Combination Sum II – Find all unique combinations

In this article we will solve the most asked interview question "Combination Sum II – Find all unique combinations".

Problem Statement: 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.

Examples:

Example 1:

Input: candidates = [10,1,2,7,6,1,5], target = 8

Output:

[

[1,1,6],

[1,2,5],

[1,7],

[2,6]]

Explanation: These are the unique combinations whose sum is equal to target.

Example 2:

Input: candidates = [2,5,2,1,2], target = 5

Output: [[1,2,2],[5]]

Explanation: These are the unique combinations whose sum is equal to target.

Solution:

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1: Brute force

Approach : same as combination 1, but we use HashSet instead List

Code:

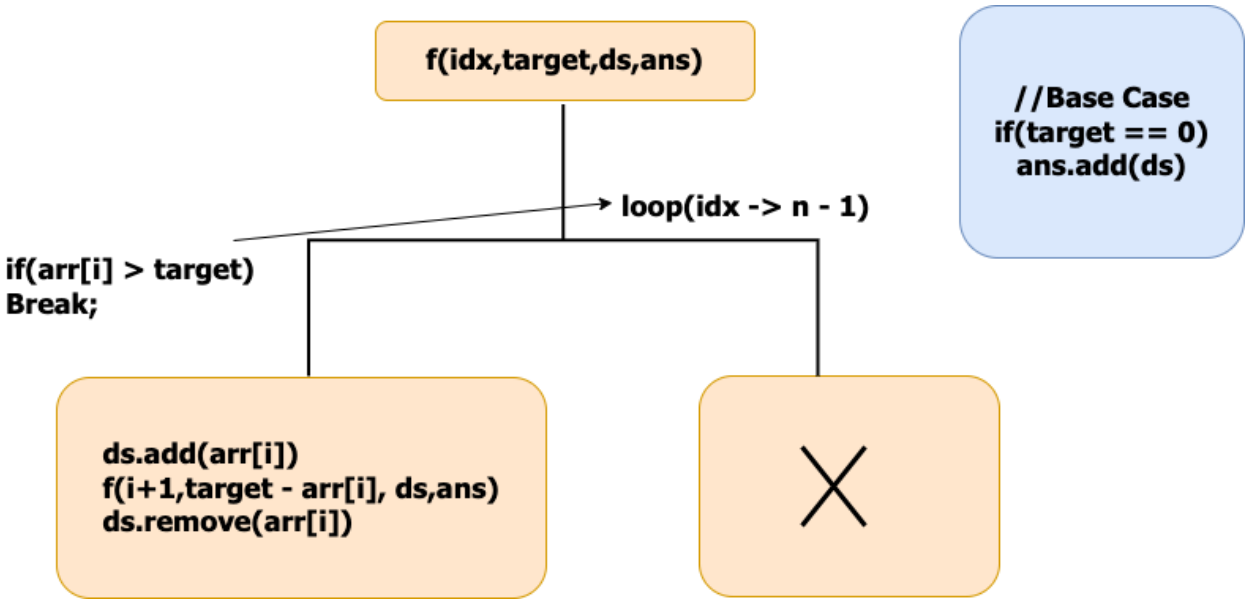
```
class Solution {

    public List<List<Integer>> combinationSum2(int[] candidates, int target)
{

        HashSet<List<Integer>> h=new HashSet<>();

        helper(0,candidates,target,h,new ArrayList<Integer>());

        return new ArrayList<>(h);

    }

    public void helper(int i, int[] a,int
sum,HashSet<List<Integer>>res,List<Integer> container)

    {

        if(i==a.length)

        {
```

```
            if(sum==0)
            {
                List<Integer> t=new ArrayList(container);
                Collections.sort(t);
                res.add(t);
            }
            return;
        }
        if(sum>=a[i])
        {
            container.add(a[i]);
            helper((i+1),a,(sum-a[i]),res,container);
            container.remove(container.size()-1);
        }
        helper((i+1),a,sum,res,container);
    }
}
```

Solution 2: Using extra space and time complexity

Approach:

Defining the Recursive Function:



Before starting the recursive call make sure to sort the elements because the ans should contain the combinations in sorted order and should not be repeated.

Initially, We start with the index 0, At index 0 we have n – 1 way **to pick the first element of our subsequence.**

Check if the current index value can be added to our ds. If yes add it to the ds and move the index by 1. while moving the index skip the consecutive repeated elements because they will form duplicate sequences.

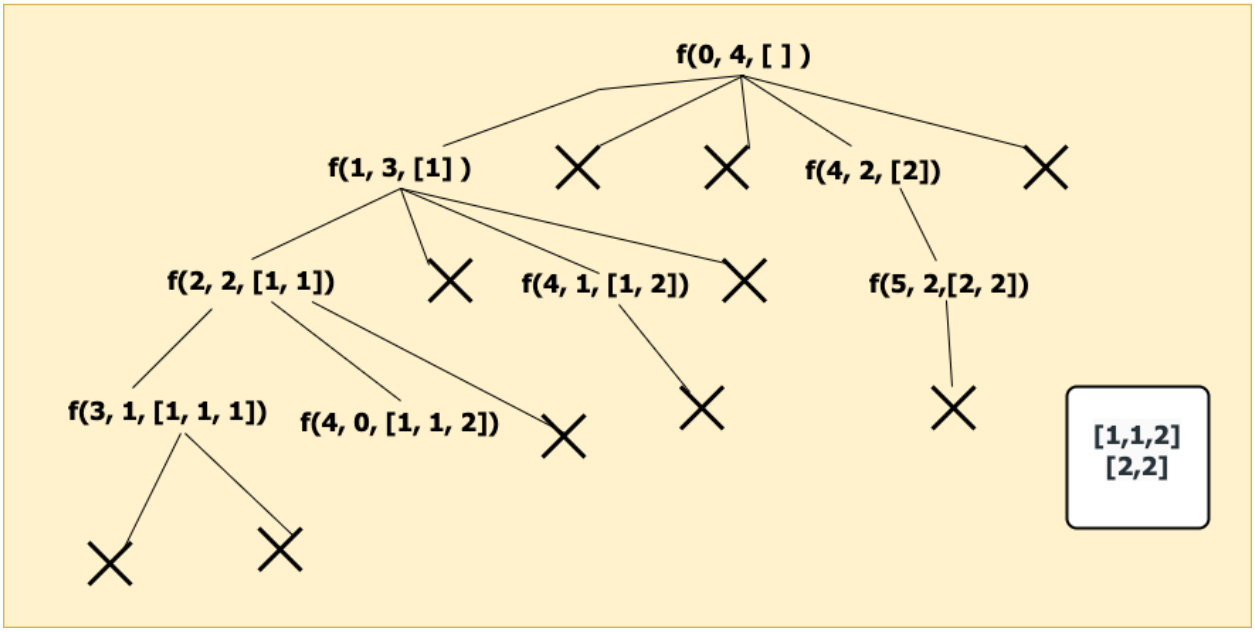
Reduce the target by arr[i],call the recursive call for f(idx + 1,target – 1,ds,ans) after the call make sure to pop the element from the ds.(By seeing the example recursive You will understand).

if(arr[i] > target) then terminate the recursive call because there is no use to check as the array is sorted in the next recursive call the index will be moving by 1 all the elements to its **right will be in increasing order.**

Base Condition:

Whenever the target value is zero add the ds to the ans return.

Representation of Recursive call for the example given below:



If we observe the recursive call for f(2,2,[1,1]) when it is returning the ds doesn't include 1 so make sure to remove it from ds after the call.

Code:

```
class Solution {
    public List<List<Integer>> combinationSum2(int[] candidates, int target)
    { Arrays.sort(candidates);
       List<List<Integer>> h=new ArrayList<>();

        helper(0,candidates,target,h,new ArrayList<Integer>());
       return h;
    }
    public void helper(int i, int[] a,int
sum,List<List<Integer>>res,List<Integer> container)
    {
```

```
            if(sum==0)
            {

                res.add(new ArrayList(container));
                 return;
            }


        for(int j=i;j<a.length;j++)
        {

           if(j>i&&a[j]==a[j-1]) continue;
            if(a[j]>sum) break;

            container.add(a[j]);
            helper(j+1,a,sum-a[j],res,container);
            container.remove(container.size()-1);
        }

    }
}
```

Time Complexity:O(2^n*k)

Reason: Assume if all the elements in the array are unique then the no. of subsequence you will get will be O(2^n). we also add the ds to our ans when we reach the base case that will take "k"//average space for the ds.

Space Complexity:O(k*x)

Reason: if we have x combinations then space will be x*k where k is the average length of the combination.