paths like '2, 3, 4, 1, 2'



Hamiltonian



Not Hamiltonian

function hamiltonianCycle(adjacency Matrix):
    n = size of adjacencyMatrix
    x = array of size n+1   store Hamiltonian cycle
    used = array of size n+1   keep track of visited vertices
    for i from 1 to n:
        used [i] = false   indicate that no arrays have been used
    x[1] = 1   cycle starts from here
    used [1] = true and marks it as visited

Page No.
Date

recursive function is called

starting from 2^{nd} vertex

if rHamiltonian (adjacency Matrix, 2, x, used, n):

passing initial state

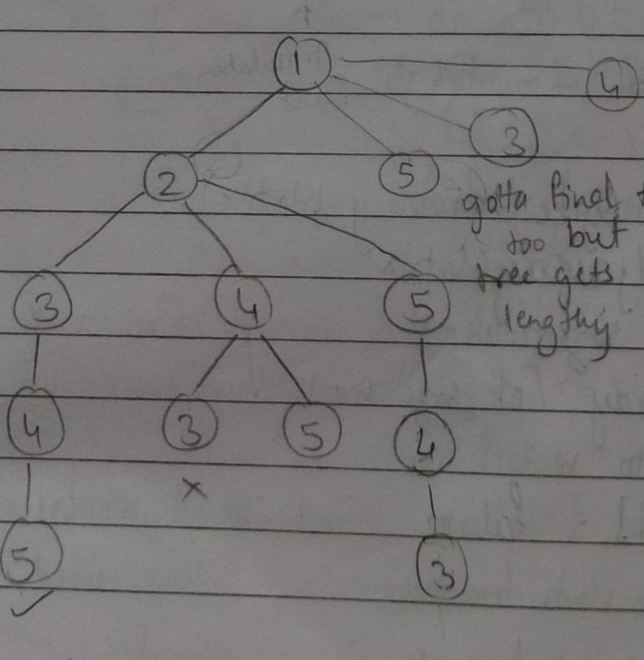
    return x
else:
    return false

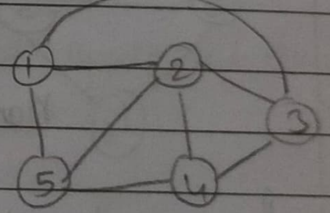
function rHamiltonian (adjacency Matrix, k, x, used, n):
    if k == n+1: all vertices have been visited
        if adjacency Matrix [x[n]][x[1]] check if there's an edge from last vertex to first to complete cycle
            return true
        else:
            return false
    for v from 2 ~~to~~ to n : iterates over vertices v from 2 to n
        if not used [v] and adjacency Matrix [x[k-1]][v]:
            x[k] = v
            used [v] = true and marked as visited
            if rHamiltonian (adjacency Matrix, k+1, x, used, n):
                return true and the function is recursively called to extend the cycle further
            used [v] = false
            x[k] = 0
    return ~~return~~ false if no Hamiltonian cycle is found after exhausting all possibilities

vertices

considering not visited & connected to previous vertex x[k-1], it is added to cycle

if no valid vertex is found for k, it backtracks by unmarking vertex as visited and resets its value in x





gotta find these too but tree gets lengthy

	1	2	3	4	5
1	0	1	1	0	1
2	1	0	1	1	1
3	1	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Solutions :- 1, 2, 3, 4, 5, 1 and 1, 2, 5, 4, 3, 1

Time Complexity : $O(N!)$ ~ no. of vertices

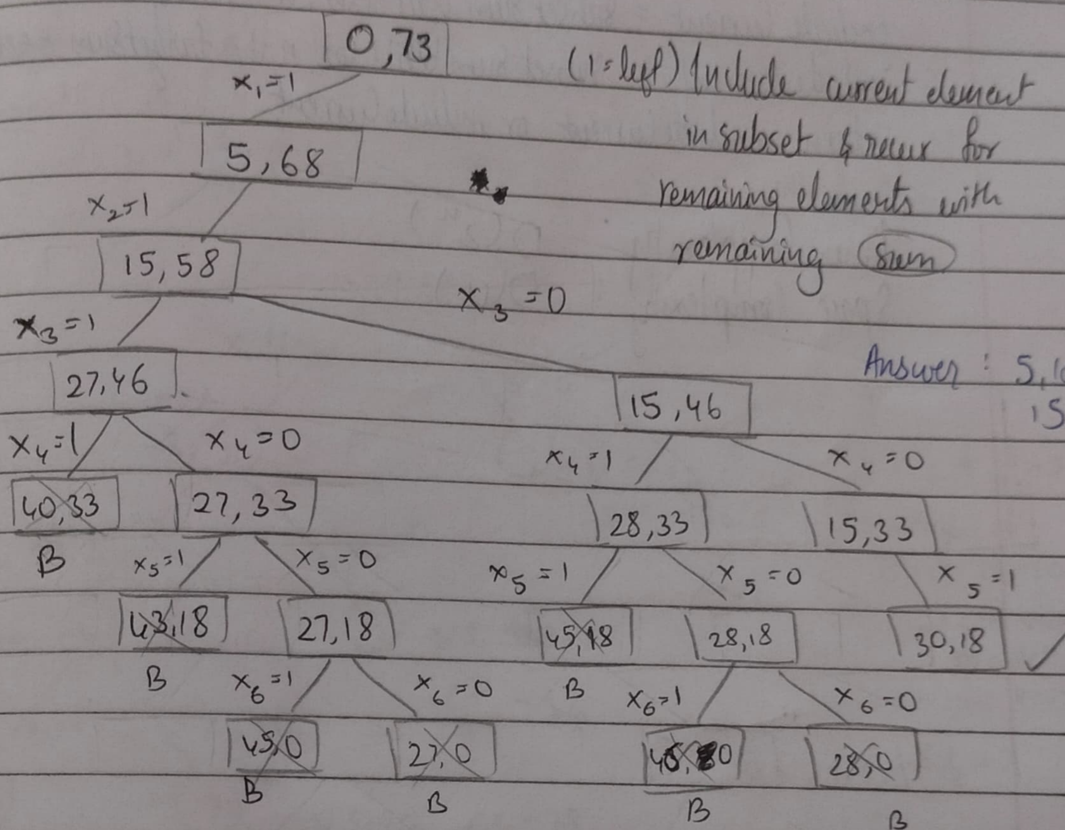
Space Complexity : $O(1)$ No extra space used.

Sum of Subsets

Set [ ] of non-negative values is given along with a sum, solution should be subset whose sum is ≈ given sum

$w[1:6] = \{5, 10, 12, 13, 15, 18\}$ $n = 6$ $m = 30$

(1 = left) Include current element in subset & recur for remaining elements with remaining sum



Answer : 5, 10, 15

(0 = right) Exclude current element from subset and recur for the remaining elements.

If the sum becomes 0, print elements of current subset.

~~function subsetSum(num, targetSum).~~
~~n = size of nums~~
~~dp = 2D array of size~~

Page No.
Date

```
function subsetSum (set, targetSum):
    n = size of set
    return subsetSum (set, n, targetSum)
function subsetSum (set, n, targetSum):
    if targetSum == 0:          subset with sum = Target Sum has
        return true             been found, so true is returned
    if n == 0 or targetSum < 0: no subset is possible for
        return false            sum = Target Sum, so, false
    excludeCurrent = subsetSumUtil (set, n-1, targetSum)
    includeCurrent = subsetSumUtil (set, n-1, targetSum - set[n-1])
    return excludeCurrent or includeCurrent
```

Time Complexity : $O(2^n)$
Space Complexity : $O(n)$