LC 47

class Solution:

def permuteUnique(self, nums: List[int]) -> List[List[int]]:

def backtrack(cur,remain):

if len(remain) == 0:

res.append(cur)

return

for i,n in enumerate(remain):

if i and n == remain[i-1]:

continue

backtrack(cur+[n],remain[:i]+remain[i+1:])

res = []

nums.sort()

backtrack([],nums)

return res

LC 216

class Solution:

def combinationSum3(self, k: int, n: int) -> List[List[int]]:

if n > (19-k) \* k // 2 or n < k \* (k+1) // 2:

return []

def comb(k,a,n):

if k == 1:

if a <= n and n <= 9:

return [[n]]

else:

return []

res = []

for i in range(a,9):

temp = comb(k-1,i+1,n-i)

if temp:

for t in temp:

res += [[i]+t]

return res

return comb(k,1,n)

LC 526

class Solution:

def countArrangement(self, n: int) -> int:

def backtrack(cur,remain):

if len(remain) == 0:

nonlocal count

count += 1

return

for i,r in enumerate(remain):

if r % cur == 0 or cur % r == 0:

backtrack(cur+1,remain[:i]+remain[i+1:])

count = 0

backtrack(1,list(range(1,n+1)))

return count

LC 140

class Solution:

def wordBreak(self, s: str, wordDict: List[str]) -> List[str]:

res = []

def backtrack(s,wordDict,sentence):

if s == "":

res.append(sentence[:-1])

return

for i in range(1,min(10,len(s)+1)):

if s[:i] in wordDict:

backtrack(s[i:],wordDict,sentence+s[:i]+" ")

backtrack(s,set(wordDict),"")

return res

LC 37

class Solution:

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

"""

Do not return anything, modify board in-place instead.

"""

def row(board,i,k):

for j in range(9):

if board[i][j] == str(k):

return False

return True

def col(board,j,k):

for i in range(9):

if board[i][j] == str(k):

return False

return True

def box(board,i,j,k):

i = i - i % 3

j = j - j % 3

for m in range(3):

for n in range(3):

if board[i+m][j+n] == str(k):

return False

return True

to\_be\_filled = []

checklist = dict()

for i in range(9):

for j in range(9):

if board[i][j] == ".":

to\_be\_filled.append((i,j))

checklist[(i,j)] = set()

for k in range(1,10):

if row(board,i,k) and col(board,j,k) and box(board,i,j,k):

checklist[(i,j)].add(k)

def backtrack(board,processed,remaining):

if len(remaining) == 0:

nonlocal success

success = True

return

for num in checklist[remaining[0]]:

a = remaining[0][0]

b = remaining[0][1]

if row(board,a,num) and col(board,b,num) and box(board,a,b,num):

board[a][b] = str(num)

checklist[(a,b)] = checklist[(a,b)] - {num}

backtrack(board,processed+[(a,b)],remaining[1:])

if not success:

board[a][b] = "."

checklist[(a,b)].add(num)

success = False

backtrack(board,[],to\_be\_filled)