RECURSION

Q: Subset Sum:

Input:
N = 2
arr[] = {2, 3}
Output:
0 2 3 5

```python
class Solution:
    def subsetSums(self, arr, N):
        ans=[]
        def f(arr,index,n,sum,ans):
            if index==n:
                ans.append(sum)
                return
            #take the element
            f(arr,index+1,n,sum+arr[index],ans)
            #not take
            f(arr,index+1,n,sum,ans)
        f(arr,0,N,0,ans)
        ans.sort()
        return ans
```

Time Complexity: O(2^n)+O(2^n log(2^n)). Each index has two ways. You can either pick it up or not pick it. So for n index time complexity for O(2^n) and for sorting it will take (2^n log(2^n)).

Space Complexity: O(2^n) for storing subset sums, since 2^n subsets can be generated for an array of size n.

Q: Subset II:

Time Complexity: O(2^n *(k log (x))). 2^n for generating every subset and k* log(x) to insert every combination of average length k in a set of size x. After this, we have to convert the set of combinations back into a list of list /vector of vectors which takes more time**.**

Space Complexity: O(2^n * k) to store every subset of average length k. Since we are initially using a set to store the answer another O(2^n *k) is also used.

Brute-force :

```
class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        ans = set()
        res = []

        def f(index, temp):
            if index == len(nums):
                ds=sorted(temp)
                ans.add(tuple(ds))  # Sort the temp list before adding to ans
                return
            # take the element
            temp.append(nums[index])
            f(index + 1, temp)
            temp.pop()
            # not take
            f(index + 1, temp)

        f(0, [])
        for i in ans:
            res.append(list(i))
        return res
```

Optimal approach: Without using extra set data structure and skipping the duplicates by sorting the list initially.

Time Complexity: O(2^n) for generating every subset and **O(k)** to insert every subset in another data structure if the average length of every subset is **k.** Overall **O(k * 2^n).**

Space Complexity: O(2^n * k) to store every subset of average length k. Auxiliary space is **O(n)** if n is the depth of the recursion tree.

```
class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        ans = []
        nums.sort()
        temp=[]
        def subset(index):
            ans.append(temp[:])
            for i in range(index,len(nums)):
                if i!=index and nums[i]==nums[i-1]:
                    continue
                temp.append(nums[i])
                subset(i+1)
                temp.pop()
        subset(0)
        return ans
```

Q: Combinations-I:

```
Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]
```

```
class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        ans=[]
        ds=[]
        def cs(index,target):
            if index==len(candidates):
                if target==0:
                    ans.append(ds[:])          #append the copy of list in ans
                return
            if candidates[index]<=target:
                #take
                ds.append(candidates[index])
                cs(index, target-candidates[index])
                ds.pop()   #as list is refrenced by address
            #not take
            cs(index+1,target)

        cs(0,target)
        return ans
```

Q: Combination-II: This approach is similar to subset sum-II

Time Complexity:O(2^n*k)

Reason: Assume if all the elements in the array are unique then the no. of subsequence you will get will be O(2^n). we also add the ds to our ans when we reach the base case that will take “k”//average space for the ds.

Space Complexity:O(k*x)

Reason: if we have x combinations then space will be x*k where k is the average length of the combination.

```
class Solution:
    def combinationSum2(self, candidates: List[int], target: int) -> List[List[int]]:
        candidates.sort()
        ans=[]
        temp=[]
        def cs(index,target):
            if target==0:
                ans.append(temp[:])
                return
            for i in range(index,len(candidates)):
                if i!=index and candidates[i]==candidates[i-1]:
                    continue
                #take
                if candidates[i]<=target:
                    temp.append(candidates[i])
                    cs(i+1,target-candidates[i])
                    temp.pop()
        cs(0,target)
        return ans
```

Q: Palindrome Partitioning:

Time Complexity: O((2^n) *k*(n/2))

Reason: O(2^n) to generate every substring and O(n/2) to check if the substring generated is a palindrome. O(k) is for inserting the palindromes in another data structure, where k is the average length of the palindrome list.

Space Complexity: O(k * x)

Reason: The space complexity can vary depending upon the length of the answer. k is the average length of the list of palindromes and if we have x such list of palindromes in our final answer. The depth of the recursion tree is n, so the auxiliary space required is equal to the O(n).

```
class Solution:
    def partition(self, s: str) -> List[List[str]]:
        def isPallindrome(st):
            st1=st[::-1]
            if(st==st1):
                return True
            else:
                return False
        ans=[]
        temp=[]
        def p(index):
            if index==len(s):
                ans.append(temp[:])
                return
            for i in range(index,len(s)):
                if isPallindrome(s[index:i+1]):
                    temp.append(s[index:i+1])
                    p(i+1)
                    temp.pop()
        p(0)
        return ans
```

Q: Kth permutation:

```python
import math
class Solution:
    ans=""
    def getPermutation(self, n: int, k: int) -> str:
        A=n
        l=[]
        for i in range(1,n+1):
            l.append(i)
        # print(l)
        def find(n,k,A)->str:
            if len(self.ans)==A:
                return
            f=math.factorial(n-1)
            index=int(k/f)
            self.ans+=str(l[index])
            del l[index]
            find(n-1,k%f,A)

        find(n,k-1,A)
        return self.ans
```

BACKTRACKING:

Q: Print all permutation:

```python
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        ans=[]
        ds=[]
        freq=len(nums)*[-1]
        def f():
            if len(ds)==len(nums):
                ans.append(ds[:])
                return
            for i in range(len(nums)):
                if freq[i]==-1:
                    ds.append(nums[i])
                    freq[i]=0
                    f()
                    freq[i]=-1
                    ds.pop()
        f()
        return ans
```

Q:N-QUEEN PROBLEM:

```python
class Solution:
    def solveNQueens(self, n: int) -> List[List[str]]:
        ans =[]
        #this for loop create the 2-D matrix like
        #ans=[['.','.','.','.'],['.','.','.','.'],['.','.','.','.'],['.','.','.','.']]
        for i in range(n):
            temp=[]
            for j in range(n):
                temp.append('.')
            ans.append(temp)
        res=[]
        def isSafe(r,c):
            #check safe for same row
            j=c
            while(j>=0):
                if ans[r][j]=='Q':
                    return False
                j=j-1
            #check for upper triangle
            i=r
            j=c
            while(i>=0 and j>=0):
                if ans[i][j]=='Q':
                    return False
                i=i-1
                j=j-1
            #check for lower traingle
            i=r
            j=c
            while(i<n and j>=0):
                if ans[i][j]=='Q':
                    return False
                i=i+1
                j=j-1
            return True
        def f(col,n):
            if col==n:
                # add the current configuration to the result
                config=[]
                for row in ans:                 #row=['.','.','.','.']
                    #join each row and then append to list
                    config.append(''.join(row))
                res.append(config)
                return

            for row in range(n):
                #place in every row of each column and check that is it safe or not
                if(isSafe(row,col)):
                    ans[row][col]='Q'
                    #again call the recursive function for col+1
                    f(col+1,n)
                    #during backtracking undo everything
                    ans[row][col]='.'
        f(0,n)
        return res
```

Q: Sudoku solver:

```
def solveSudoku(self, board: List[List[str]]) -> None:

    def isPossible(k,board,row,col):
        for i in range(9):
            if board[i][col]==k or board[row][i]==k:
                return False
        start_row=(row//3)*3
        start_col=(col//3)*3
        end_row=start_row+2
        end_col=start_col+2
        for i in range(start_row,end_row+1):
            for j in range(start_col,end_col+1):
                if board[i][j]==k:
                    return False
        return True

    def solve(board):
        for i in range(9):
            for j in range(9):
                if board[i][j]==".":
                    for k in range(1,10):
                        if(isPossible(str(k),board,i,j)):
                            board[i][j]=str(k)
                            if solve(board):
                                return True
                            board[i][j]="."
                    return False
        return True
    solve(board)
```

Q: M-coloring problem:

```
#Function to determine if graph can be coloured with at most M colours such
#that no two adjacent vertices of graph are coloured with same colour.
def graphColoring(graph, k, V):
    color=V*[-1]
    def f(node):
        if node==V:
            return True
        for i in range(k):
            if isPossible(node,i):
                color[node]=i
                if f(node+1)==True:
                    return True
                #backtrack
                color[node]=-1
        return False

    def isPossible(node,i):
        for j in range(V):
            if graph[node][j]==1 and color[j]==i:
                return False
        return True
    if f(0):
        return True
    return False
```

Q:Rat in a maze:

```python
def findPath(self, m, n):
    ans = []
    vis = [[0 for _ in range(n)] for _ in range(n)]
    temp = []

    def f(i, j):
        if i == n - 1 and j == n - 1:
            ans.append(''.join(temp[:]))
            return

        # Up
        if i > 0 and m[i - 1][j] != 0 and vis[i - 1][j] == 0:
            vis[i][j] = 1
            temp.append('U')
            f(i - 1, j)
            temp.pop()
            vis[i][j] = 0

        # Down
        if i < n - 1 and m[i + 1][j] != 0 and vis[i + 1][j] == 0:
            vis[i][j] = 1
            temp.append('D')
            f(i + 1, j)
            temp.pop()
            vis[i][j] = 0

        # Left
        if j > 0 and m[i][j - 1] != 0 and vis[i][j - 1] == 0:
            vis[i][j] = 1
            temp.append('L')
            f(i, j - 1)
            temp.pop()
            vis[i][j] = 0

        # Right
        if j < n - 1 and m[i][j + 1] != 0 and vis[i][j + 1] == 0:
            vis[i][j] = 1
            temp.append('R')
            f(i, j + 1)
            temp.pop()
            vis[i][j] = 0

    if m[0][0] == 1:
        f(0, 0)
    if not ans:
        return []
    return ans
```

Same code with little changes:

```python
#User function Template for python3

class Solution:
    def findPath(self, m, n):
        vis=[]
        for i in range(n):
            temp=[]
            for j in range(n):
                temp.append(0)
            vis.append(temp)
        ans=[]
        def f(i, j, move):
            if i == n - 1 and j == n - 1:
                ans.append(move)
                return

            # Up
            if i> 0 and m[i - 1][j] != 0 and vis[i - 1][j] == 0:
                vis[i][j] = 1
                f(i - 1, j, move + 'U')
                vis[i][j] = 0

            # Down
            if i < n-1 and m[i + 1][j] != 0 and vis[i + 1][j] == 0:
                vis[i][j] = 1
                f(i + 1, j, move + 'D')
                vis[i][j] = 0

            # Left
            if j > 0 and m[i][j - 1] != 0 and vis[i][j - 1] == 0:
                vis[i][j] = 1
                f(i, j - 1, move + 'L')
                vis[i][j] = 0

            # Right
            if j < n-1 and m[i][j + 1] != 0 and vis[i][j + 1] == 0:
                vis[i][j] = 1
                f(i, j + 1, move + 'R')
                vis[i][j] = 0


        if m[0][0] == 1:
            f(0, 0, '')

        if not ans:
            return [-1]  # Return a list containing -1

        return ans  # Sort the list of paths
```