

Subset Sum : Sum of all Subsets

Problem Statement: Given an array print all the sum of the subset generated from it, in the increasing order.

Examples:

Example 1:

Input: N = 3, arr[] = {5,2,1}

Output: 0,1,2,3,5,6,7,8

Explanation: We have to find all the subset's sum and print them. in this case the generated subsets are [[], [1], [2], [2,1], [5], [5,1], [5,2], [5,2,1], so the sums we get will be 0,1,2,3,5,6,7,8

Input: N=3, arr[] = {3,1,2}

Output: 0,1,2,3,3,4,5,6

Explanation: We have to find all the subset's sum and print them. in this case the generated subsets are [[], [1], [2], [2,1], [3], [3,1], [3,2], [3,2,1], so the sums we get will be 0,1,2,3,3,4,5,6

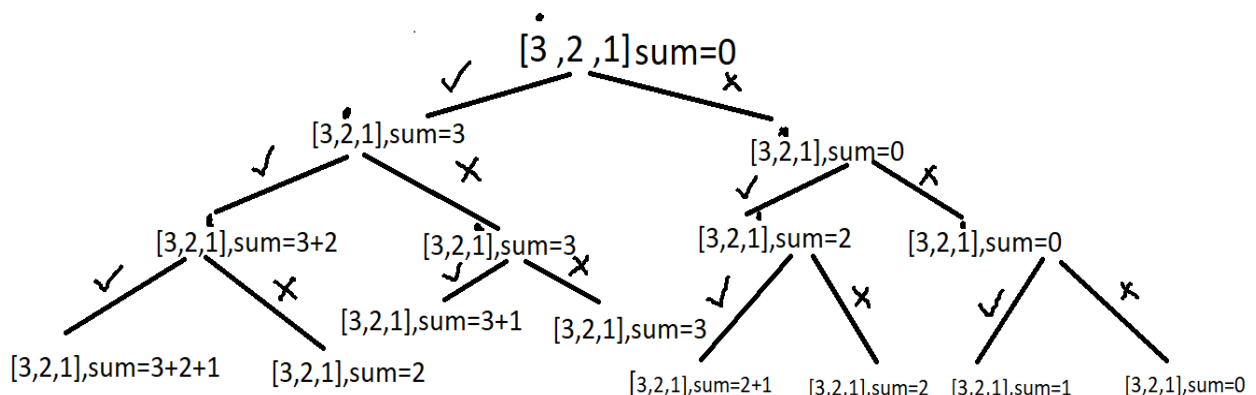
Solution

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

Solution 1: Using recursion

Intuition: The main idea is that on every index you have two options either to select the element to add it to your subset(pick) or not select the element at that index and move to the next index(non-pick).

Approach: Traverse through the array and for each index solve for two arrays, one where you pick the element, i.e add the element to the sum or don't pick and move to the next element, recursively, until the base condition. Here when you reach the end of the array is the base condition.



Code:

```
#include<bits/stdc++.h>

using namespace std;
class Solution {
public:
    void solve(int ind, vector < int > & arr, int n, vector < int > &
ans, int sum) {
        if (ind == n) {
            ans.push_back(sum);
            return;
        }
        //element is picked
        solve(ind + 1, arr, n, ans, sum + arr[ind]);
        //element is not picked
        solve(ind + 1, arr, n, ans, sum);
    }
    vector < int > subsetSums(vector < int > arr, int n) {
        vector < int > ans;
        solve(0, arr, n, ans, 0);
        return ans;
    }
};

int main() {
    vector < int > arr{3,1,2};
    Solution ob;
    vector < int > ans = ob.subsetSums(arr, arr.size());
    sort(ans.begin(), ans.end());
    cout<<"The sum of each subset is "<<endl;
    for (auto sum: ans) {
        cout << sum << " ";
    }
    cout << endl;

    return 0;
}
```

Output:

The sum of each subset is

0 1 2 3 3 4 5 6

Time Complexity: $O(2^n) + O(2^n \log(2^n))$. Each index has two ways. You can either pick it up or not pick it. So for n index time complexity for $O(2^n)$ and for sorting it will take $(2^n \log(2^n))$.

Space Complexity: $O(2^n)$ for storing subset sums, since 2^n subsets can be generated for an array of size n.

Subset – II | Print all the Unique Subsets

Problem Statement: Given an array of integers that **may contain duplicates** the task is to return all possible subsets. Return only **unique subsets** and they can be in any order.

Examples:

Example 1:

Input: array[] = [1,2,2]

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

Explanation: We can have subsets ranging from length 0 to 3. which are listed above. Also the subset [1,2] appears twice but is printed only once as we require only unique subsets.

Input: array[] = [1]

Output: [[], [1]]

Explanation: Only two unique subsets are available

Solution

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

Solution 1: Brute Force

Approach:

At every index, we make a decision whether to pick or not pick the element at that index. This will help us in generating all possible combinations but does not take care of the duplicates. Hence we will use a set to store all the combinations that will discard the duplicates.

Code:

```
#include <bits/stdc++.h>

using namespace std;
void printAns(vector < vector < int >> & ans) {
    cout << "The unique subsets are " << endl;
    cout << "[ ";
    for (int i = 0; i < ans.size(); i++) {
        cout << "[ ";
        for (int j = 0; j < ans[i].size(); j++)
            cout << ans[i][j] << " ";
        cout << " ]";
    }
    cout << " ]";
}
```

```
class Solution {
public:
    void fun(vector < int > & nums, int index, vector < int > ds, set
< vector < int >> & res) {
        if (index == nums.size()) {
            sort(ds.begin(), ds.end());
            res.insert(ds);
            return;
        }
        ds.push_back(nums[index]);
        fun(nums, index + 1, ds, res);
        ds.pop_back();
        fun(nums, index + 1, ds, res);
    }
    vector < vector < int >> subsetsWithDup(vector < int > & nums) {
        vector < vector < int >> ans;
        set < vector < int >> res;
        vector < int > ds;
        fun(nums, 0, ds, res);
        for (auto it = res.begin(); it != res.end(); it++) {
            ans.push_back( * it);
        }
        return ans;
    }
};

int main() {
    Solution obj;
    vector < int > nums = {1, 2, 2};
    vector < vector < int >> ans = obj.subsetsWithDup(nums);
    printAns(ans);
    return 0;
}
```

Output:

The unique subsets are

[[[1] [1 2] [1 2 2] [2] [2 2]]]

Time Complexity: $O(2^n * (k \log(x)))$. 2^n for generating every subset and $k \log(x)$ to insert every combination of average length k in a set of size x . After this, we have to convert the set of combinations back into a list of list /vector of vectors which takes more time.

Space Complexity: $O(2^n * k)$ to store every subset of average length k . Since we are initially using a set to store the answer another $O(2^n * k)$ is also used.

Solution 2: Optimal

Approach:

In the previous method, we were taking extra time to store the unique combination with the help of a set. To make the solution efficient we will have to decide on a method that will consider only the unique combinations without the help of additional data structure.

Lets understand with an example where arr = [1,2,2].

Initially start with an empty data structure. In the first recursion, call make a subset of size one, in the next recursion call a subset of size 2, and so on. But first, in order to make a subset of size one what options do we have?

We can pick up elements from either the first index or the second index or the third index.

However, if we have already picked up two from the second index, picking up two from the third index will make another duplicate subset of size one. Since we are trying to avoid duplicate subsets we can avoid picking up from the third index. This should give us an intuition that whenever there are duplicate elements in the array we pick up only the first occurrence.

The next recursion calls will continue from the point the previous one ended.

Let's summarize:-

- Sort the input array. Make a recursive function that takes the input array, the current subset, the current index and a list of list/ vector of vectors to contain the answer.
- Try to make a subset of size n during the nth recursion call and consider elements from every index while generating the combinations. Only pick up elements that are appearing for the first time during a recursion call to avoid duplicates.
- Once an element is picked up, move to the next index. The recursion will terminate when the end of array is reached. While returning backtrack by removing the last element that was inserted.

Code:

```
#include <bits/stdc++.h>

using namespace std;
void printAns(vector < vector < int >> & ans) {
    cout << "The unique subsets are " << endl;
    cout << "[ ";
    for (int i = 0; i < ans.size(); i++) {
        cout << "[ ";
        for (int j = 0; j < ans[i].size(); j++)
            cout << ans[i][j] << " ";
        cout << " ]";
    }
    cout << " ]";
}

class Solution {
private:
    void findSubsets(int ind, vector < int > & nums, vector < int >
& ds, vector < vector < int >> & ans) {
        ans.push_back(ds);
```

Recursion

```
        for (int i = ind; i < nums.size(); i++) {
            if (i != ind && nums[i] == nums[i - 1]) continue;
            ds.push_back(nums[i]);
            findSubsets(i + 1, nums, ds, ans);
            ds.pop_back();
        }
    }
public:
    vector < vector < int >> subsetsWithDup(vector < int > & nums)
{
    vector < vector < int >> ans;
    vector < int > ds;
    sort(nums.begin(), nums.end());
    findSubsets(0, nums, ds, ans);
    return ans;
}
};
int main() {
    Solution obj;
    vector < int > nums = {1,2,2 };
    vector < vector < int >> ans = obj.subsetsWithDup(nums);
    printAns(ans);
    return 0;
}
```

Output:

The unique subsets are

[[], [1], [1, 2], [1, 2, 2], [2], [2, 2]]

Time Complexity: $O(2^n)$ for generating every subset and $O(k)$ to insert every subset in another data structure if the average length of every subset is k . Overall $O(k * 2^n)$.

Space Complexity: $O(2^n * k)$ to store every subset of average length k . Auxiliary space is $O(n)$ if n is the depth of the recursion tree.

Combination Sum – 1

Problem Statement:

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

The same number may be chosen from the given array 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.

Examples:**Example 1:**

Input: array = [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.

Example 2:

Input: array = [2], target = 1

Output: []

Explanation: No combination is possible.

Solution

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

Solution: Recursion**Intuition:**

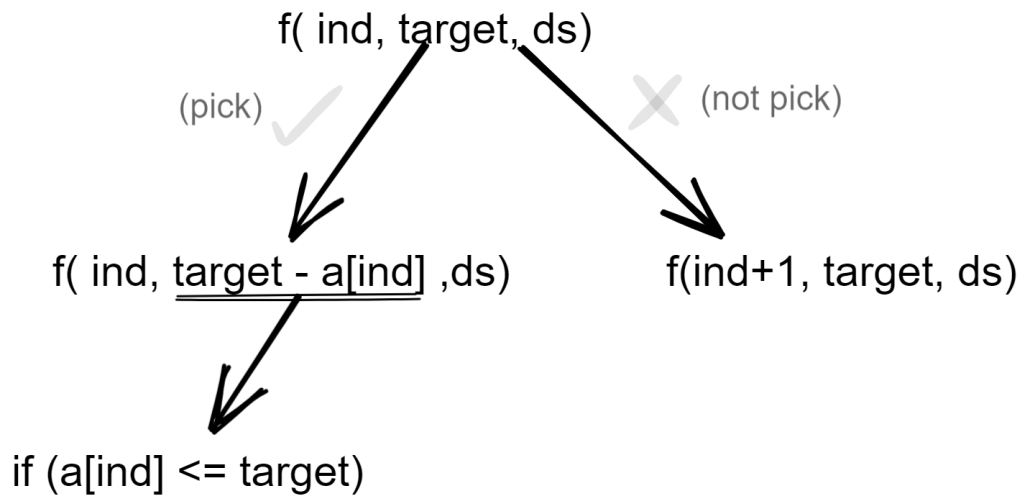
For questions like printing combinations or subsequences, the first thing that should strike your mind is recursion.

How to think recursively?

Whenever the problem is related to picking up elements from an array to form a combination, start thinking about the “**pick and non-pick**” approach.

Approach:

Defining recursive function:



Initially, the index will be 0, target as given and the data structure(vector or list) will be empty. Now there are 2 options viz to **pick or not pick** the current index element.

If you **pick** the element, again come back at the same index as multiple occurrences of the same element is possible so the target reduces to $\text{target} - \text{arr}[\text{index}]$ (where $\text{target} - \text{arr}[\text{index}] \geq 0$) and also insert the current element into the data structure.

If you decide **not to pick** the current element, move on to the next index and the target value stays as it is. Also, the current element is not inserted into the data structure.

While backtracking makes sure to pop the last element as shown in the recursion tree below.

Keep on repeating this process while $\text{index} < \text{size of the array}$ for a particular recursion call.

You can also stop the recursion when the target value is 0, but here a generalized version without adding too many conditions is considered.

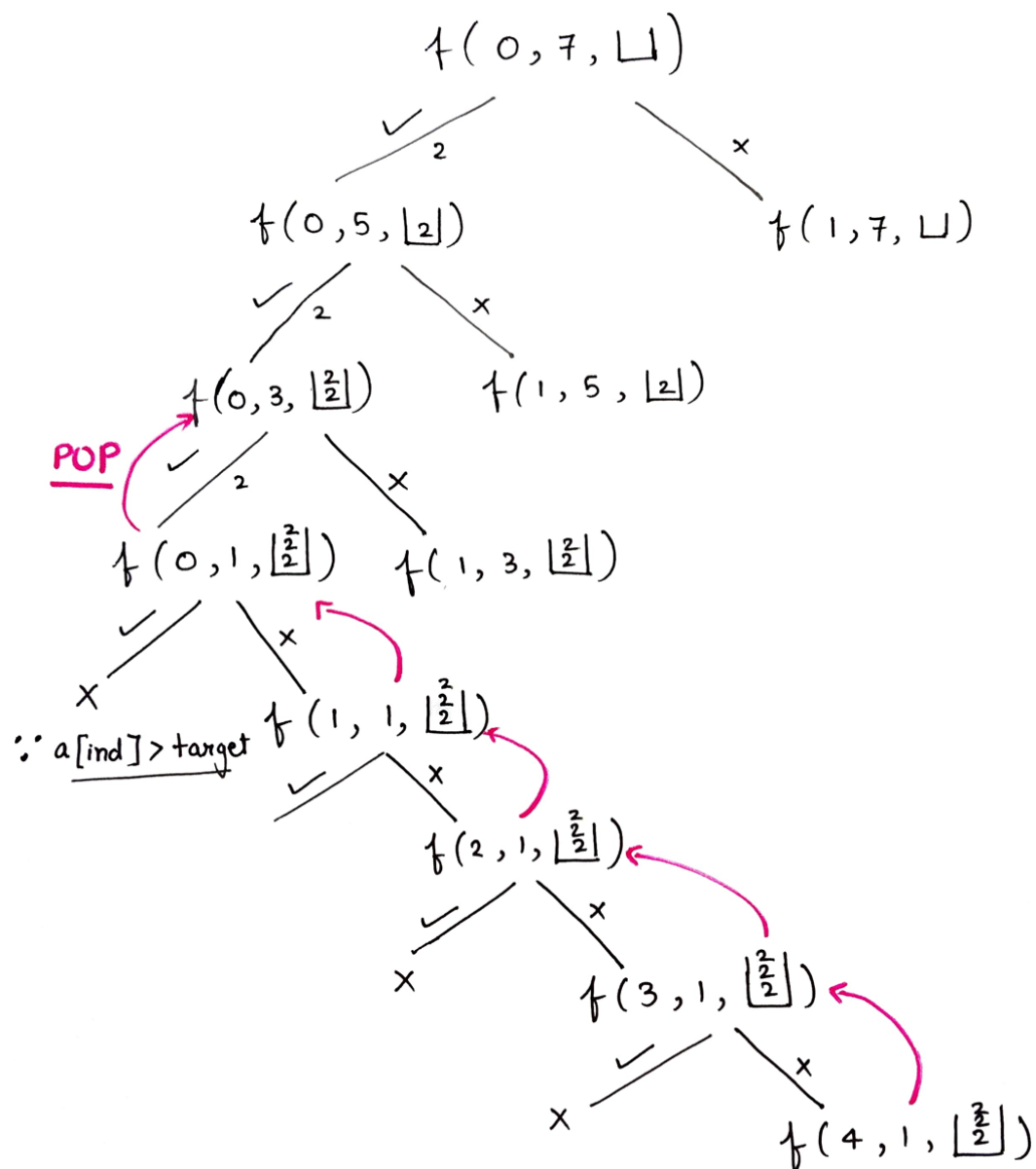
Using this approach, we can get all the combinations.

Base condition

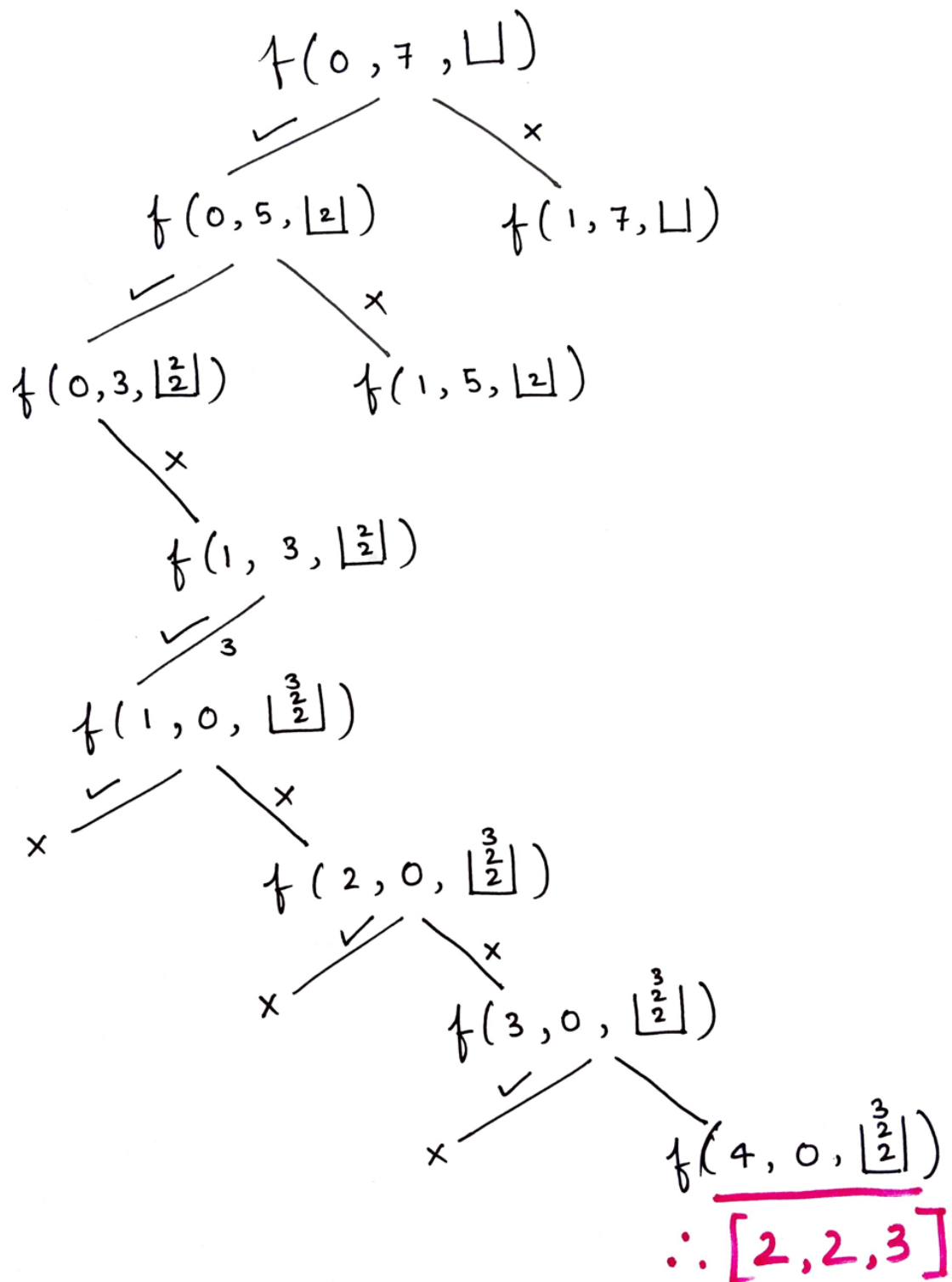
If $\text{index} == \text{size of array}$ and $\text{target} == 0$ include the combination in our answer

Diagrammatic representation for Example 1:

Case 1:



Case 2:



Code:

```
#include<bits/stdc++.h>
using namespace std;
class Solution {
public:
    void findCombination(int ind, int target, vector< int > & arr,
vector< vector< int >> & ans, vector< int > & ds) {
        if (ind == arr.size()) {
            if (target == 0) {
                ans.push_back(ds);
            }
            return;
        }
        // pick up the element
        if (arr[ind] <= target) {
            ds.push_back(arr[ind]);
            findCombination(ind, target - arr[ind], arr, ans, ds);
            ds.pop_back();
        }
        findCombination(ind + 1, target, arr, ans, ds);
    }
public:
    vector< vector< int >> combinationSum(vector< int > &
candidates, int target) {
        vector< vector< int >> ans;
        vector< int > ds;
        findCombination(0, target, candidates, ans, ds);
        return ans;
    }
};

int main() {
    Solution obj;
    vector< int > v {2,3,6,7};
    int target = 7;
    vector< vector< int >> ans = obj.combinationSum(v, target);
    cout << "Combinations are: " << endl;
    for (int i = 0; i < ans.size(); i++) {
        for (int j = 0; j < ans[i].size(); j++)
            cout << ans[i][j] << " ";
        cout << endl;
    }
}
```

Output:

Combinations are:

2 2 3

7

Time Complexity: $O(2^t * k)$ where t is the target, k is the average length

Reason: Assume if you were not allowed to pick a single element multiple times, every element will have a couple of options: pick or not pick which is 2^n different recursion calls, also assuming that the average length of every combination generated is k . (to put length k data structure into another data structure)

Why not (2^n) but (2^t) (where n is the size of an array)?

Assume that there is 1 and the target you want to reach is 10 so 10 times you can “pick or not pick” an element.

Space Complexity: $O(k * x)$, k is the average length and x is the no. of combinations

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.

Recursion

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.

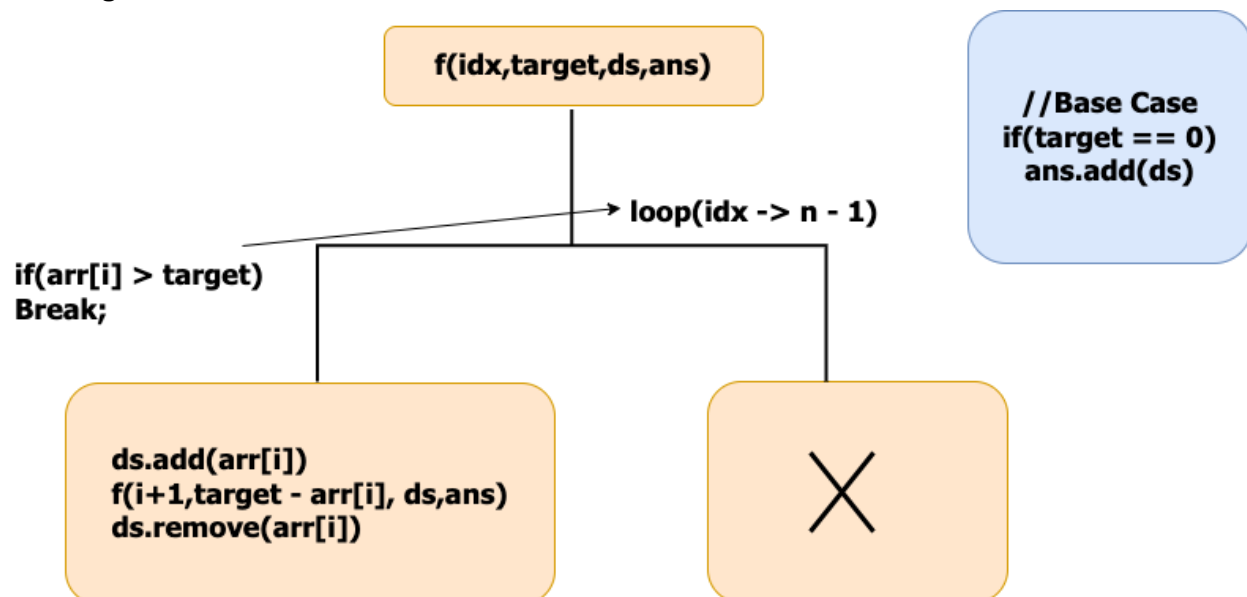
Solution:

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

Solution 1: 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).

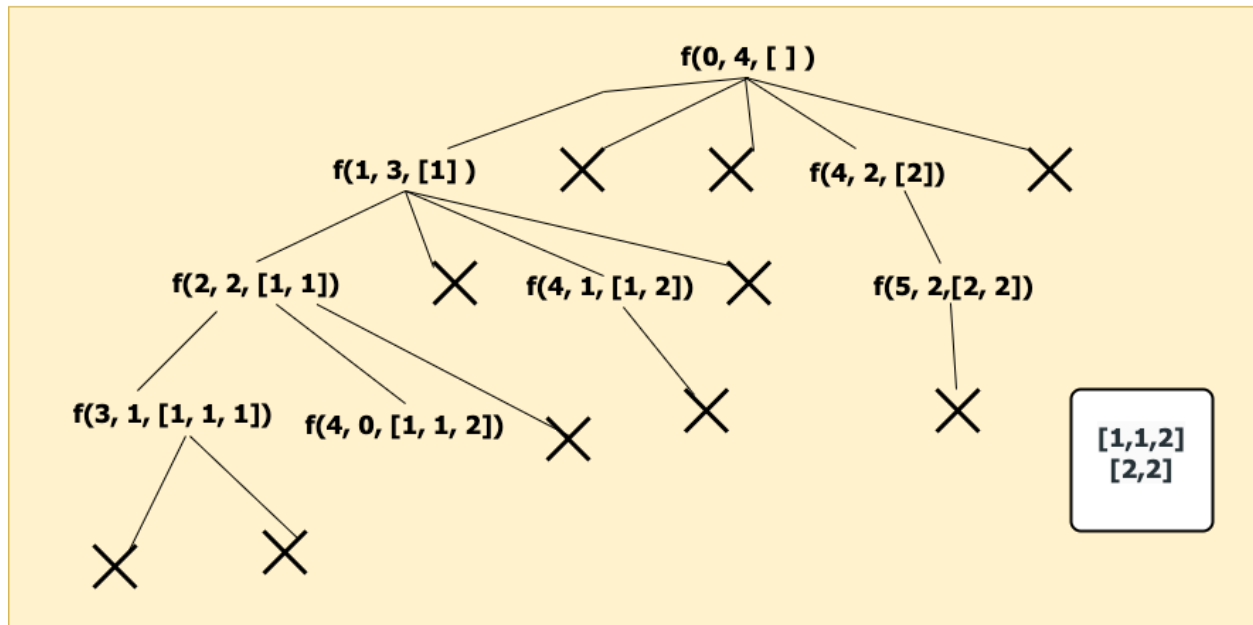
`if(arr[i] > target)` then terminate the recursive call because there is no use to check as the array is sorted in the next recursive call the index will be moving by 1 all the elements to its **right will be in increasing order.**

Base Condition:

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

Representation of Recursive call for the example given below:

0 1 2 3 4
arr[] = [1, 1, 1, 2, 2] target = 4



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:

```
#include<bits/stdc++.h>

using namespace std;
void findCombination(int ind, int target, vector < int > & arr,
vector < vector < int >> & ans, vector < int > & ds) {
    if (target == 0) {
        ans.push_back(ds);
        return;
    }
    for (int i = ind; i < arr.size(); i++) {
        if (i > ind && arr[i] == arr[i - 1]) continue;
        if (arr[i] > target) break;
        ds.push_back(arr[i]);
        findCombination(i + 1, target - arr[i], arr, ans, ds);
        ds.pop_back();
    }
}
```

```
vector < vector < int >> combinationSum2(vector < int > & candidates,
int target) {
    sort(candidates.begin(), candidates.end());
    vector < vector < int >> ans;
    vector < int > ds;
    findCombination(0, target, candidates, ans, ds);
    return ans;
}

int main() {
    vector<int> v{10,1,2,7,6,1,5};
    vector < vector < int >> comb = combinationSum2(v, 8);
    cout << "[ ";
    for (int i = 0; i < comb.size(); i++) {
        cout << "[ ";
        for (int j = 0; j < comb[i].size(); j++) {
            cout << comb[i][j] << " ";
        }
        cout << " ]";
    }
    cout << " ]";
}
```

Output:

```
[[ 1 1 6 ][ 1 2 5 ][ 1 7 ][ 2 6 ]]
```

Time Complexity: $O(2^n \cdot k)$

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: $O(k \cdot x)$

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

Palindrome Partitioning

Problem Statement: You are given a string s , partition it in such a way that every substring is a palindrome. Return all such palindromic partitions of s .

Note: A palindrome string is a string that reads the same backward as forward.

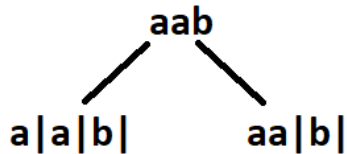
Examples:

Example 1:

Input: $s = \text{"aab"}$

Output: $[[\text{"a"}, \text{"a"}, \text{"b"}], [\text{"aa"}, \text{"b"}]]$

Explanation: The first answer is generated by making three partitions. The second answer is generated by making two partitions.

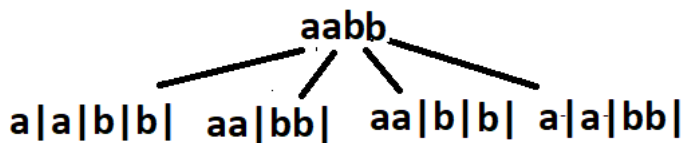


Example 2:

Input: s = "aabb"

Output: [["a","a","b","b"], ["aa","bb"], ["a","a","bb"], ["aa","b","b"]]

Explanation: See Figure



Solution

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

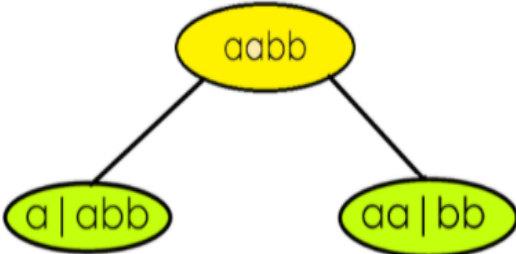
Approach: The initial idea will be to make partitions to generate substring and check if the substring generated out of the partition will be a palindrome. Partitioning means we would end up generating every substring and checking for palindrome at every step. Since this is a repetitive task being done again and again, at this point we should think of recursion. The recursion continues until the entire string is exhausted. After partitioning, every palindromic substring is inserted in a data structure. When the base case has reached the list of palindromes generated during that recursion call is inserted in a vector of vectors/list of list.

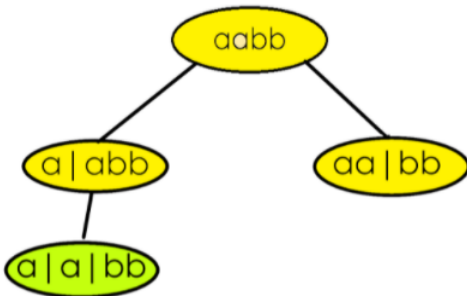
We have already discussed the initial thought process and the basic outline of the solution. The approach will get clearer with an example.

Say s = "aabb" and assume indexes of string characters to be 0-based. For a better understanding, we have divided recursion into some steps.

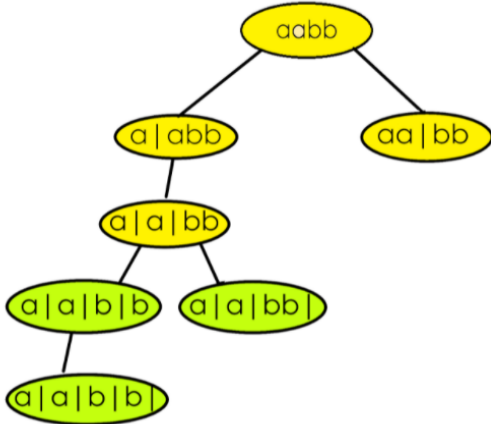
STEP 1: We consider substrings starting from the 0th index. [0,0] is a palindrome, so partition right after the 0th index. [0,1] is another palindrome, make a partition after 1st index. Beyond this point, other substrings starting from index 0 are "aab" and "aabb". These are not palindromes, hence no more partitions are possible. The strings remaining on the right side of the partition are used as input to make recursive calls.

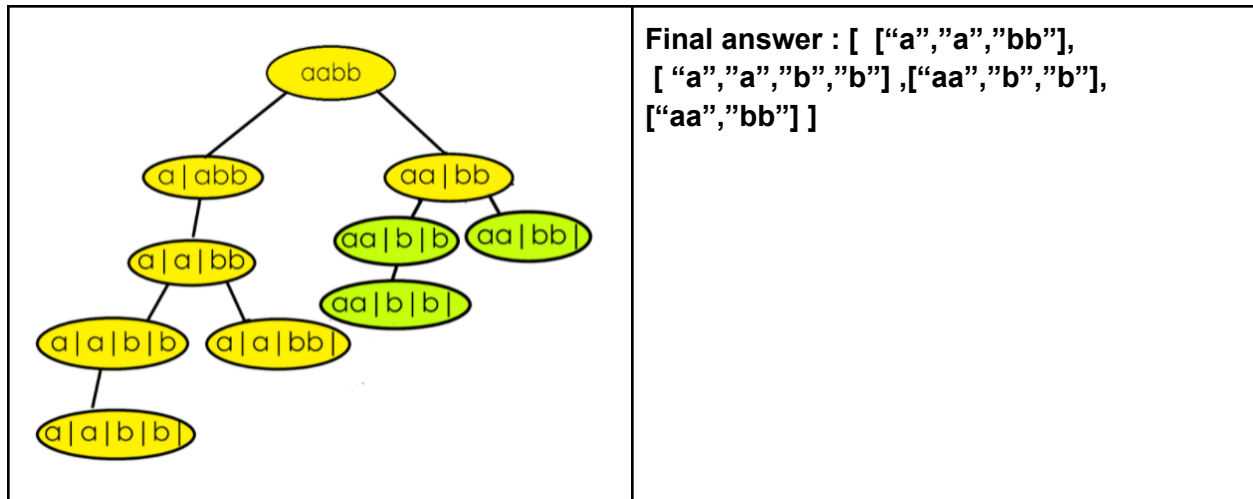
Recursion

	<p>STEP 2: Consider the recursive call on the left(refer to image) where “abb” is the input. [1,1] is a palindrome, make a partition after it. [1,2] and [1,3] are not palindromes.</p>
---	---

	<p>STEP 3: Here “bb” is the input. [2,2] as well as [2,3] are palindromes. Make one partition after the 2nd index and one after the 3rd index The entire string is exhausted after the 3rd index, so the right recursion ends here. Palindromes generated from the right recursion are inserted in our answer.</p>
---	---

Our answer at this point :[“a”, ”a”, ”bb”]

	<p>The left recursion will continue with “b” as its input. [3,3] is a palindrome so one last partition for the left recursion is made after the 3rd index. Insert the palindromes. ans = [[“a”, ”a”, ”bb”], [“a”, ”a”, ”b”, ”b”]] STEP 4: After the list of palindromic substrings are returned from the left recursive call, continue the same process for the call on the right that was left to recur. The right recursion is having “bb” as input, something we have already encountered in step 3. Hence we will repeat the same task which was done in step 3 onwards.</p>
---	--



Code:

```
#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    vector < vector < string >> partition(string s) {
        vector < vector < string > > res;
        vector < string > path;
        func(0, s, path, res);
        return res;
    }
    void func(int index, string s, vector < string > & path,
        vector < vector < string > > & res) {
        if (index == s.size()) {
            res.push_back(path);
            return;
        }
        for (int i = index; i < s.size(); ++i) {
            if (isPalindrome(s, index, i)) {
                path.push_back(s.substr(index, i - index + 1));
                func(i + 1, s, path, res);
                path.pop_back();
            }
        }
    }
    bool isPalindrome(string s, int start, int end) {
        while (start <= end) {
            if (s[start++] != s[end--])
                return false;
        }
        return true;
    }
};
```

```
int main() {
    string s = "aabb";
    Solution obj;
    vector < vector < string >> ans = obj.partition(s);
    int n = ans.size();
    cout << "The Palindromic partitions are :-" << endl;
    cout << " [ ";
    for (auto i: ans) {
        cout << "[ ";
        for (auto j: i) {
            cout << j << " ";
        }
        cout << "] ";
    }
    cout << " ]";

    return 0;
}
```

Output:

The Palindromic partitions are :-

[[a a b b][a a bb][aa b b][aa bb]]

Time Complexity: $O(2^n * k * (n/2))$

Reason: $O(2^n)$ to generate every substring and $O(n/2)$ to check if the substring generated is a palindrome. $O(k)$ is for inserting the palindromes in another data structure, where k is the average length of the palindrome list.

Space Complexity: $O(k * x)$

Reason: The space complexity can vary depending upon the length of the answer. k is the average length of the list of palindromes and if we have x such list of palindromes in our final answer. The depth of the recursion tree is n , so the auxiliary space required is equal to the $O(n)$.

Find K-th Permutation Sequence

Problem Statement: Given N and K , where N is the sequence of numbers from 1 to N ($[1, 2, 3, \dots, N]$) find the **Kth permutation sequence**.

For $N = 3$ the $3!$ Permutation sequences in order would look like this:-

$K = 1$	"123"
$K = 2$	"132"
$K = 3$	"213"
$K = 4$	"231"
$K = 5$	"312"
$K = 6$	"321"

Note: $1 \leq K \leq N!$ Hence for a given input its Kth permutation always exists

Examples:

Example 1:

Input: $N = 3, K = 3$

Output: "213"

Explanation: The sequence has $3!$ permutations as illustrated in the figure above. $K = 3$ corresponds to the third sequence.

Example 2:

Input: $N = 3, K = 5$

Result: "312"

Explanation: The sequence has $3!$ permutations as illustrated in the figure above. $K = 5$ corresponds to the fifth sequence.

Solution

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

Solution 1: Brute Force Solution

Approach:

The extreme naive solution is to generate all the possible permutations of the given sequence. This is achieved using recursion and every permutation generated is stored in some other data structure (here we have used a vector). Finally, we sort the data structure in which we have stored all the sequences and return the Kth sequence from it.

Code:

```
#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    //function to generate all possible permutations of a string
    void solve(string & s, int index, vector < string > & res) {
        if (index == s.size()) {
            res.push_back(s);
            return;
        }
        for (int i = index; i < s.size(); i++) {
            swap(s[i], s[index]);
            solve(s, index + 1, res);
            swap(s[i], s[index]);
        }
    }
    string getPermutation(int n, int k) {
        string s;
        vector < string > res;
        //create string
        for (int i = 1; i <= n; i++) {
            s.push_back(i + '0');
        }
        solve(s, 0, res);
        //sort the generated permutations
        sort(res.begin(), res.end());
        //make k 0-based indexed to point to kth sequence
        auto it = res.begin() + (k - 1);
        return *it;
    }
};

int main() {
    int n = 3, k = 3;
    Solution obj;
    string ans = obj.getPermutation(n, k);
    cout << "The Kth permutation sequence is " << ans << endl;

    return 0;
}
```

Output:

The Kth permutation sequence is 213

Time complexity: $O(N! * N) + O(N! \log N!)$

Recursion

Reason: The recursion takes $O(N!)$ time because we generate every possible permutation and another $O(N)$ time is required to make a deep copy and store every sequence in the data structure. Also, $O(N! \log N!)$ time required to sort the data structure

Space complexity: $O(N)$

Reason: Result stored in a vector, we are auxiliary space taken by recursion

Solution 2:(Optimal Approach)

Say we have $N = 4$ and $K = 17$. Hence the number sequence is $\{1,2,3,4\}$.

Intuition:

Since this is a permutation we can assume that there are four positions that need to be filled using the four numbers of the sequence. First, we need to decide which number is to be placed at the first index. Once the number at the first index is decided we have three more positions and three more numbers. Now the problem is shorter. We can repeat the technique that was used previously until all the positions are filled. The technique is explained below.

Approach:

STEP 1:

Mathematically speaking there can be 4 variations while generating the permutation. We can have our permutation starting with either 1 or 2 or 3 or 4. If the first position is already occupied by one number there are three more positions left. The remaining three numbers can be permuted among themselves while filling the 3 positions and will generate $3! = 6$ sequences. Hence each block will have 6 permutations adding up to a total of $6 \times 4 = 24$ permutations. If we consider the sequences as 0-based and in the sorted form we observe:-

- The **0th – 5th permutation** will start with 1
- The **6th – 11th permutation** will start with 2
- The **12th – 17th permutation** will start with 3
- The **18th – 23rd permutation** will start with 4.

(For better understanding refer to the picture below.)

BLOCK NO.	NUMBER AT 1ST POSITION	REMAINING NUMBERS
0TH BLOCK	1	{2,3,4}
1ST BLOCK	2	{1,3,4}
2ND BLOCK	3	{1,2,4}
3RD BLOCK	4	{1,2,3}

We make **$K = 17-1$ considering 0-based indexing**. Since each of the four blocks illustrated above comprises 6 permutations, therefore, the 16th permutation will lie in $(16 / 6) = 2$ nd block, and our answer is the $(16 \% 6) = 4$ th sequence from the 2nd block. Therefore 3 occupies the first position of the sequence and **$K = 4$** .

3

STEP 2:

Our new search space comprises three elements {1,2,4} where $K = 4$. Using the previous technique we can consider the second position to be occupied can be any one of these 3 numbers. Again one block can start with 1, another can start with 2 and the last one can start with 4. Since one position is fixed, the remaining two numbers of each block can form $2! = 2$ sequences. In sorted order :

- The **0th – 1st sequence** starts with 1
- The **2nd – 3rd sequence** starts with 2
- The **4th – 5th sequence** starts with 4

BLOCK NO.	NUMBER AT 1ST POSITION	REMAINING NUMBERS
0TH BLOCK	1	{2,4}
1ST BLOCK	2	{1,4}
2ND BLOCK	4	{1,2}

The 4th permutation will lie in $(4/2) = 2\text{nd block}$ and our answer is the $4\%2 = 0\text{th sequence}$ from the 2nd block. Therefore 4 occupies the second position and $K = 0$.

3 4

STEP 3:

The new search space will have two elements {1,2} and $K = 0$. One block starts with 1 and the other block starts with 2. The other remaining number can form only one $1! = 1$ sequence. In sorted form –

- The **0th sequence** starts with 1
- The **1st sequence** starts with 2

BLOCK NO.	NUMBER AT 1ST POSITION	REMAINING NUMBERS
0TH BLOCK	1	{2}
1ST BLOCK	2	{1}

The 0th permutation will lie in the $(0/1) = 0\text{th block}$ and our answer is the $0\%1 = 0\text{th sequence}$ from the 0th block. Therefore 1 occupies the 3rd position and $K = 0$.

3 4 1

STEP 4:

Now the only block has 2 in the first position and no **remaining number is present**.

BLOCK NO.	NUMBER AT 1ST POSITION	REMAINING NUMBERS
0Th BLOCK	2	{}

This is the point where we place 2 in the last position and stop.

3 4 1 2

The final answer is 3412.

Code:

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    string getPermutation(int n, int k) {
        int fact = 1;
        vector < int > numbers;
        for (int i = 1; i < n; i++) {
            fact = fact * i;
            numbers.push_back(i);
        }
        numbers.push_back(n);
        string ans = "";
        k = k - 1;
        while (true) {
            ans = ans + to_string(numbers[k / fact]);
            numbers.erase(numbers.begin() + k / fact);
            if (numbers.size() == 0) {
                break;
            }

            k = k % fact;
            fact = fact / numbers.size();
        }
    }
};
```



```
        return ans;
    }
};

int main() {
    int n = 3, k = 3;
    Solution obj;
    string ans = obj.getPermutation(n, k);
    cout << "The Kth permutation sequence is " << ans << endl;

    return 0;
}
```

Output:

The Kth permutation sequence is 213

Time Complexity: $O(N) * O(N) = O(N^2)$

Reason: We are placing N numbers in N positions. This will take $O(N)$ time. For every number, we are reducing the search space by removing the element already placed in the previous step. This takes another $O(N)$ time.

Space Complexity: $O(N)$

Reason: We are storing the numbers in a data structure (here vector)