L11 Combination Sum IV

Friday, December 23, 2022 10:17 AM

Given an array of distinct integers nums and a target integer target, return *the number of possible combinations that add up to* target.
The test cases are generated so that the answer can fit in a **32-bit** integer.

Example 1:
Input: nums = [1,2,3], target = 4
Output: 7
Explanation:
The possible combination ways are:
(1, 1, 1, 1)
(1, 1, 2)
(1, 2, 1)
(1, 3)
(2, 1, 1)
(2, 2)
(3, 1)
Note that different sequences are counted as different combinations.

→ 7 different and unique possible Combination

↳ This is the Extendended Version of the Combination Sum II
→ our task is count the All possible and Unique Subsequence

☺

→ In Combination Sum II our task is print all the unique, lexicographical, and sorted order Combination

→ Similarly in this problem our task is count all the possible Combinations of the given array and print

Thought Process

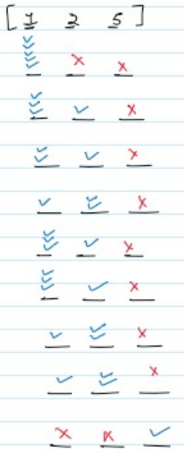
→ Concept and Logic are same that we learn in Combination Sum II only few steps are change and every things are same

→ Again here we used the Concept of pick and Not pick

Important point
We chose any element any Number of times its totally depends on you for constructing the Unique Combination

Example
arr= [1,2,5]
n=3
target = 5

Combination



= [1,1,1,1,1]
= [2,1,1,1]
= [1,2,1,1]
= [2,2,1]
= [1,1,2,1]
= [1,1,1,2]
= [2,1,2]
= [1,2,2]
= [5]

7 different Combination possible

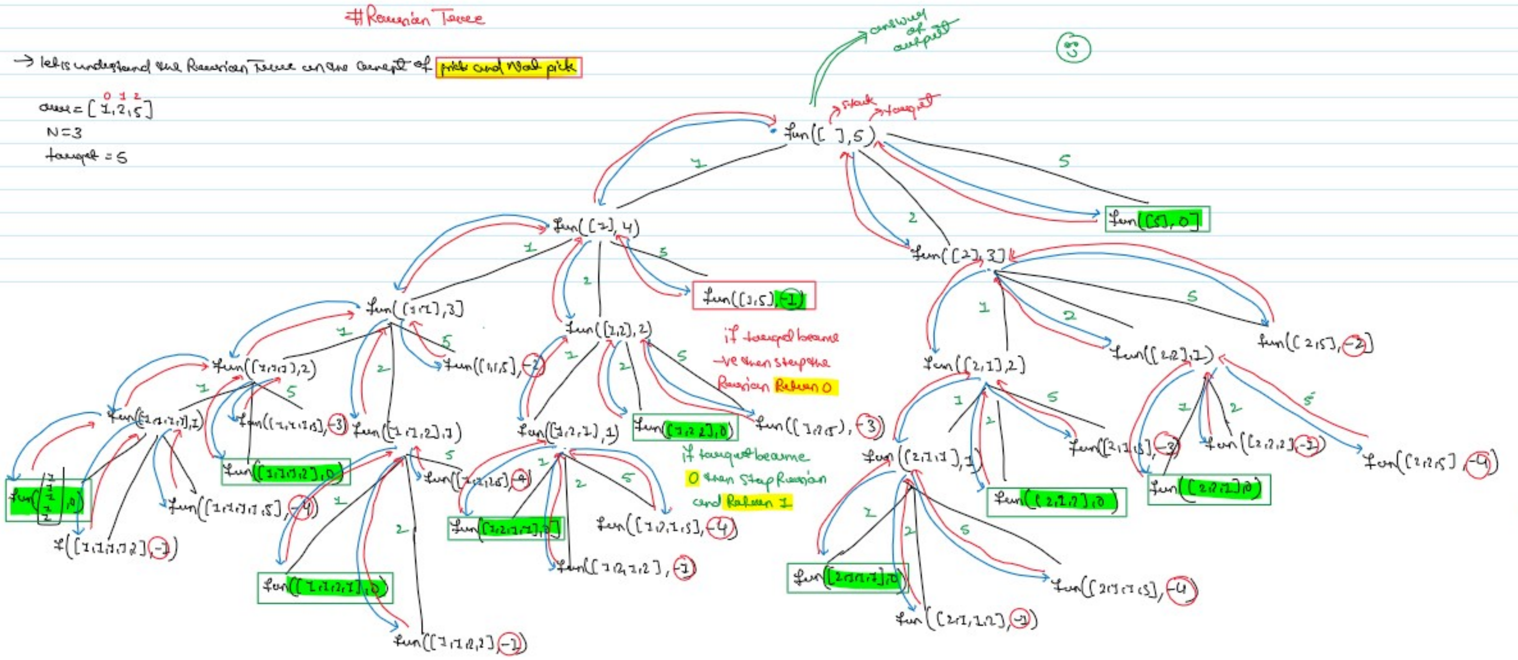
☺

ans=7

Recursion Tree

→ let's understand the Recursion Tree on the Concept of pick and Not pick

arr = [$\overset{0}{1}$, $\overset{1}{2}$, $\overset{2}{5}$]
N=3
target = 5



if target became -ve then stop the Recursion Return 0

if target became 0 then stop Recursion and Return 1

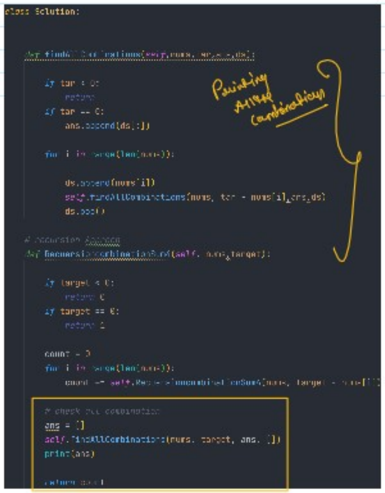
☺

→ that is the way how to Recursion Work and its Recursive Tree

```
myfunction(arr, target){

    if target<0{
        return 0
    }
    if target==0{
        return 1
    }
                                   → Base case
    count=0;
    for i in range(0,len(arr)) {
        count += myfunction(arr, target - arr(i));
    }                              → for possible all sub-Sequence
    return count;
}
```

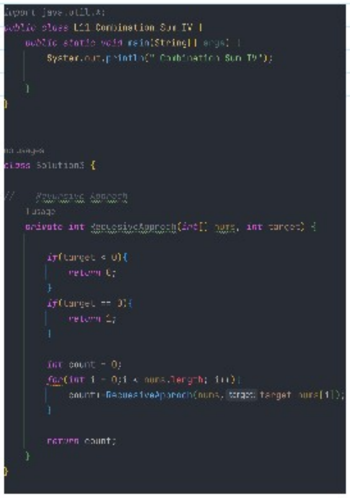
Time Complexity = $2^n \times K$
→ Number of Combination
→ for push into the ds every time

Space Complexity = $K \times x \times N$
→ average length of the every Subsequence
→ for Combination
→ Recursion stack Spaced



```
class Solution:

    def findAllCombinations(self,nums, tar,ans,ds):

        if tar < 0:
            return
        if tar == 0:
            ans.append(ds[:])

        for i in range(len(nums)):

            ds.append(nums[i])
            self.findAllCombinations(nums, tar - nums[i],ans,ds)
            ds.pop()

    # recursion Approach
    def RecursionCombinationSum4(self, nums,target):

        if target < 0:
            return 0
        if target == 0:
            return 1

        count = 0
        for i in range(len(nums)):
            count += self.RecursionCombinationSum4(nums, target - nums[i])

        # check all combination
        ans = []
        self.findAllCombinations(nums, target, ans, [])
        print(ans)

        return count
```

Printing All the combination



```
import java.util.*;
public class L11 Combination Sum IV {
    public static void main(String[] args) {
        System.out.println(" Combination Sum IV");

    }
}

no usages
class Solution3 {

//    #Recursive Approach
    1 usage
    private int RecursiveApproach(int[] nums, int target) {

        if(target < 0){
            return 0;
        }
        if(target == 0){
            return 1;
        }

        int count = 0;
        for(int i = 0;i < nums.length; i++){
            count+=RecursiveApproach(nums, target: target-nums[i]);
        }

        return count;
    }
}
```

→ Now the Constrain is Very highe so Recursion Soution Not Except then try to Optimized that Approach using

- → Recursion ✓
- → Memoization ✓] optimal
- → Tabulation ✓

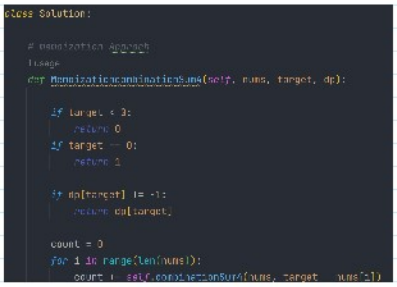
↳ This Solution show TLE because the Constrain is big >100 and ≤200

#Memoization

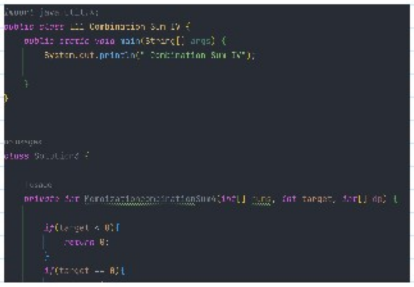
→ We Learn in dp How to Convert Recursion to Memoization in Upcomming time

→ As we See in Recursion Tree Multiple Sum Overlapping problem are present that Means dp and Memoization possible

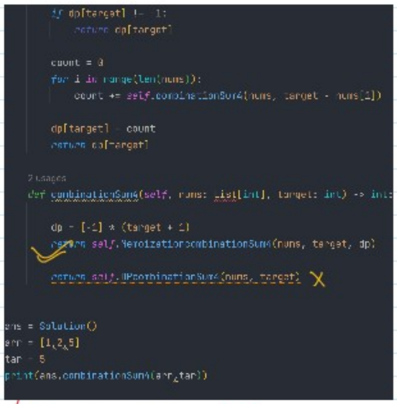
→ Initisized the dp array with size of target+1



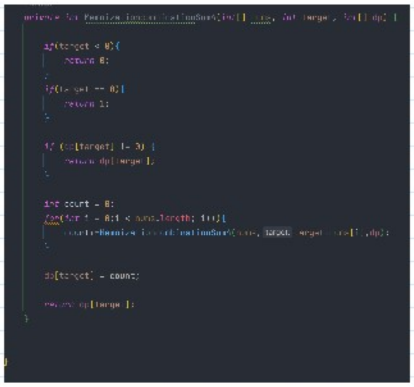
```
class Solution:

    # memoization Approach
    1 usage
    def MemoizationcombinationSum4(self, nums, target, dp):

        if target < 0:
            return 0
        if target == 0:
            return 1

        if dp[target] != -1:
            return dp[target]

        count = 0
        for i in range(len(nums)):
            count += self.combinationSum4(nums, target - nums[i])
```



```
import java.util.*;
public class L11 Combination Sum IV {
    public static void main(String[] args) {
        System.out.println(" Combination Sum IV");

    }
}

no usages
class Solution4 {

    1 usage
    private int MemoizationcombinationSum4(int[] nums, int target, int[] dp) {

        if(target < 0){
            return 0;
        }
        if(target == 0){
```



```
        if dp[target] != -1:
            return dp[target]

        count = 0
        for i in range(len(nums)):
            count += self.combinationSum4(nums, target - nums[i])

        dp[target] = count
        return dp[target]

    2 usages
    def combinationSum4(self, nums: List[int], target: int) -> int:

        dp = [-1] * (target + 1)
        return self.MemoizationcombinationSum4(nums, target, dp)

        return self.DPcombinationSum4(nums, target)


ans = Solution()
arr = [1,2,5]
tar = 5
print(ans.combinationSum4(arr,tar))
```



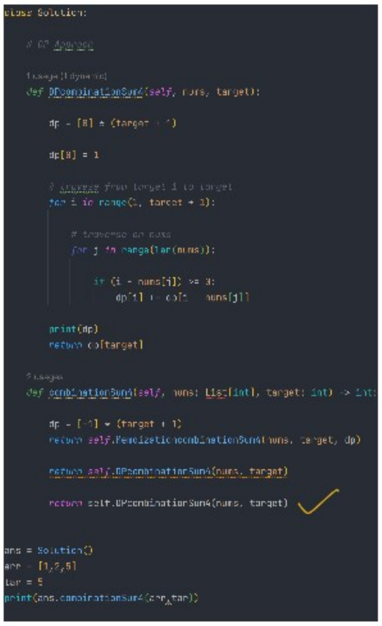
```
    private int MemoizationcombinationSum4(int[] nums, int target, int[] dp) {

        if(target < 0){
            return 0;
        }
        if(target == 0){
            return 1;
        }

        if (dp[target] != 0) {
            return dp[target];
        }

        int count = 0;
        for(int i = 0;i < nums.length; i++){
            count+=MemoizationcombinationSum4(nums, target: target - nums[i],dp);
        }

        dp[target] = count;

        return dp[target];
    }


}
```

```
public int combinationSum4(int[] nums, int target) {

    int[] dp = new int[target+1];

    int ans2 = MemoizationcombinationSum4(nums,target,dp);

    return ans2;
}
```

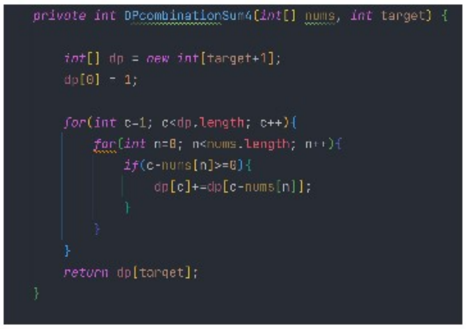
→ Again show TLE error ☺
Now Time to Make Optimize our Solution Tabulati

Tabulation

→ Now try to Optimized our Code using tabulation Concept



```
class Solution:

    # DP Approach

    1 usage (1 dynamic)
    def DPcombinationSum4(self, nums, target):

        dp = [0] * (target + 1)

        dp[0] = 1

        # traverse from target 1 to target
        for i in range(1, target + 1):

            # traverse on nums
            for j in range(len(nums)):

                if (i - nums[j]) >= 0:
                    dp[i] += dp[i - nums[j]]

        print(dp)
        return dp[target]

    2 usages
    def combinationSum4(self, nums: List[int], target: int) -> int:

        dp = [-1] * (target + 1)
        return self.MemoizationcombinationSum4(nums, target, dp)

        return self.DPcombinationSum4(nums, target)

        return self.DPcombinationSum4(nums, target)


ans = Solution()
arr = [1,2,5]
tar = 5
print(ans.combinationSum4(arr,tar))
```



```
private int DPcombinationSum4(int[] nums, int target) {

    int[] dp = new int[target+1];
    dp[0] = 1;

    for(int c=1; c<dp.length; c++){
        for(int n=0; n<nums.length; n++){
            if(c-nums[n]>=0){
                dp[c]+=dp[c-nums[n]];
            }
        }
    }
    return dp[target];
}
```