Backtracking :-

↳ We have a given problem and if we have to check all the possible combination to solve that problem but in optimised way then we use backtracking.

↳ Question arises what we meant by optimised way? (Why!)
⇒ Rather than checking all permutation or combination we will focus only on some few combinations. this will optimsed our code in terms of time complexity.

↳ Simple lang:-
Backtracking is a depth-first search with bounding function.

↳ We use. Recursion Here to build our solution incrementally. one piece at a time and removing those solution that fails to satisfy the contraint.

Example

Rat in a Maze Problem :-

(V.Imp)

Movement :- Forward and downward

Start → | 1 | 0 | 0 | 0 |
1	1	0	1	0
0	1	1	1	0
0	0	0	1	0
1	1	1	1	1	→ end.

Soln :-

1	0	0	0	0
1	1	0	0	0
0	1	1	1	0
0	0	0	1	0
0	0	0	1	1

↳ In Soln we consider only that place which is necessary. means सभी जगह '1' लिखने के जगह सिर्फ वहाँ 1 लिखे जहाँ rat जाए तो उसके लिए easy पड़े।

↳ Question में recursion इस बात पर निर्भर करेगा कि movement कैसा हो रहा है। ऊपर वाले Ques में only forward and downward हो रहा है तो recusion भी दो ही case के लिए होगा।

Code Part :-

P.T.O ⇒

Step-I ↳ **Main method** :- Here we have to declare the maze on which we have to find the solution.
↳ after then we have to call the Solve maze function by passing the maze.

```
if __name__ == "__main__":
    maze = [ [1, 0, 1, 0]
             [1, 1, 1, 1]
             [0, 1, 0, 1]
             [1, 1, 0, 1] ]

    Solve maze (maze)
```

declaring the maze 2-D array or we can say this Adjacency matrix.

Step-II. We have to declare the size of matrix this can be Static or dynamic :

यह static है

```
For Static   N = 4
For dynamic  row = len(maze)
for dynamic  col = len(maze[0])
```

यह array से ही data fetch कर लेता है कि कितना Row है कितना colum है।

Step-III We have to create a function named "Safe" Just to check that the move is happen within the maze not outside of maze. and the rat is allowed to move. on the next maze.

```
def Safe (maze, x, y):
    if( x>=0 and x<N and y>=0 and y<N and
                maze [x] [y] = 1):
        return True
    return False
```

यह function यहा पे check करेगा कि rat का move valid है कि नही
↳ वो maze के अंदर होना चाहिए
↳ move करने के लिए 1 होना चाहिए।

Step-IV Backbone of the algo:

Here we have two funtion named 'Solvemaze' and Solvemazeutil

a) Solvemaze:

↳ It work is just to create an Soln 2D list on which ans is reflected

↳ It will also detect whether there is any solution or not. And if there is Soln then it call the display function which will print all the solution.

```
def Solvemaze(maze):
    Sol = [ [0 For i in range(N)] For j in range(N)]
    if (Solvemazeutil(maze,0,0,Sol) == False):
        Print("Solution Does not exists")
        return False.
    else:
        PrintSol(Sol)
    return True.
```

यहाँ से Solvemaze util function को 0,0 index से start करवा रहा है क्योंकि ये कुछ fun recursion पर based है इसलिए ये recursively call होकर last में return करेगा value. base इसका base case (N-1, N-1) पर जाकर रुकेगा होगा और तभी से True return करेगा अगर base case तक नहीं पहुँचा है तब return होगा।

b)

```
Solvemazeutil(maze, x, y, Sol):
    if(x == N-1 and y == N-1 and maze[x][y] = 1):
        Sol[x][y] = 1   ← ये destination है mean the lower right part.
        return true:
    if(Safe(maze, x, y, Sol)=True):   ← यहाँ हम move को check कर रहे है अगर true तभी आगे जाएगा
        Sol[x][y] = 1
        if(Solvemazeutil(maze, x+1, y, Sol) == True:   ← यहाँ recursion हो रहा है
            return True,
        if(Solvemazeutil(maze, x, y+1, Sol) == True:
            return True.
        Sol[x][y] = "."   # ← This is V.Imp step backtracking यदि है रहा है तो जो Path का हिस्सा नहीं उसे वैल्यू कर दे रहा है।
        return false.
```

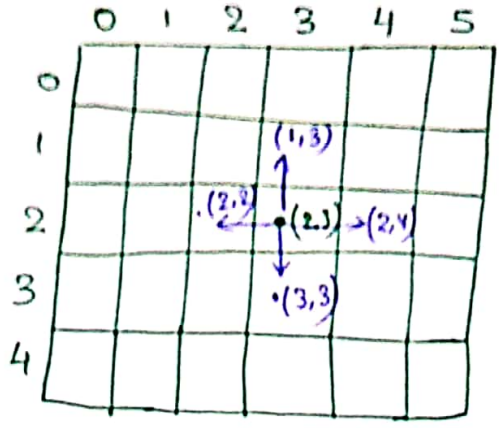
Trick to find Recursion increment Strategy for (x, y)

x y

Suppose u are in (2, 3) Pos

For

	up	down	left	right
x =	-1	+1	x	x
y =	x	x	-1	+1



If mean

- If (Solve maze Util (maze, x+1, y, Sol) == True) # downward
  return True
- If (Solve maze Util (maze, x-1, y, Sol) == True) # upward
  return True.
- If (Solve maze Util (maze, x, y+1, Sol) == True) # Right direction
  return True
- If (Solve maze Util (maze, x, y-1, sol) == True) # left direction
  return True.

These two are used in Rat - maze Problem

Implementation of Rat maze Problem.

N = 4

```
def Safe(maze, x, y):
    if(x >= 0 and x < N-1 and y >= 0 and y < N-1 maze[x][y] = 1):
        return True.
    else:
        return False

def SolveMaze(maze):
    Sol = [["." for j in range(N)] for i in range(N)]

    if (SolvemazeUtil(maze, 0, 0, Sol) == False):
        Print("Solution does not exist").
    else:
        PrintSol(Sol)
    return true!
```

क्योंकि यह recursive fun है तो पहले baseCase design करो।

```
def SolvemazeUtil(maze, x, y, Sol):
    if(x == N-1 and y == N-1 and maze[x][y] == 1):
        Sol[x][y] = 1
        return True.

    if(Safe(maze, x, y) = True):
        Sol[x][y] = 1
        return True.
        if(SolvemazeUtil(maze, x+1, y, Sol) == True:
            return True
        if(SolvemazeUtil(maze, x, y+1, Sol) == True:
            return True,
        Sol[x][y] = "."
        return false.

if __name__ == '__main__':
    maze = [ [1,0,1,0]
             [1,1,1,1]
             [0,1,0,1]
             [1,1,0,1]]

    SolveMaze(maze)
```

* Now the Scenario is that the rat can able to move in all direction Code it:-

#2 The Knight Tour Problem (Backtracking)

Book Page

Given :
- ↳ N * N board with Knight placed in 1st block.
- ↳ Move the Knight according to chess rule so that it can visit all the chess board squares.
- ↳ Print order of each cell in which they are visited.

Step 1 Main method ~~Yet~~ Here we have to take input from the user (the size of chess board Ideal : 8) and call the Solve function.

```
if __name__ = '__main__':
    n = int(input("Enter the size of chess : "))
    Solve().
```

Step 2 After that we have to create the basic Safe function which check the validity of the horse move.

```
def Safe(board, x, y):
    if(x>=0 and x<n and y>=0 and y<n
            and board[x][y]==-1):
        return True.
    else:
        return False:
```

← only check the validy of move. Helps in backtracking.

Step 3 A function that used to print the movement of Knight in the board. Simply used 2 for loop to print the 2-D array data.

```
def printSol(n, board):
    for i in range(n):
        for j in range(n):
            print(board[i][j], end=" ")
        print(" ")
```

Step 4. Backbone of the algorithm.

↳ Here we have to create the Solve() and SolveUtil function.

↳ a) **Solve() function:-**

↳ Here we create the 2D board array in which the movement of knight is stored.

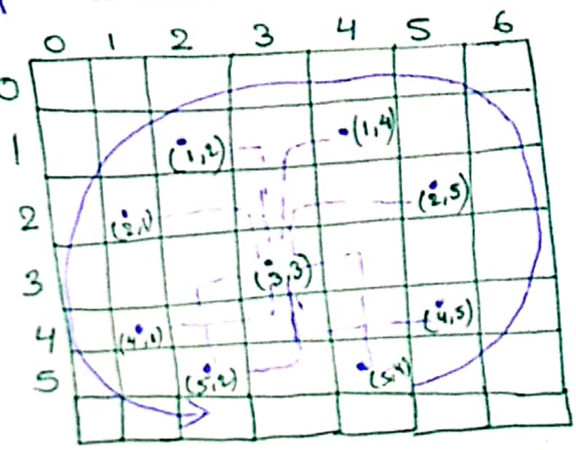
↳ Here we have to create 2 array that store all the possible move of the knight.

एक arbitrary point चुन लो

(Trick) **Note:** direction का ध्यान रखें नहीं तो $8^8 = 1$ crore loop चलेगा

x_arr =	2	1	-1	-2	-2	-1	1	2
y_arr =	1	2	2	1	-1	-2	-2	-1

✓✓ Correct form.



Then we have to set the initial pos of knight that is (0,0) and create a counter pos that store the record of knight to move.

```
def Solve():
    board = [[-1 for i in range(n)] for j in range(n)]
    x_arr = [2, 1, -1, -2, -2, -1, 1, 2]
    y_arr = [1, 2, 2, 1, -1, -2, -2, -1]

    board[0][0] = 0
    pos = 1

    if (SolveUtil(board, 0, 0, x_arr, y_arr, pos) == False)
        Print("Not Possible Sol^n")

    else:
        PrintSol(n, board)
```

Scanned with CamScanner

b) **Solvemazeutil Function :-**

↳ यहाँ पर recursive function है तो पहले base case design करेंगे.

↳ Then run a loop which goes from 0 to 8 whyso because we use the (curr pos + the move_arr pos) to determine all the possible path of knight.

↳ Every iteration we get new co-ordinates of knight by Summing current pos + pos in array then we simply check whether this move is valid if valid then add to the board 2-D array.

↳ Here instead of boolean flag we use a counter which count knight step. After that we again call the Solve util function.

↳ Need of Backtracking in this question is :

- ↳ It only used in last few steps.
- ↳ There are possibilites that the knight visit the same place.
- ↳ If this happen and we didn't use backtrack then redundant moves will comes in board.
- ↳ In order to neglect this whenever the knight visit the same place the safe function will fail.
- ↳ If it fails then solveUtil function also get fail in its just one previous one.
- ↳ which result in again making the visited place marked as -1 or unvisited

```
def SolveUtil ( board , x, y, x_arr, y_arr, pos):
    if (pos == n*n):
        return True
    For i in range (0,8):
        new_x = x + x_arr[i]
        new_y = y + y_arr[i]
        if (board, new_x, new_y) == True:
            board[new_x, new_y] = pos
            if (SolveUtil (board, new_x, new_y, x_arr, y_arr, pos+1)==True:
                return True
            else:
                board[new_x, new_y] == -1
```

Backtracking

* Implementation of Knight tour Problem:-

n = 8

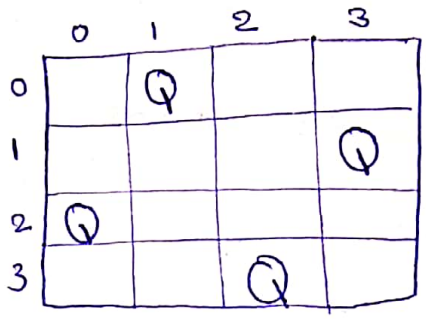
```
def Safe (Board, x, y):
    if (x >= 0 and y >= 0 and x < n and y < n and board[x][y] == -1)
        return True.
    else
        return False.

def PrintSol (n, board)
    For i in range (n)
        For j in range (n)
            Print (board[i][j], end = " ")
        Print.

def Solve ():
    board = [[-1 For i in range (n)] For j in range]
    x_arr = [2, 1, -1, -2, -2, -1, 1, 2]
    y_arr = [1, 2, 2, 1, -1, -2, -2, -1]
    board [0][0] = 0
    pos = 1
    if (SolveUtil (board, 0, 0, x_arr, y_arr, pos) == False)
        Print ("Sol^n is not Possible")
    else PrintSol (n, board).

def SolveUtil (board, x, y, x_arr, y_arr, pos):
    if (pos == n*n):
        return True.
    For i in range (0, 8)
        new_x = x + x_arr[i]
        new_y = y + y_arr[i]
        if (Safe (board, new_x, new_y) == True):
            board [new_x][new_y] = pos
            if (SolveUtil (board, new_x, new_y, x_arr, y_arr, pos+1) == True):
                return True.
            else board [new_x][new_y] = -1   <-- backtracking

if __name__ == '__main__'
    n = int (input ("Enter the Size of Chess"))
    Solve().
```

(3)

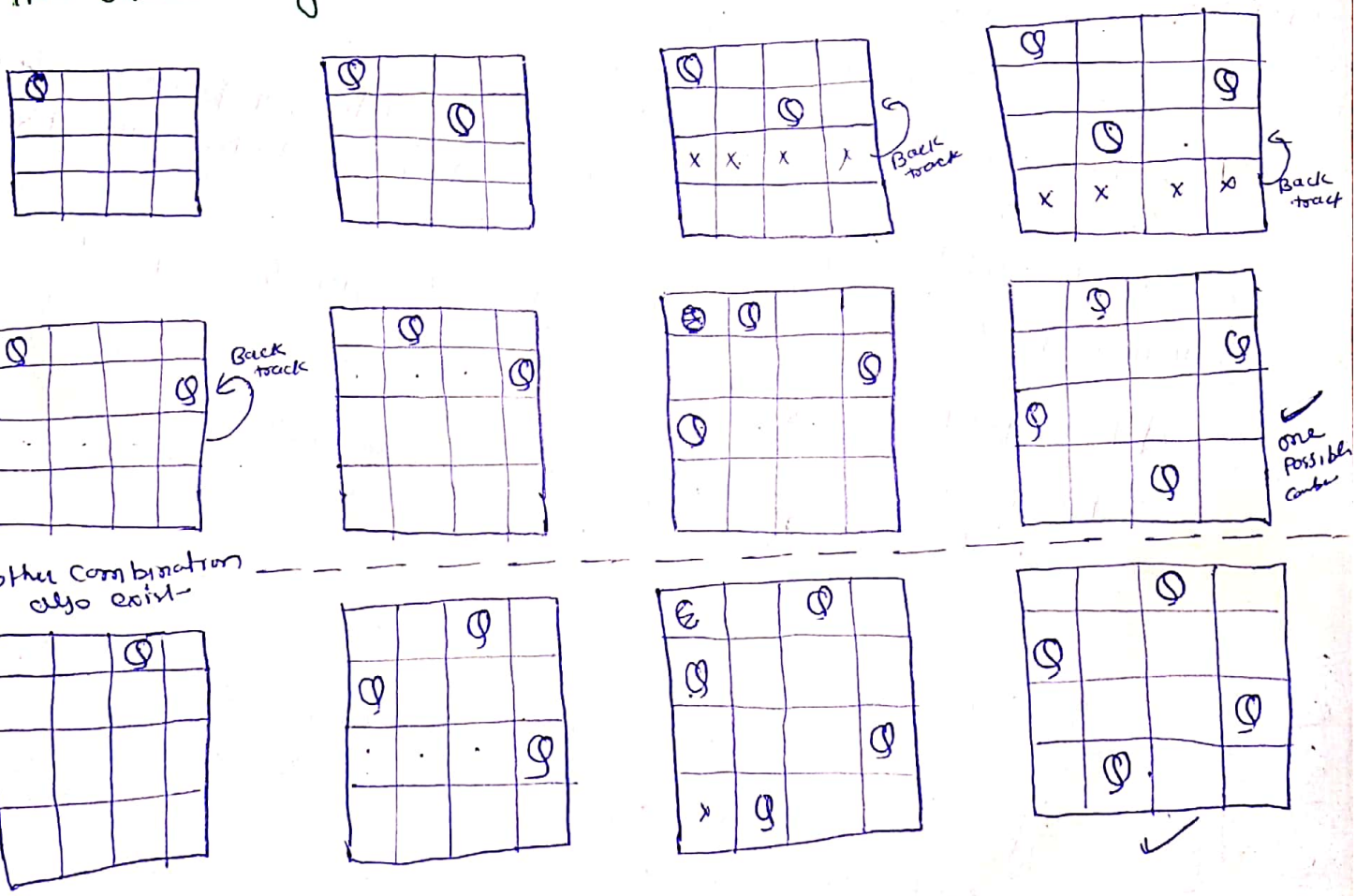
The 'N'-Queen Problem :-

* Given :- ↳ U have Given N x N board with N Queen.
↳ Your task is to assign the N Queen in such a way So that it can't attack each other.

e.g. :- ↳ check rowwise, colloumwise, diagonal
↳ Here no queen attack any other queen.



How back tracking is evolved in it :-



Step-1 Main method: In main method we simply ask the user that what is the size of the chessboard. (mean N×N board having N Queen मैं 'N' का Value user से input लेना है).

```
def __name__ = '__main__':
    n = int(input("Enter the Value of the N"))
    Solve().
```

Step-2 **Backbone of Algorithm** :- "Safe function" which Check the validity of moves.

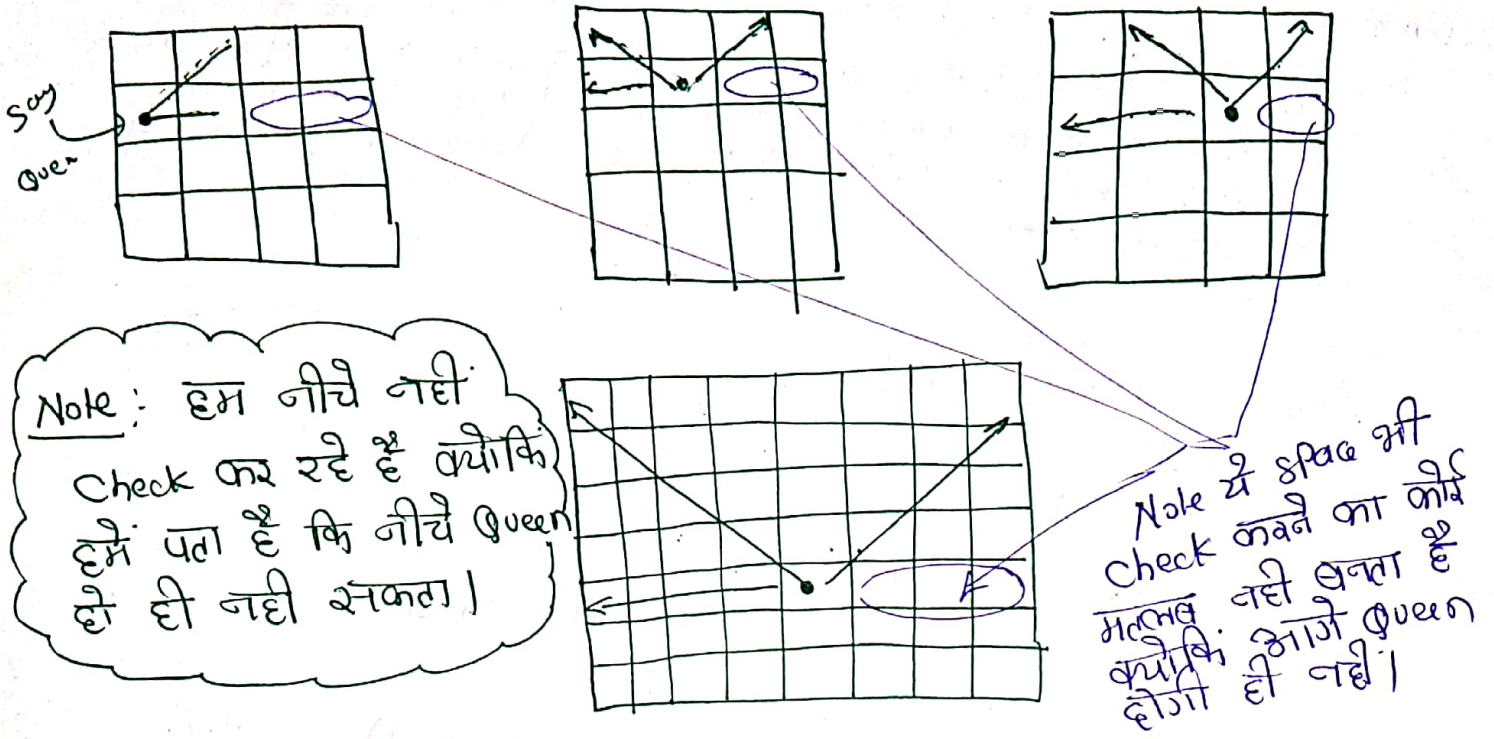
↳ इसमें ये check किया जाएगा कि किसी particular place पर जब Queen को रखते हैं तो वो दूसरे Queen द्वारा attack किया जा रहा है या नहीं।

Similar to other Safe function but check multiple case.

```
def Safe(board, r, c)
    For i in range(c):                    # यहाँ हमें c use करना है न कि n क्योंकि recursion में c increment होगा ही
        if(board[r][i]==1):               # ये coloum को check करेगा की कहीं Queen coloum से attack नहीं कर रही
            return False.

    row = r
    col = c
    For i in range(n):
        if(row>=0 and col>=0):            # ये line left upper diagonal को check कर रहा है।
            if(board[row][col]==1)
                return False.
        row = row-1
        col = col-1
    row=r
    col=c
    For i in range(n)
        if(row>n and col>=0):             # ये left upper diagonal को
            if(board[row][col]==1):       # check कर रहा है।
                return False;
        row = row+1
        col = col-1
    return True.
```

Use while loop Not optimized

See the outline next



Note: हम नीचे नहीं Check कर रहे हैं क्योंकि हमें पता है कि नीचे Queen हो ही नहीं सकता।

Note ये space भी Check करने का कोई मतलब नहीं बनता है क्योंकि आगे Queen होगी ही नहीं।

Step-3 The Solve and Solve util function

a) Solve :- यहाँ पर simple सा board generate करेंगे then हम check करेंगे Solve util function काम कर रहा है कि नहीं अगर कर रहा है तो Print करवा देंगे नहीं तो "No Solution Print".

```
def Solve():
    board = [[0 for i in range(n)] for j in range(n)]
    if (SolveUtil(board, 0) == False):
        print("Solution does not exist");
    else:
        PrintSol(board).
```

Step-3 (b) => SolveUtil function :-

↳ ये function एक row लेगा फिर उस row के सारे coloum में recursion चलाकर moves की validity check करेगा। अगर कहीं भी false आएगा उस time से backtrack भी करवाएगा।

```
def SolveUtil ( board, coloum):
    if (coloum >= n):
        return True.

    For i in range (n):
        if ( safe (board, i, coloum) == True):
            board [i] [coloum] = 1
            if (SolveUtil (board, coloum +1) == True):
                return True.
            board [i] [coloum] = 0

    return False.
```

* Implementation of N-Queen Problem (Backtracking)

Code :-

```
def Safe(board, r, c):
    For i in range(c):
        if (board[r][i]==1)
            return false.
    row = r
    col = c
    while (row>=0 and col>=0):
        if (board[row][col]==1)
            return false.
        row -= 1
        col -= 1
    row = r
    col = c
    while (row<n and col>=0):
        if (board[row][col]==1)
            return false.
        row += 1
        col -= 1
    return True.

def Solve():
    board = [[0 For i in range(n)] for j in range(n)]
    if (SolveUtil(board, 0) == false):
        Print("Solution does not exist")
        return false.
    else:
        Print Sol(board)

def SolveUtil(board, Coloum);
    if (coloum>=n):
        return True.
    For i in range(n):
        if (Safe(board, i, coloum)==True):
            board[i][coloum]=1
            if (SolveUtil(board, coloum+1)=True:
                return True
            board[i][coloum]=0
    return false.

if --name-- == --main--:
    n = int(input("Enter the size n:"))
    Solve()
```

backtrack

यदि Recursion हो रहा है,

V.V.9 (To Print all the Solution):- V.V.9

* Implementation of N-Queen Problem: (V.V.9mP)

Code

```
def Safe(board, r, c):
    For i in range(c)
        if(board[r][i]==1):
            return False.

    row = r
    col = c
    while(row>=0 and col>=0):
        if(board[row][col]==1)
            return False.
        row = row-1
        col = col-1

    row = r
    col = c
    while(row<n and col>=0):
        if(board[row][col]==1):
            return False.

        row = row+1
        col = col-1
    return True.

def Solve():
    board [[0 for i in range(n)] for j in range(n)]
        if(Solveutil(board,0)==False):
            Print("Solution does not exit")

def Solveutil(board, column):
    if(column>=n):
        Printsol(board)
        return True.
    res = False.
    For i in range(n):
        if(Safe(board, i, column)==True):
            board[i][column] = 1
            res = Solveutil(board, column+1) or res
            board[i][column] = 0
    return res.

if __name__ == '__main__'
    n = int(input("Enter the Size of board"))
    Solve().
```

enhanced code from Previous one.

* Word break problem using backtracking :-

Given :- ① A dictionary which contain a number of words.
② A String without space.

Task : * To find all possible sentences using the dictionary and String.
* The sentence are resembles with the string.

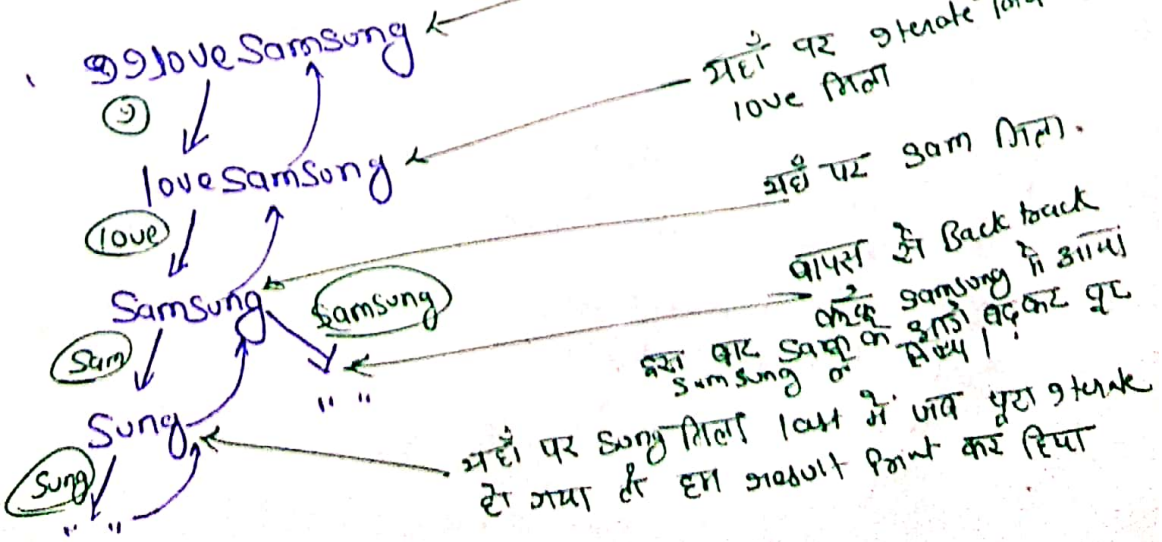
e.g: dict = ['I', 'love', 'Sam', 'Sung', 'SamSung']

Str = 'IloveSamsung'

Output : I love Sam Sung ← ये दोनों आना चाहिए।
I love SamSung.

Logic :
① हम एक loop चला रहे हैं जो Str से 1 letter लेगा उसको dict में check करेगा।
② अगर letter वहीं है तो loop फिर से iterate होगा और दूसरा letter भी जोड़ देगा। फिर जोड़ने के बाद Check करेंगे कि वो dict में है कि नहीं।
③ अगर कोई ऐसा cond^n हो जिससे वो dict में मौजूद हो तो हम एक अन्य variable say result में उसको Concat करवाते जाएंगे।
④ अब क्योंकि हमें Possible combination find करना है तो Simple loop is not sufficient we have to use the recursion (back backing).

Soln :



* Implementation of word-break using backtracking :-

```
def Solve ( str, res):
    For i in range (1, len(str)+1):
        Prefix = str [0 : i]
        if (Prefix in dict):
            Solve ( str [i:], res + Prefix + " ")
            if ( i == len(str)):
                res += Prefix
                Print (res)
                return True.
```

← यहाँ 1 से चलेगा क्योंकि slicing के लिए compulsory है)

```
if __name__ = '__main__':
    dict = ['I', 'Sam', 'love', 'Samsung', 'Sung']
    str = 'IloveSamSung'.
    Solve (dict, res)
```

Output आना चाहिए :

I love Sam Sung.
I love SamSung.

* Find Permutation of Strings using Backtracking :-

Given: A String is given and asked all the possible Permutation that can be formed by changing the character of that String.

Note : The permutation must be unique means no repetition of Same permutated String twice.

logic :-

① सबसे पहले दिये हुए String को हम array में convert करेंगे।

```
Strn= "AABC"
Str = list(Strn)  ⟶  ['A','A','B','C']
```

② हम Hast table का help लेंगे और सारे letter को frequency wise Seperate कर लेंगे।

```
dict= {}
For i in Str:
    If i not in dict:
        dict[i] = 1
    else
        dict[i] += dict[i]   ⟶  {'A':2, 'B':1, 'C':1}
```

③ इसके बाद हमें दो array की जरूरत पड़ेगी ✱ जो data ['A','B','C'] और उसका क्रमशः frequency [2,1,1] को Store कर सके

Note : क्रमशः data को उसके frequency के अनुसार Store करना जरूरी है

```
data=[], count=[]
dict = {'A':2, 'B':1, 'C':1}

For key in dict.items():
    data.append(d)   ⟶  ['A','B','C']
    count.append(key)  ⟶  [2,1,1]
```

④ एक अन्य array भी बनाना है जो as a buffer जैसा काम करेगा और result को Store करके बताएगा

```
res = [0] * len(Str)   ← Note: हमें index का काम होगा इसलिए इसे by-default '0' initialize करना जरूरी है।
```

↳ res = [0,0,0,0]

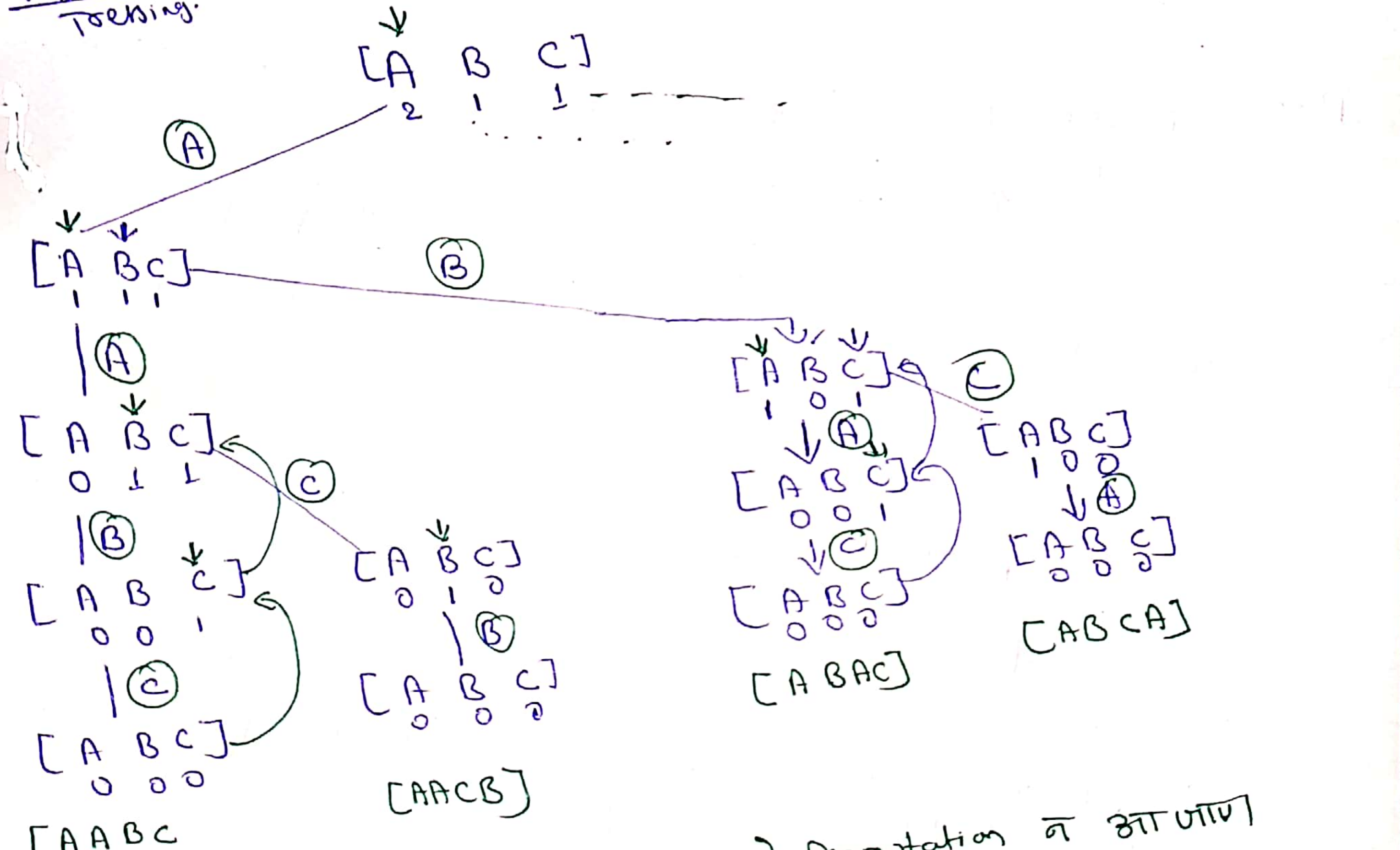
Main funda :

Permutation of String करने के लिए हमें दो array की जरूरत पड़ेगी :

① 1st array जो unique data को hold करे
② 2nd array जो क्रमशः उसके frequency को hold करे

See Previous Procedure :

logical Thinking



ये Process चलता रहेगा जब तक सारे Permutation न आ जाए।

'↓' किसी भी Pointer को दिखाने के लिए use किया गया है या कह सकते है कि for loop का Iteration इससे समझा जा सकता है।

Note : क्योंकि हम Count array को बार बार update करके बदल रहे है तो हमें जब भी back track करना होगा तो simple recursion करने से पहले जिस count को हमने decrement किया था उसे increment कर देना होगा ठीक recursion को call करने के बाद यहाँ पर यही back track करेगा Just one step back.

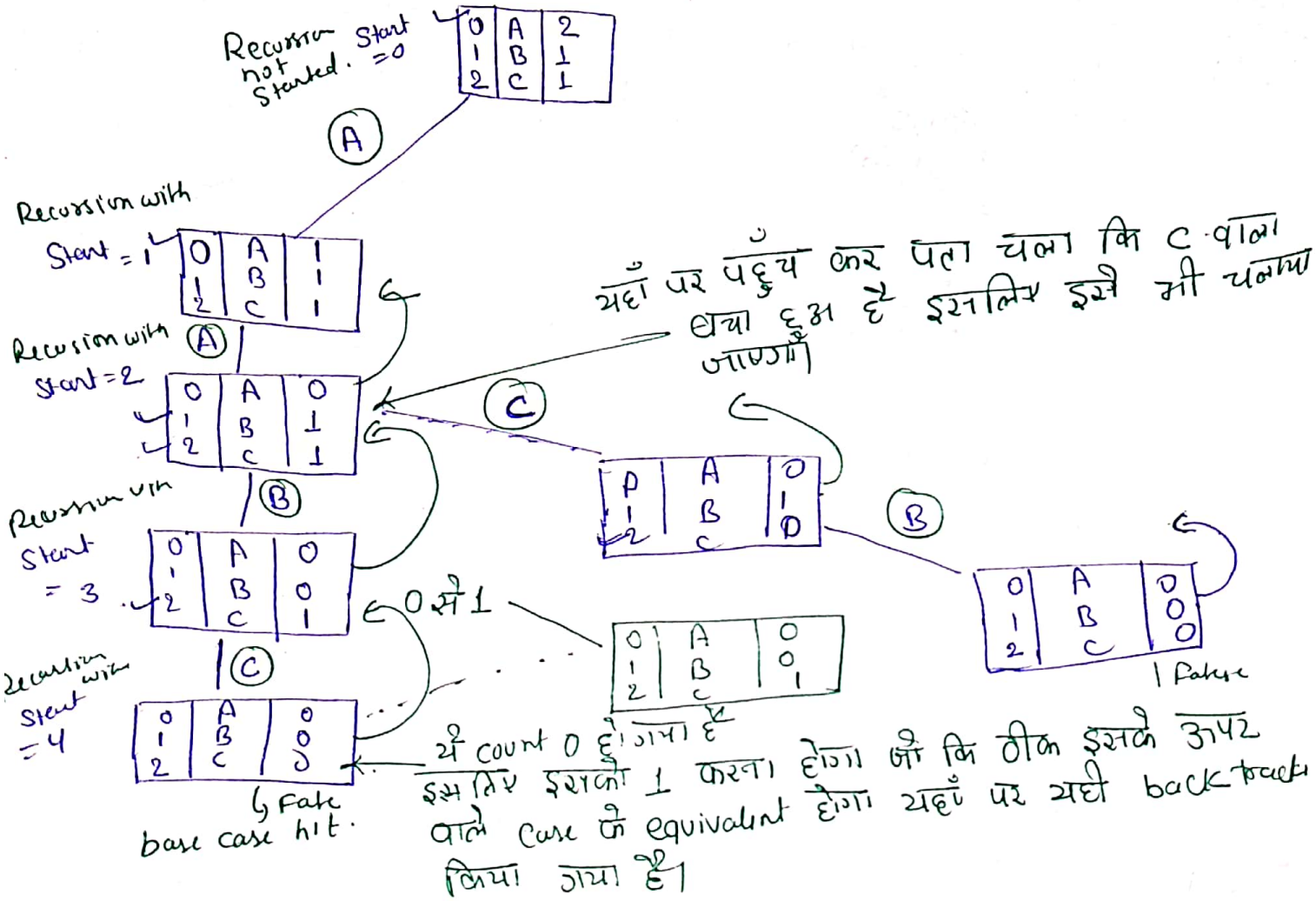
↓ ← इस arrow का मतलब की वो Phase Processing में है।

* Implementation of String using back tracking :-

```
if __name__ = '__main__':
    Strin = 'ABAC'
    Str = list(Strin)
    dict = {}

    For i in Str:
        If i not in dict:
            dict[i] = 1
        else:
            dict[i] += dict[i]

    data = []
    count = []
    For d, Key in dict.items():
        data.append(d)
        Count.append(Key)

    res = [0] * len(Str).
    Permutation(data, count, res, 0)
```

```
def Permutation(data, Count, res, start):
    If (Start == len(data))      }
        For i in res:            }  ———— Base case
            Print(i, end=" ")    }
        Print('')

    For i in range(len(data)):                     }
        res[Start] = data[i]                       }
        Count[i] = count[i] -1                     }  Backbone
        Permutatation(data, count, res, start+1)   }
        Count[i]= Count[i] + 1                     }
```

Function behind it



* Subset Sum Problem (backtracking) :-

Given : ↳ Array of Integer (arr = [2, 3, 5, 6, 8, 10]
↳ Sum (Any value) (Sum = 10)

Task : We have to find all possible Subset from the set whose Summation is equal to the Sum value.

arr = [2 , 3 , 5 , 6 , 8 , 10]

Possible Subset are :- 2, 3, 5
:- 8, 2
:- 10

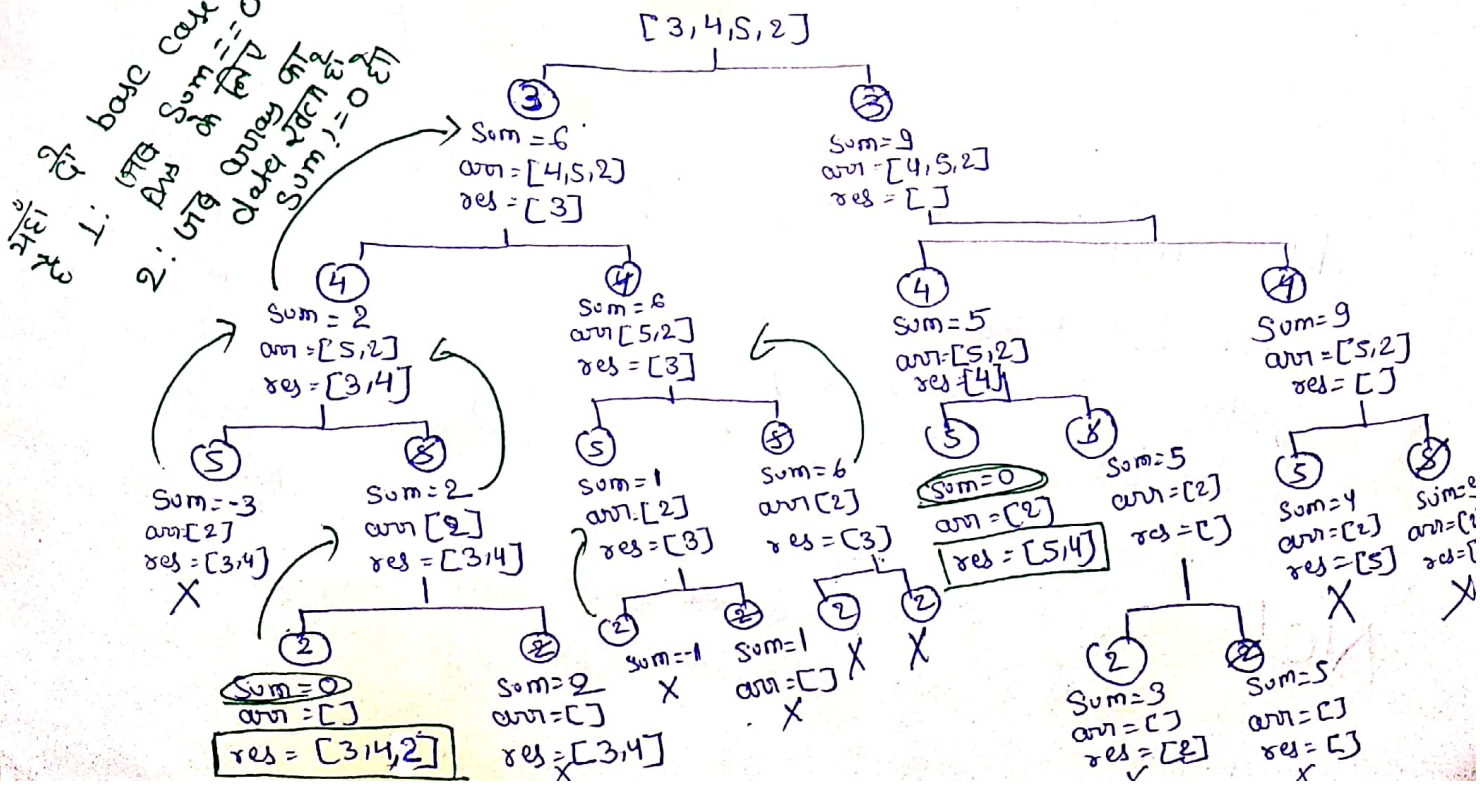
Time Complexity
$O(2^n)$ ← Set का Subset कितने होते हैं

Imp इस type के Subset वाले Problem (using recursion) करने का logic.

⇒ हमें use करना होगा Recursion.
⇒ इस Question को करने के लिए एक नया Strategy है।
Named :- **'Accept' and 'Reject' approach.**

for e.g : arr = [3,4,5,2] Sum = 9 चाहिए → [3,4,2] या [5,4]

यहाँ दो base case है
1: जब Sum == 0 के लिए
2: जब array का data खत्म हो और Sum != 0 हो



* Implementation of Sum of Subset problem in $O(2^n)$: on can be reduced using DP.

```
def Subset(arr, Sum, res, n):
    if (Sum == 0)
        Print res
        return;
    if (n == 0)
        return
Reject = Subset(arr, Sum, res, n-1)
r = res + [ ]
r.append(arr[n-1])
accept = Subset(arr, Sum - arr[n-1], r, n-1)
```

2 base case

Reject वाले section में सब चीज़ same वरन arr से एक data कम हो जाएगा

क्योंकि ये Accepted wala है तो इसमें हम modified Sum भेजेंगे, और जो resultant (res) arrays है उसमें data को include करना होगा क्योंकि ये accepted वाला है।

```
def __name__ = '__main__':
    arr = [3,4,5,2]
    sum = 10, res = [ ]
    Subset(arr, Sum, res, len(arr))
```

Note: यहाँ recursion पिछे से शुरु किया गया है। आगे से भी शुरु कर सकते है।

* Finding Subset of a Set using backtracking:-

Given:- ↳ An array (as a Set) or you have to make a set
↳ Set doesnot have any duplicate value.

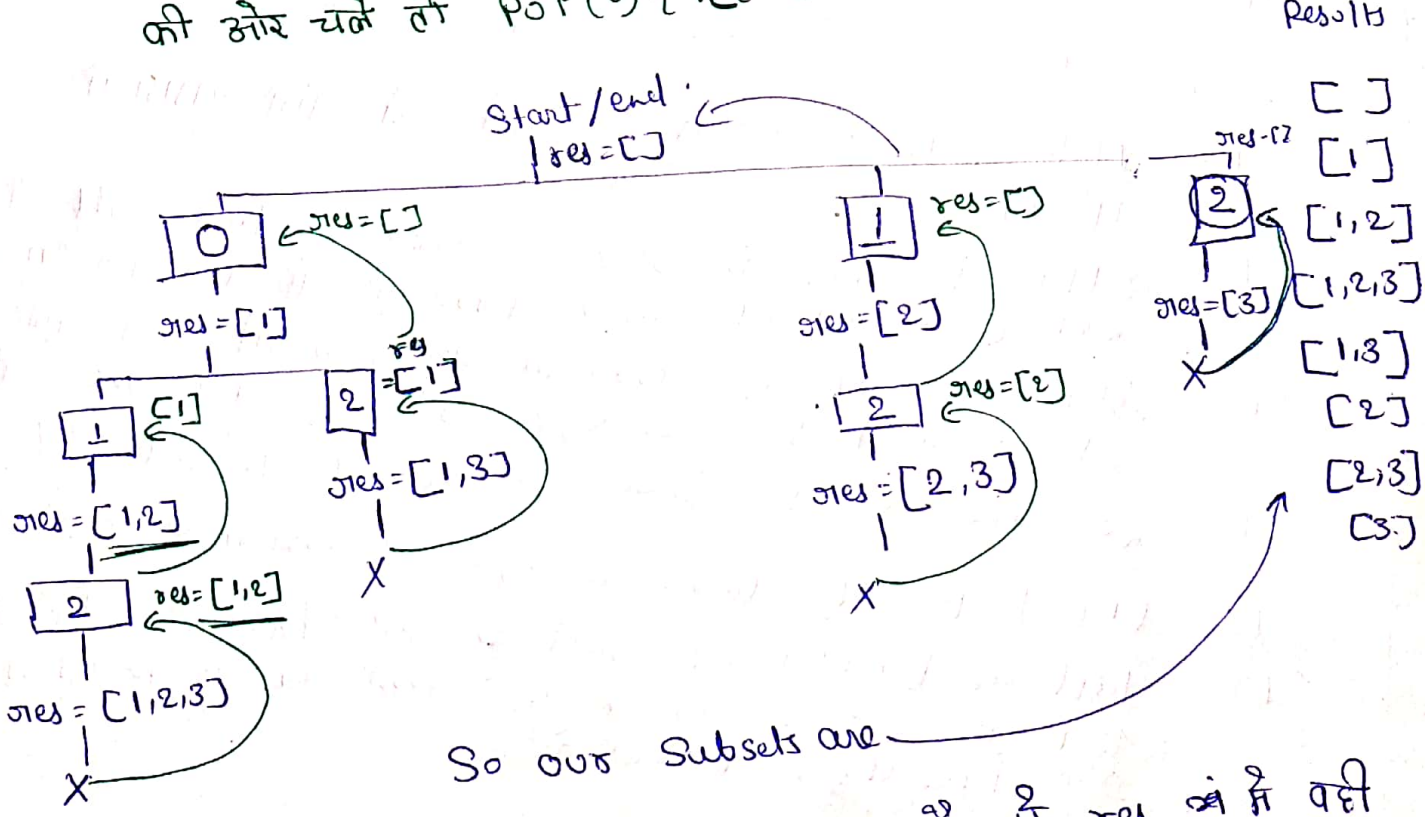
Task:- To print all the subset of Given set. Means if it has n-elements then print 2^n subsets of it.

e.g:- arr = [1,2,3] Subset = [[], [1], [2], [3], [1,2], [1,3], [2,3], [1,2,3]]

Logic :-

⇒ हम loop चलाते है और recursion का use करके back track करते है।

Note अगर loop 0 से len(arr) तक चलता है तो हमेशा last element को pop करे or ठीक इसका उल्टा len(arr) से 0 की ओर चले तो pop(0) ? पहला element को उड़ाए ?.



Clearly देखे जब backtrack हो रहा है तो res में वही data रह रहा है जो recursion रुक होने से पहले था यही backtracking है।

* Implementation to find Subset of Given Set :-

```
def SubSet (arr, res, Sub, start):
    Print (res)
    Sub . append (res + [])

    For i in range (start, len(arr)):
        res . append (arr[i])
        Subset (arr, res, Sub, start+1)
        res . Pop ()

    return
```

Question : अगर इसमें Recursion used हुआ है तो बिना base case के ये exit कैसे हो रहा है?

↳ Note इसमें base case है but इसे थोड़ा समझना पड़ेगा।

↳ अपना for loop के अंदर recursive function call हो रहा है means जब loop चलेगा तो वो recursion करेगा

↳ अब जब loop नहीं चलेगा तो recursion भी नहीं होगा it means base case खुद For loop decide कर रहा

↳ इस कारण से यहाँ base case नहीं बनाया गया है।

↳ जब loop नहीं चलेगा तो खुद recursion रुक जाएगा

↳ last में back track करने के लिए ठीक एक data को Pop कर रहे हैं जिससे हो return होने की स्थिति में ठीक पिछे पहुंच जा रहा है।

```
if __name__ == __main__
    arr = [1,2,3,3,4,5,5]
    arr = set (arr)
    arr = list (arr)
    N = len (arr)   res = [] , Sub = []
    Subset (arr, res, Sub, 0)
```

Time Complexity

$O(2^n)$

DP के through काम किया जा सकता है!