L8 Combination Sum I

Tuesday, December 20, 2022 10:50 AM

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.
The test cases are generated such that the number of unique combinations that sum up to target is less than 150 combinations for the given input.

Example 1:
Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]
Explanation:
2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times.
7 is a candidate, and 7 = 7.
These are the only two combinations.

↳ Every all over tasks is point all the combination that sum is equal to the target

↳ Same Numbers we used Many Numbers of times According to the Need

Important Point

Whenever we talk about point all the combination can All Subsequence of then there 100% Recursion used No doubt

#Thought Process

arr = [2,3,6,7], target = 7

→ Whenever we talk about the pick the element from the array for pointing All the combination then

↳ Always Start with pick and Not pick Concept that we learn previously

[2, 3, 6, 7] = target 7

many times pick that element

✓ ✓ X X = [2, 2, 3]

X X X ✓ = [7]

} possible All combination

arr = [2, 3, 6, 7]
0 1 2 3

→ Now we decide that which index and how many times of that index pick ever next pick

#Recursive Sudo Code

```
myfunction (index, candidate, target, ans, ds) {
    if(index == len(candidate)) {
        if target == 0;
            ans.append(ds);
            return;
    }
    return;
```

→ Base case

```
if candidate[index] <= target {
    ds.append(candidate[index]);
    myfunction(index, candidate, target - candidate[index], ans, ds);
    ds.removed(candidate[index]);
}
myfunction(index+1, candidate, target, ans, ds);
}
```

Pick part

maybe single Integer Repeat more than 1

if we pick then target - element at every time

next pick part

#Recursion Tree

arr = [2, 3, 6, 7], target = 7

Answer Return = [[2,2,3],[7]]

fun(0,7,[ ])

fun(0,5,[2])

fun(0,3,[2,2])

fun(1,7,[ ])

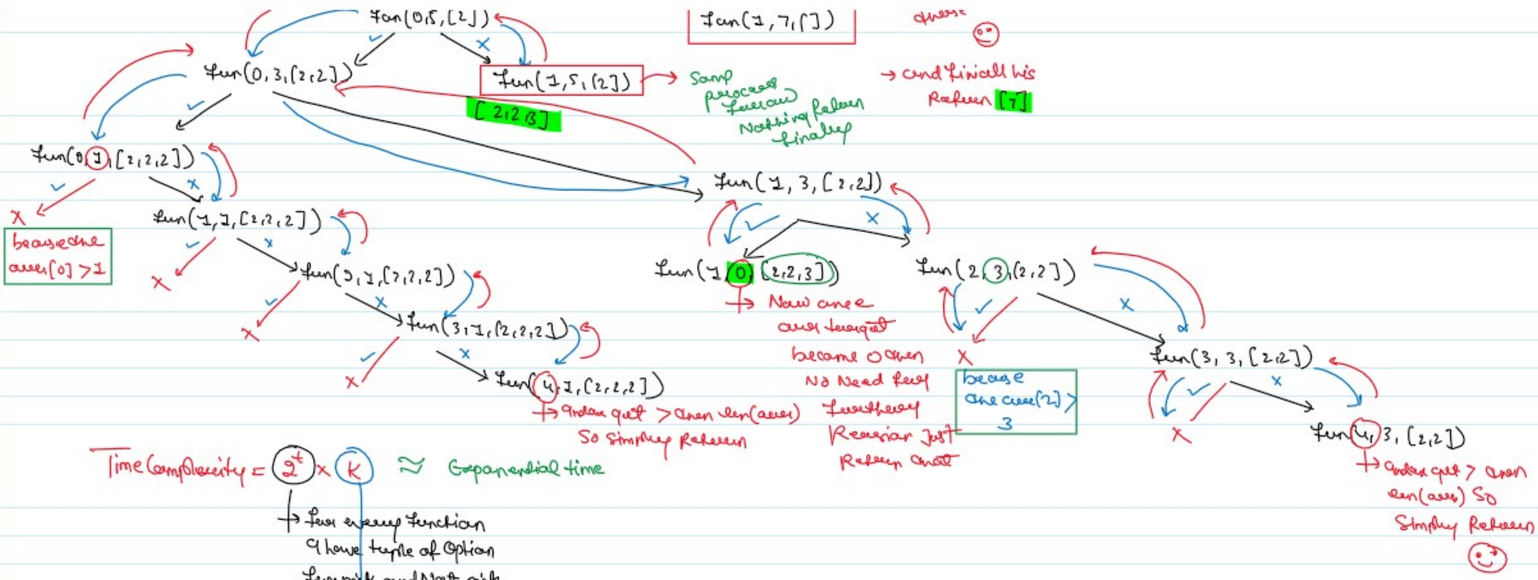
fun(1,5,[2])

[2,2,3]

[7]

Same Approach Follow with change function

→ and finally his Return [7]



Time Complexity = $2^t \times k$ ≈ Exponential time

- 2^t → for every function I have two type of Option for pick and Not pick
- k → average length of every Combination of ds that put into the ans

Important point ‼

→ If one data structure put into the another data Structure then take Some time this is Not a Constant time Operation

Q Why 2^n is Not our time?

→ Because the Not every time We distribute the our Recursion into type of part

that why on an average our distribution is 2^t Not 2^n

Space Complexity = $O(N) \times O(N) \approx O(N^2)$

- $O(N)$ → Recursion Stack Space
- $O(N)$ → Number of Combinations

→ In Recursion We can Not Calculate that eight Space Complexity because its totally depends on the Number of calls of Recursion, Combinations as well as Your Machine also

```python
class Solution:

    def findAllcombinationSum(self, index, candidates, target, ans, ds):

        if index == len(candidates):
            if target == 0:
                ans.append(ds[:])
                return
            return

        if candidates[index] <= target:
            ds.append(candidates[index])
            self.findAllcombinationSum(index, candidates, target - candidates[index], ans, ds)
            ds.remove(candidates[index])

        self.findAllcombinationSum(index + 1, candidates, target, ans, ds)

    def combinationSum(self, candidates, target):

        ans = []
        self.findAllcombinationSum(0, candidates, target, ans, [])

        return ans


ans = Solution()
candidates = [2,3,6,7]
target = 7
print(ans.combinationSum(candidates, target))
```

```java
import java.util.*;
        System.out.println("LS Combination Sum I");
    }
}

    private void findCombinations(int index, int[] candidates, int target, List<List<Integer>> ans, List<Integer> ds) {
        if (index == candidates.length) {
            if (target == 0) {
                ans.add(new ArrayList<>(ds));
            }
            return;
        }

        if (candidates[index] <= target) {
            ds.add(candidates[index]);
            findCombinations(index, candidates, target - candidates[index], ans, ds);
            ds.remove(ds.size() - 1);
        }
        findCombinations(index + 1, candidates, target, ans, ds);
    }
    public List<List<Integer>> combinationSum(int[] candidates, int target) {

        List<List<Integer>> ans = new ArrayList<>();
        findCombinations(0, candidates, target, ans, new ArrayList<>());
        return ans;


    }
}
```