Combination Sum II

- You are given the array { 1 , 1 , 2, 1 , 2 } and you have to make the sum as 4.
- You can take each element only once and you have to print non redundant combinations lexicographically
- answer would be { 1 , 1 , 2 } and { 2 , 2 }

APPROACH I : Inspired by Combination Sum I

```
#include<bits/stdc++.h>
using namespace std;

void f(int index, int target, vector<int> &ds,int arr[], int n){
    // base case

    if( index == n){
        if( target == 0){
            for( auto it : ds){
                cout << it << " ";
            }
            cout << endl;
        }
        return;
    }

    // pick
    if(arr[index] <=  target){

        ds.push_back(arr[index]);
        f(index + 1, target - arr[index], ds, arr, n);

        ds.pop_back();

    }

    f( index + 1, target, ds, arr, n);

}


int main(){

    vector<int> arr = {1,1,2,1,2};
    int target = 4;
    vector<int> ds;
    sort(arr.begin(), arr.end() )

    f(0,4,ds,arr,5);
    return 0;

}
```

- In this case, we will be doing pick and not pick but here we know that we can pick one element only one time , so in the condition, even if we do not pick the element, we move to index + 1 and even if we pick we do index + 1.

- In this approach we will get the sum properly but there will be a lot of redundancies.

- The base case is such that, when you have reached the index n which is non existing index for a n sized array, you actually cant go further now and then you look at the target , if target has reduced to zero then the data structure formed is correct combination.

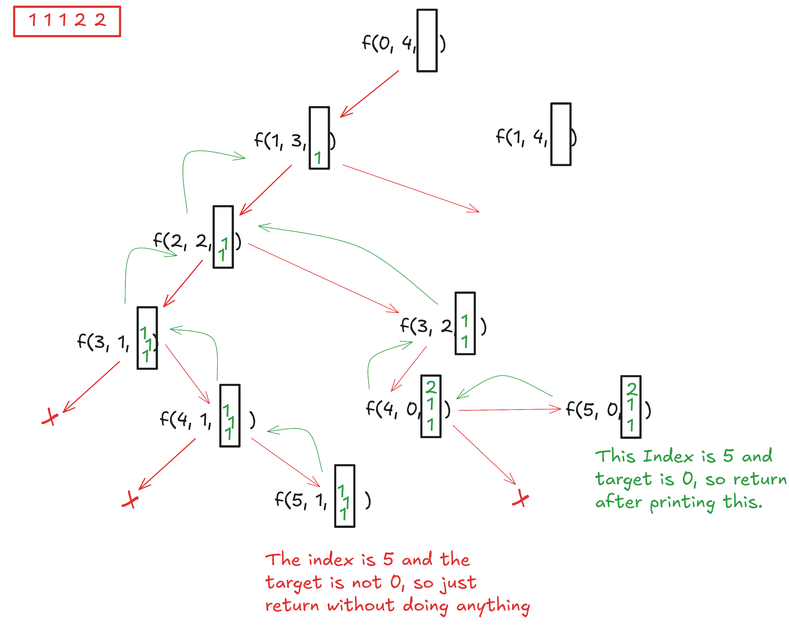
- When the base case is achieved you simple print or add the combination to another data structure and then simply return.

- The core logic is pick and not pick, if the element at that index is less than target, only then you can pick it else you cant pick it and then move onto the not pick.

- if the element is less than target then you can pick it and call further

- After this call is returned you need to unpick it and to unpick it properly you need to remove the picked element from the data structure.

- Then call the unpicking logic.



APPROACH II : Checking at each step

```
void fu(int index, int target, vector<int> ds, vector<int> arr, int n){
    // base case

    if( target == 0){
        for(auto it : ds){
            cout << it << " ";
        } cout << endl;

        return;
    }

    for( int i = index; i < n; i++){
        if( i > index && arr[i] == arr[i-1]){
            continue;
        }
        if( arr[i] > target ){
            break;
        }
        ds.push_back(arr[i]);
        fu( i + 1, target - arr[i], ds, arr, n);
        ds.pop_back();
    }
}

int main(){

    vector<int> arr = {1,1,1,2,2};
    int n = arr.size();
    int target =4 ;
    vector<int> ds;
    sort(arr.begin(), arr.end()); // This is the important step for your logic

    fu(0,target,ds,arr,n);


    return 0;
}
```

- The main logic is that when you are forming the combination you take all the elements of the array within a loop and do not take the element which you already taken
if(i > index && arr[i] == arr[i - 1]) continue : This ensures this thing

- Do not pick the element if it is greater than target, and as you have sorted your array you can simply break, if you just fond an element which is greater than target as obv subsequent ones will not be able to picked as they themselves would be greater than target

- you run the loop from the index you are currently in to the end.