Backtracking (record partial solution)（偏难）

分析

- plot the implicit tree: path, cur, children, partial solution
- all solutions(paths) <= DFS + Backtracking
  - **DFS(PATH::partial solution, REST::candidate choices, MISC::relevant info)**
- optimal solution(path) <= DP + Backtracking
- number of solutions <= easier; DP
  - **DFS(REST::candidate choices, MISC::relevant info) + CACHE**
- k-th dfs solution <= perfect implicit tree

subset, combination & permutation

Subsets I (distinct case)
Subsets II (duplicate case)
Combinations (distinct case, k-subset)

Permutations I (distinct case)
Permutations II (duplicate case)
next permutatioin `从右往左找到首降A[j],从j往右找到最右A[i]>A[j],交换ij，翻转A[j+1:]`
Permutation Sequence (k-th permutation) (perfect implicit tree)

Combination Sum I (distinct case, all positive, reuse)
Combination Sum II (duplicate case, all positive, no reuse)
Combination Sum III (distinct case, all positive, no reuse)
Combination Sum IV (distinct case, all positive, reuse, only size)

Palindrome Permutation II
Beautiful Arrangement (divisible rule)(go from N to 1)
Factor Combinations (画树)
Letter Combinations of a Phone Number
Android Unlock Patterns (DP)
Palindrome Partitioning

misc

Generate Parentheses
Letter Case Permutation
Flip Game I
Flip Game II (if the starting player can guarantee a win)
Generalized Abbreviation
Count Numbers with Unique Digits
Binary Watch
Gray Code
Stickers to Spell Word
Split Array into Fibonacci Sequence
Restore IP Addresses

matrix

Sudoku Solver
N-Queens I
N-Queens II

word

Minimum Unique Word Abbreviation
Additive Number
Word Squares `trie`
Word Pattern I
Word Pattern II
Word Break I
Word Break II
Word Search I
Word Search II
Word Ladder I
Word Ladder II
Regular Expression Matching
Wildcard Matching

```
class Solution:
    def subsets1_distinct(self, nums):
        res = []
```

```
        def dfs(path=[], rest=0):
            res.append(path)
            for i in range(rest, len(nums)):
                dfs(path+nums[i:i+1], i+1)
        dfs()
        return res

    def subsets2_duplicate(self, nums):
        nums.sort()                                             # sorted
        res = []
        def dfs(path=[], rest=0):
            res.append(path)
            for i in range(rest, len(nums)):
                if i > rest and nums[i] == nums[i-1]: continue    # skip
                dfs(path + nums[i:i+1], i+1)
        dfs()
        return res

    def subsets_distinct_sizeK(self, n, k):
        res = []
        def dfs(path=[], rest=0, n=n, k=k):
            if k == 0:
                res.append(path)
            else:
                for i in range(rest, n):
                   dfs(path+[i], i+1, n, k-1)
        dfs()
        return res

    def permute1_distinct(self, nums):
        res = []
        def dfs(path=[], rest=nums):
            if not rest:
                res.append(path)
            else:
                 for i in range(len(rest)):
                     dfs(path+rest[i:i+1], rest[:i]+rest[i+1:])
        dfs()
        return res

    def permute2_duplicate(self, nums):
        nums.sort()                                               # sorted
        res = []
        def dfs(path=[], rest=nums):
            if not rest:
```

```
                res.append(path)
            else:
                 for i in range(len(rest)):
                     if i > 0 and rest[i] == rest[i-1]: continue   # skip
                     dfs(path+rest[i:i+1], rest[:i]+rest[i+1:])
        dfs()
        return res

    def combinationSum1_distinct_reuse(self, nums, target):
        res = []
        def dfs(path=[], rest=0, target=target):
            if target == 0:
                res.append(path)
            elif target > 0:
                for i in range(rest, len(nums)):
                    dfs(path + nums[i:i+1], i, target - nums[i])  # reuse i
        dfs()
        return res

    def combinationSum2_duplicate_noreuse(self, nums, target):
        nums.sort()                                               # sorted
        res = []
        def dfs(path=[], rest=0, target=target):
            if target == 0:
                res.append(path)
            elif target > 0:
                for i in range(rest, len(nums)):
                    if i > rest and nums[i] == nums[i-1]: continue  # skip
                    dfs(path + nums[i:i+1], i+1, target - nums[i])
        dfs()
        return res

    def combinationSum3_distinct_noreuse_sizeK(self, k, n):
        res = []
        def dfs(path=[], rest=1, target=n, k=k):
            if k == 0:
                if target == 0: res.append(path)
            else:
                for i in range(rest, 10):
                    dfs(path+[i], i+1, target-i, k-1)
        dfs()
        return res

    def combinationSum4_onlySize(self, nums, target):         # DP for size
        cache = {}
```

```
        def dfs(target=target):
            if target < 0: return 0
            if target in cache: return cache[target]
            ret = 1 if target == 0 else sum(dfs(target-n) for n in nums)
            cache[target] = ret
            return ret
        return dfs()

    def generateParenthesis(self, n):
        """
        :type n: int
        :rtype: List[str]
        """
        res = []
        def dfs(path="", left=0, right=0, n=n):
            if left == n and right == n:
                res.append(path)
            elif left == right:
                dfs(path+'(', left+1, right, n)
            elif right < left:
                if left < n:
                    dfs(path+'(', left+1, right, n)
                dfs(path+')', left, right+1, n)
        dfs()
        return res
```