Backtracking

- Backtracking is used for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing the solutions that fails to satisfy constraint of problem at anytime.

it is basically optimized recursion
(base case, recursive equation, self work, backtracking condⁿ)

Backtracking condⁿ is used to give same initial conditions to all possible paths of possible solution.

eg Find all subsets of given string.

Recursion

```
def sub(a,i,n,s):
    if (i==n):          # base case
        global ans.
        ans.append(s)
        return
    else:
                                self work.
        sub(a,i+1,n,s+a[i])   # recursive eqn
        sub(a,i+1,n,S)

ans = []
sub("123", 0, 3, "")
print(ans)
```

→ here we everytime making new string, so more time & space complexity.

Backtracking

```
def sub(a,i,n,v):
    if (i==n):
        ans.append(''.join(v))   remember to do this, don't append v. as if other v change this v also change. So need copy of v here
        return
    else:
        v.append(a[i])
        sub(a,i+1,n,v)
        v.pop()          # backtracking condn
        sub(a,i+1,n,v)

global ans
ans, v = [],[]
sub("123", 0, 3, v)
print(ans)
```

here we use a stack where we push a[i] & for next step we pop to provide same input for other condⁿ.

→ Take or don't take approach.

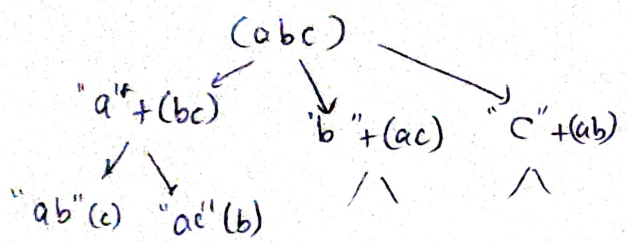
Q When to use backtracking ?
When we build soln's incrementally by searching all feasible soln's.

eg Given string of all unique characters, find all permutations of string.

eg for abc → abc acb bac bca cab cba

all characters want to come at first + permute for left substring.

recursion



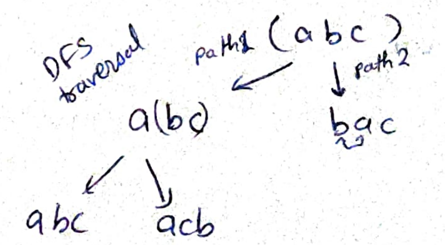

```
def permute (s, ans):
    if (len (s) == 0):
        print (ans)
        return
    for i in range (len (s)):
        ch = s [i]
        left_substr = s[0:i]
        right_substr = s[i+1:]
        rest = left_substr + right_substr
        permute (rest, ans + ch)
```

here to much space & time for finding substr

removed ith character

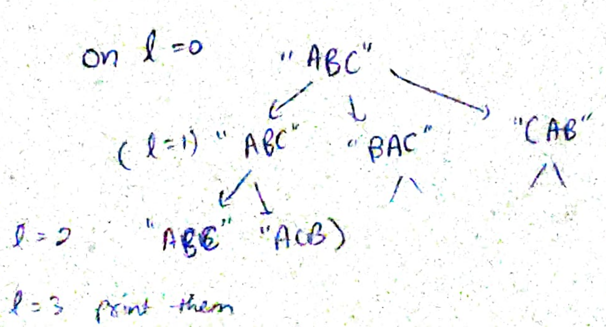
Backtracking

```
def permute (a, l, r):
    if l == r:
        print ('' .join (a))
    else:
        for i in range (l, r):
            a[l], a[i] = a[i], a[l]
            permute (a, l+1, r)
            a[l], a[i] = a[i], a[l]
```



but at this point original str changed so need to do backtrack so also get path 2 abc as initial string

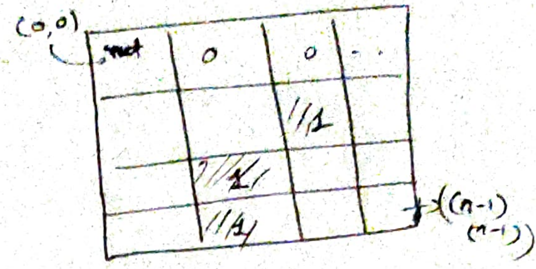
```
str = "ABC"
n = len(str)
a = list (str)
permute (a, 0, n)
```



Q1) Rat in a Maze

Rat has to go from (0,0) to (n-1, n-1) he can move up, down, right, left. Find no. of ways to reach there.

⇒ If just used normal recursion so can have infinite recursion. So we have to maintain a array for visited cells.



1 -> blocked.

Code

```
def isSafe (i, j, n, visited):
    return (i>=0 and i<n and j>=0 and j<n and visited [i][j]==0)

def helper (i, j, n, a, visited):
    if (i==n-1 and j==n-1):
        global totalpaths
        totalpaths += 1
        return
    if (not isSafe (i, j, n, visited)):
        return
    visited [i][j] = 1
    if (i+1<n and a[i+1][j]==0):
        helper (i+1, j, n, a, visited)
    if (i-1>0 and a[i-1][j]==0):
        helper (i-1, j, n, a, visited)
    if (j+1<n and a[i][j+1]==0):
        helper (i, j+1, n, a, visited)
    if (j-1>0 and a[i][j-1]==0):
        helper (i, j-1, n, a, visited)

    visited [i][j] = 0        # Backtracking
    return
```

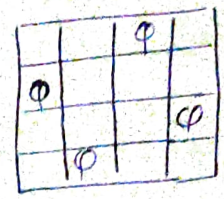
```
# input

n = int (input ())
a = []
for j in range (n):
    a.append (list (map(int,
        input(). split())))

global totalpaths
totalpaths = 0
visited = [[0]*n for i in
        range (n)]
helper (0, 0, n, a, visited)
print (totalPaths)
```

print path using output so far concept
(GFG submitted soln)

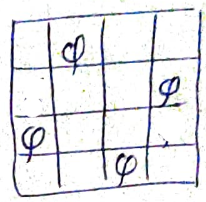
Q N Queen Problem

Given a N x N board find no. of ways to place N queens, so that no queens attacks the other.

for N = 4



&



Concept

1) Before placing a queen, check if place is safe.
2) N x N with N queens should always have only 1 queen at given row and column.
3) So if we find a row where no queen can be placed we stop & backtrack.

⇒ in code we consider filling queens column wise from col 0 to n-1

\# isSafe is called when col queens are already placed from 0 to n-1. So we only need to check for left side for attacking queens.

Code

```
def isSafe (board, row, col):
    for i in range (col):
        if board [row] [i] == 1:
            return False

    for i, j in zip (range (row, -1, -1), range (col, -1, -1)):
        if board [i] [j] == 1:
            return False

    for i, j in zip (range(row, N, 1), range (col, -1, -1)):
        if board [i] [j] == 1:
            return False

    return True
```

```
def  SolveNQUtil (board, col):
    if  col >= N :                          # base case if all queens are
                                              placed
        return True

    for i in range (N):                     # for given col check all rows
        if isSafe (board, i, col):
                                            # place at cell of given col which is
            board [i][col] = 1                safe
            if SolveNQUtil (board, col+1) == True:   # place at given col & check
                print (board)                          for col + 1

            board [i] [col] = 0             # backtrack

    return False


n = int (input())
board = [[0]*n for i in range(n)]
global N
N = n
SolveNQUtil (board, 0)
```

Q Sum closest to given target (Leetcode 1774)

You would like to make desert given n base flavours & m types of topings

Rules :- 1) exactly use 1 base   2) one or more type of toppings
3) at most 2 of each type of topping.

given 3 inputs 1) base cost   2) topping Cost   3) target

You want to make desert with total Cost close to target as possible.
return closest cost.

eg
[1,7] base   [2,3]   [10]
[3,4] topping   [4,5,100]   [1]
10 target   18   1
7+3=10   3+4+2*5   return
return 10   =17   10

```
class Solution :
    def closestCost (self , base , toppingCost , target) -> int :
        self.ans = self.min_d = float ('inf')
        def sub (top , i , cost , target) :
            diff = abs ( target - cost)
            if ( diff < self. min_d) :
                self. min_d = diff
                self . ans = cost
            elif (diff == self. min_d):
                self.ans = min (self.ans, cost)

            if (i > len( top) - 1):
                return
            if (cost > target ):
                return
            else :
                sub (top, i+1 , cost , target)
                sub ( top, i+1 , cost + top[i] , target)
                sub (top, i+1 , cost + top[i]+ top[i] , target)


        for cost in base :
            sub ( toppingCost , 0, cost, target)

        return self.ans .
```