Combination/Permutation

Backtracking

77. Combinations

Given two integers `n` and `k`, return *all possible combinations of* `k` *numbers out of the range* `[1, n]`.

You may return the answer in any order.

```python
class Solution:
    def combine(self, n: int, k: int) -> List[List[int]]:
        nums = list(range(1, n + 1))
        res = []

        def dfs(nums, k, path, pos):
            if k == 0:
                res.append(path)
                return
            if k > len(nums) - pos + 1 or pos == n:
                return

            for i in range(pos, n):
                dfs(nums, k - 1, path + [nums[i]], i + 1)

        dfs(nums, k, [], 0)
        return res
```

39. Combination Sum

Given an array of distinct integers `candidates` and a target integer `target`, return *a list of all unique combinations of* `candidates` *where the chosen numbers sum to* `target`. You may return the combinations in any order.

The same number may be chosen from `candidates` an unlimited number of times. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

It is guaranteed that the number of unique combinations that sum up to `target` is less than `150` combinations for the given input.

- `1 <= candidates.length <= 30`
- `1 <= candidates[i] <= 200`
- All elements of `candidates` are distinct.
- `1 <= target <= 500`

```python
class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        # candidates.sort() # no need to sort cuz unlimitted
        res = []
        def dfs(nums, path, target, pos):
            if target == 0:
                res.append(path[:])
                return
            if target < 0:
                return

            for i in range(pos, len(nums)): # can use the same number unlimited times
                path.append(nums[i])
                dfs(nums, path, target - nums[i], i) # but dont use before i to avoid dups
                path.pop()

        dfs(candidates, [], target, 0)
        return res
```

40. Combination Sum II

Given a collection of candidate numbers (`candidates`) and a target number (`target`), find all unique combinations in `candidates` where the candidate numbers sum to `target`.

Each number in `candidates` may only be used once in the combination.

Note: The solution set must not contain duplicate combinations.

```python
class Solution:
    def combinationSum2(self, candidates: List[int], target: int) -> List[List[int]]:
        candidates.sort() # need to sort and dedup later
        res = []
        def dfs(nums, pos, target, path):
            if target == 0:
                res.append(path[:])
                return
            if target < 0 or pos >= len(nums):
                return

            for i in range(pos, len(nums)):
                if i > pos and nums[i] == nums[i - 1]:
                    continue # dedup before the recursion
                path.append(nums[i])
                dfs(nums, i + 1, target - nums[i], path) # only be used once
                path.pop()

        dfs(candidates, 0, target, [])
        return res
```

216. Combination Sum III

Find all valid combinations of `k` numbers that sum up to `n` such that the following conditions are true:

- Only numbers `1` through `9` are used.
- Each number is used at most once.

Return *a list of all possible valid combinations*. The list must not contain the same combination twice, and the combinations may be returned in any order.

```python
class Solution:
    def combinationSum3(self, k: int, n: int) -> List[List[int]]:
        nums = list(range(1, 10))
        res = []

        def dfs(nums, target, k, path, pos):
            if target == 0 and k == 0:
                res.append(path)
                return
            if target != 0 and k == 0:
                return
            if pos == n:
                return

            for i in range(pos, len(nums)):
                dfs(nums, target - nums[i], k - 1, path + [nums[i]], i + 1)

        dfs(nums, n, k, [], 0)
        return res
```

377. Combination Sum IV

Given an array of distinct integers `nums` and a target integer `target`, return *the number of possible combinations that add up to* `target`.

The answer is guaranteed to fit in a 32-bit integer.

No dedup at all

```python
class Solution:
    def combinationSum4(self, nums: List[int], target: int) -> int:
        @lru_cache(None)
        def dfs(target):
            if target < 0:
                return 0
            if target == 0:
                return 1
            res = 0
            for num in nums:
                res += dfs(target - num)
            return res

        return dfs(target)
```

78. Subsets

Given an integer array `nums` of unique elements, return *all possible subsets (the power set)*.

The solution set must not contain duplicate subsets. Return the solution in any order.

```python
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        # O(2^n), for loop, easy to dedup
        self.res = []
        n = len(nums)

        def dfs(pos, path):
            self.res.append(path)
            for i in range(pos, n):
                dfs(i + 1, path + [nums[i]])

        dfs(0, [])
        return self.res

        # O(2^n), take or no take
        self.res = []
        n = len(nums)

        def dfs(pos, path):
            if pos == n:
                self.res.append(path)
                return
            # take current num
            dfs(pos + 1, path + [nums[pos]])
            # no take
            dfs(pos + 1, path)

        dfs(0, [])
        return self.res
```

90. Subsets II

Given an integer array `nums` that may contain duplicates, return *all possible subsets (the power set)*.

The solution set must not contain duplicate subsets. Return the solution in any order.

```
Input: nums = [1,2,2]

Output:
[[],[1],[1,2],[1,2,2],[2],[2,2]]
```

```python
class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        # O(2^n), take or no take
        self.res = []
        n = len(nums)
        nums.sort()
        def dfs(pos, path):
            self.res.append(path)

            for i in range(pos, n):
                if i > pos and nums[i] == nums[i - 1]:
                    continue # dedup
                dfs(i + 1, path + [nums[i]])

        dfs(0, [])
        return self.res
```

```
def dfs(pos, path):
    res.append(path[:])
    for i in range(pos, n):
        if i > pos and nums[i-1] == nums[i]:
            continue
        path.append(nums[i])
        dfs(i + 1, path)
        path.pop()
```

46. Permutations

Given an array `nums` of distinct integers, return *all the possible permutations*. You can return the answer in any order.

```python
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)
        res = []

        def dfs(nums, path):
            if len(path) == n:
                res.append(path[:])
                return

            for i in range(len(nums)):
                path.append(nums[i])
                dfs(nums[:i] + nums[i+1:], path)
                path.pop()

        dfs(nums, [])
        return res
```

47. Permutations II

Given a collection of numbers, `nums`, that might contain duplicates, return *all possible unique permutations in any order.*

```python
class Solution:
    def permuteUnique(self, nums: List[int]) -> List[List[int]]:
        nums.sort()

        n = len(nums)
        res = []

        def dfs(nums, path):
            if len(path) == n:
                res.append(path[:])
                return

            for i in range(len(nums)):
                if i and nums[i - 1] == nums[i]:
                    continue # dedup [1a, 1b, 2] == [1b, 1a, 2]
                path.append(nums[i])
                dfs(nums[:i] + nums[i+1:], path)
                path.pop()

        dfs(nums, [])
        return res
```

491. Increasing Subsequences

Given an integer array `nums`, return all the different possible increasing subsequences of the given array with at least two elements. You may return the answer in any order.

The given array may contain duplicates, and two equal integers should also be considered a special case of increasing sequence.

`Input: nums = [4,6,7,7]`

`Output: [[4,6],[4,6,7],[4,6,7,7],[4,7],[4,7,7],[6,7],[6,7,7],[7,7]]`

```python
class Solution:
    def findSubsequences(self, nums: List[int]) -> List[List[int]]:
        # O(2^n)
        res = set()
        n = len(nums)

        def dfs(pos, path):
            if len(path) > 1:
                res.add(tuple(path[:]))

            for i in range(pos, n):
                if not path or nums[i] >= path[-1]:
                    dfs(i + 1, path + [nums[i]]) # take nums[i] and start from i + 1

        dfs(0, [])
        return res
```