<https://leetcode.com/problems/subsets-ii/>

Given an integer array nums that may contain duplicates, return *all possible subsets (the power set)*.

The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

**Example 1:**

Input: nums = [1,2,2]

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

**Example 2:**

Input: nums = [0]

Output: [[],[0]]

**Constraints:**

1 <= nums.length <= 10

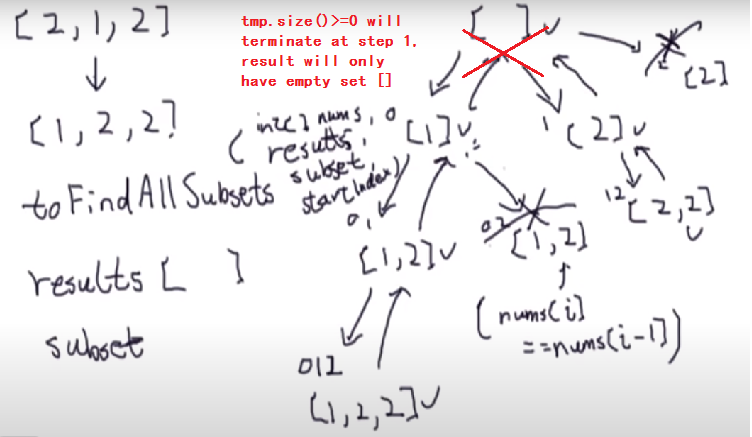
-10 <= nums[i] <= 10

**Attempt 1: 2022-10-8**

**Wrong Solution:**

**Adding unnecessary limitation for "result.add(new ArrayList<Integer>(tmp));" and then direct return, no limitation required because we have to collect all subsets, the direct return will terminate the search at early stage, the recursion will not continue and miss subsets**

**1. Wrong limitation with if(tmp.size() >= 0) {... return}**



Input: [1,2,2]

Wrong output: [[]]

Expect output: [[], [1], [1, 2], [1, 2, 2], [2], [2, 2]]

class Solution {

    public List<List<Integer>> subsetsWithDup(int[] nums) {

        List<List<Integer>> result = new ArrayList<List<Integer>>();

        Arrays.sort(nums);

        helper(nums, result, new ArrayList<Integer>(), 0);

        return result;

    }

    private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, int index) {

        // Unnecessary limitation and direct return terminate recursion at early stage

        if(tmp.size() >= 0) {

            result.add(new ArrayList<Integer>(tmp));

            return;

        }

        for(int i = index; i < nums.length; i++) {

            if(i > index && nums[i] == nums[i - 1]) {

                continue;

            }

            tmp.add(nums[i]);

            helper(nums, result, tmp, i + 1);

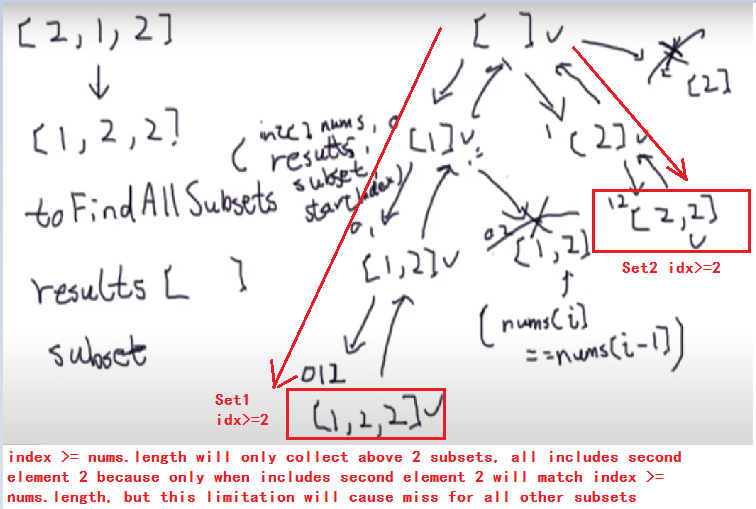
            tmp.remove(tmp.size() - 1);

        }

    }

}

**2. Wrong limitation with if(index  >= nums.length) {... return}**



Input: [1,2,2]

Wrong output: [[1, 2, 2], [2, 2]]

Expect output: [[], [1], [1, 2], [1, 2, 2], [2], [2, 2]]

class Solution {

    public List<List<Integer>> subsetsWithDup(int[] nums) {

        List<List<Integer>> result = new ArrayList<List<Integer>>();

        Arrays.sort(nums);

        helper(nums, result, new ArrayList<Integer>(), 0);

        return result;

    }

    private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, int index) {

        // Unnecessary limitation and direct return terminate recursion at early stage

        if(index >= nums.length) {

            result.add(new ArrayList<Integer>(tmp));

            return;

        }

        for(int i = index; i < nums.length; i++) {

            if(i > index && nums[i] == nums[i - 1]) {

                continue;

            }

            tmp.add(nums[i]);

            helper(nums, result, tmp, i + 1);

            tmp.remove(tmp.size() - 1);

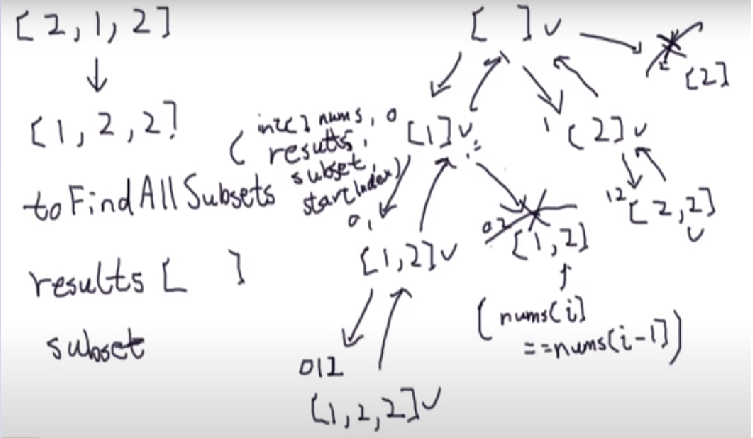
        }

    }

}

**Solution 1: Backtracking style 1 (10min)**

**No limitation on "result.add(new ArrayList<Integer>(tmp))"**



class Solution {

    public List<List<Integer>> subsetsWithDup(int[] nums) {

        List<List<Integer>> result = new ArrayList<List<Integer>>();

        Arrays.sort(nums);

        helper(nums, result, new ArrayList<Integer>(), 0);

        return result;

    }

    private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, int index) {

        result.add(new ArrayList<Integer>(tmp));

        for(int i = index; i < nums.length; i++) {

// The condition to filter out duplicate combination is elegant, like here

// 'i > index' only will happen after previous 'tmp.remove(tmp.size() - 1)'

// and iterate in for loop again, the previous 'tmp.remove(tmp.size() - 1)'

// means backtracking happen when 'index'th level of recursion ended and back

// to 'index - 1'th level of recursion, and iterate in for loop again means

// after backtracking which remove the last element in combination and attempt

// on adding new number as new last element into combination, and logically it

// sort out now, if the new number equal to removed last element in previous

// combination before backtracking, then the new combination will eqaul to

// previous combination before backtracking, then they are duplicate, we have

// to skip, test with {1, 2, 2} will be clear

            if(i > index && nums[i] == nums[i - 1]) {

                continue;

            }

            tmp.add(nums[i]);

            helper(nums, result, tmp, i + 1);

            tmp.remove(tmp.size() - 1);

        }

    }

}

Time complexity: O(N×2^N) to generate all subsets and then copy them into output list.

Space complexity: O(N). We are using O(N) space to maintain curr, and are modifying curr in-place with

backtracking. Note that for space complexity analysis, we do not count space that is only used for the

purpose of returning output, so the output array is ignored.

**Why in condition " if(i > index && nums[i] == nums[i - 1])" we have "i > index" ?**

**Refer to chatGPT**

The condition if(i > index && nums[i] == nums[i - 1]) is used to avoid generating duplicate subsets. Here's an explanation of each part of this condition:

**i > index**

**This ensures that we only skip duplicate elements within the same recursive level**. **In other words, it prevents skipping the first occurrence of a duplicate element but skips any subsequent duplicate elements at the same level of recursion.**

For example:

When index == 0, the loop starts at i = 0, and the first occurrence of nums[i] is not skipped, even if it's a duplicate of the next element.

When index > 0 and i > index, it ensures that if two identical numbers are encountered consecutively, only the first one is considered for the subset, and the others are skipped.

**nums[i] == nums[i - 1]**

This part checks whether the current element nums[i] is the same as the previous element nums[i - 1]. If they are the same, we skip nums[i] to prevent generating duplicate subsets.

**Why Is i > index Necessary?**

The condition i > index is necessary to avoid skipping the **first occurrence** of a duplicate element at the current recursion level.

Without **i > index:** The first occurrence of any duplicate element would also be skipped, which is incorrect because we want to include the first occurrence of each element but skip subsequent duplicate elements within the same recursion branch.

For instance, if nums = [1, 2, 2], we want to include both [1, 2] and [1, 2, 2], but without i > index, we might skip the first occurrence of 2 entirely, which would incorrectly exclude these subsets.

With **i > index:** The check ensures that only the **second** and subsequent duplicate elements are skipped, but the first occurrence is still considered in the subset generation.

**Example**

Consider the input array nums = [1, 2, 2]:

When index = 0, the first recursive call considers all elements, so it generates subsets including the first occurrence of 2.

In the next recursive step, where index = 1, the loop starts with i = 1. The first 2 is added, and the subset [1, 2] is generated. The loop continues, and when i = 2, the second 2 is skipped because nums[2] == nums[1] and i > index.

As a result, the duplicate subset [1, 2] is not generated twice.

**Summary of the Logic:**

i > index**:** Ensures that we only skip subsequent duplicates but still process the first occurrence of any number.

nums[i] == nums[i - 1]**:** Prevents generating duplicate subsets by skipping elements that are identical to the one that has already been processed at the current recursion level.

By combining these conditions, the algorithm avoids generating duplicate subsets while still considering all valid combinations.

**Refer to**

<https://algo.monster/liteproblems/90>

**Problem Description**

The task is to return all possible subsets (also known as the power set) of a given integer array nums, which may contain duplicates. The important constraint is that the solution set must not include duplicate subsets. These subsets can be returned in any order. A subset is a set containing elements that are all found in another set, which in this case is the array nums. **The power set is the set of all subsets including the empty set and the set itself**.

For example, if nums is [1,2,2], the possible unique subsets without considering order are:

1- []

2- [1]

3- [2]

4- [1,2]

5- [2,2]

6- [1,2,2]

The challenge here is to ensure that while generating the subsets, duplicates are not created.

**Intuition**

The core idea to avoid duplicates in the subsets is to sort the array nums. Sorting brings identical elements next to each other, making it easier to avoid duplicates when constructing the subsets.

The solution uses Depth-First Search (DFS) to explore all the possible subsets. The process starts with an empty subset, and at each level, it includes the next number from the array into the current subset, then recursively continues to add the remaining numbers to form new subsets. This is done until all numbers have been considered. After each number is processed for a given subset, it is removed again (this operation is also known as backtracking), and the next number is tried.

To ensure no duplicated subsets are generated, there are two conditions:

**Sort the Numbers:** As mentioned, we start by sorting the array to bring duplicates together.

**Skip over duplicates:** While generating subsets, if the current element is the same as the previous element and the previous element was not included in the current subset being generated (checked by i != u), it is skipped to avoid creating a subset that has already been created.

This way, the dfs function systematically constructs all subsets, but avoids any repeats, ensuring that the power set it returns is free from duplicate subsets.

**Solution Approach**

The implementation of the solution uses recursion to generate the subsets and [backtracking](https://algo.monster/problems/backtracking) to make sure all possible subsets are considered. Here's how the implementation works, with reference to the given Python code above:

**Sort the Array:** The first thing the solution does is to sort nums. This is crucial because it ensures that duplicates are adjacent and can be easily skipped when generating subsets.

nums.sort()

**Depth-First Search (DFS):** The dfs function is then defined. DFS is a common algorithm for exploring all the possible configurations of a given problem. In this case, it's used to generate all possible subsets by making a series of choices (to include or not include an element in the current subset).

**Recursive Exploration:** Inside dfs, the current subset t is first included in the answer set ans. This represents the choice of not including any more elements in the current subset.

ans.append(t[:])

**Iterative Inclusion:** The function then iterates through the remaining numbers starting from index u. **If the number at the current index is the same as the one before it and it's not the first iteration of the loop (checked by i != u), it skips adding this number to avoid duplicates.**

for i in range(u, len(nums)):

if i != u and nums[i] == nums[i - 1]:

continue

t.append(nums[i])

dfs(i + 1, t)

t.pop()

t.append(nums[i]) adds the current number to the subset.

A recursive call dfs(i + 1, t) is made to consider the next number in the array. The index i + 1 ensures that the function does not consider the current number again, thus moving forward in the array.

t.pop() is the backtracking step, which removes the last added number from the subset, allowing the function to consider other possibilities after returning from the recursive call.

Global Answer Set: An empty list ans is initialized outside the dfs function. As the subsets are created, they are added to ans. Since ans is defined outside the scope of the dfs function, it retains all the solutions across recursive calls.

ans = []

Kick Start the DFS: The dfs function is initially called with the starting index 0 and an empty subset [] to start the exploration.

dfs(0, [])

At the end of the execution, the global ans contains all the subsets without duplicates, which is then returned as the final result.

**Example Walkthrough**

Let's walk through an example to illustrate the solution approach using the array [2,1,2].

**Sort the Array:** First, sort the array to become [1,2,2]. Sorting ensures any duplicates are next to each other which helps later in the process to avoid duplicate subsets.

**Depth-First Search (DFS):** We define a dfs function that will be used to explore possible subsets.

**Recursive Exploration:** The exploration starts with an empty subset []. This subset is immediately added to the answer set ans because it is a valid subset (the empty set).

**Iterative Inclusion:** For the first call to dfs, we will look at each element starting with the first element of the sorted array.

Include the first element [1]. At this stage, dfs is recursively called to consider the next number.

At the next level, we have [1,2]. We include it and then the dfs explores further.

Now we have [1,2,2], which is also included.

After considering each of these, we backtrack to [1,2] and remove the last element to reconsider our options.

**Since the next element is the same as the previous (as we just removed it), the loop skips adding [1,2] again because it would be a duplicate.**

**Backtracking:** After the above steps, we backtrack to an empty subset [] and proceed to the next element:

The next element is [2]. We include it, and then dfs is called recursively.

**At the next level, the function sees another 2. However, since the previous element is also 2, and we are not at the start of a new subset creation (checked by i != u), the algorithm takes care not to include [2,2] as this is already considered. This is where the duplicate gets skipped.**

**Global Answer Set:** All the while, subsets like [], [1], [1,2], [1,2,2], and [2] are being added to the global ans list.

**Complete Exploration:** After the function explores all possibilities, the ans now contains, without duplicates:

[]

[1]

[2]

[1,2]

[1,2,2]

**Return Results:** Finally, the dfs has finished exploring, and ans with all the unique subsets is returned as the result.

This example demonstrates how the algorithm ensures all unique subsets of [2,1,2] are found by methodically exploring all options while skipping over duplicates due to sorting and the extra conditional skip in the recursive loop.

**Solution Implementation**

class Solution {

// List to store the final subsets

private List<List<Integer>> subsets;

// The provided array of numbers, from which we will form subsets

private int[] numbers;

// Public method to find all subsets with duplicates

public List<List<Integer>> subsetsWithDup(int[] nums) {

subsets = new ArrayList<>(); // Initialize the subsets list

Arrays.sort(nums); // Sort the array to handle duplicates

this.numbers = nums; // Store the sorted array in the numbers variable

backtrack(0, new ArrayList<>()); // Start the backtrack algorithm

return subsets; // Return the list of subsets

}

// Helper method to perform the backtrack algorithm

private void backtrack(int index, List<Integer> currentSubset) {

// Add a new subset to the answer list, which is a copy of the current subset

subsets.add(new ArrayList<>(currentSubset));

// Iterate through the numbers starting from index

for (int i = index; i < numbers.length; ++i) {

// Skip duplicates: check if the current element is the same as the previous one

if (i != index && numbers[i] == numbers[i - 1]) {

continue; // If it's a duplicate, skip it

}

// Include the current element in the subset

currentSubset.add(numbers[i]);

// Recursively call backtrack for the next elements

backtrack(i + 1, currentSubset);

// Exclude the current element from the subset (backtrack step)

currentSubset.remove(currentSubset.size() - 1);

}

}

}

**Time and Space Complexity**

The given Python code defines a method subsetsWithDup that finds all possible subsets of a given set of numbers that might contain duplicates. To avoid duplicate subsets, the algorithm sorts the array and skips over duplicates during the depth-first search (DFS).

**Time Complexity:**

The time complexity of this algorithm can be analyzed based on the number of recursive calls and the operations performed on each call.

The function dfs is called recursively, at most, once for every element starting from the current index u. In the worst case, this means potentially calling dfs for each subset of nums.

Since nums has n elements, there are 2^n possible subsets including the empty subset.

However, because the list is sorted and the algorithm skips duplicate elements within the same recursive level, the number of recursive calls may be less than 2^n.

The append and pop methods of a list run in O(1) time.

Therefore, the time complexity of the algorithm is essentially bounded by the number of recursive calls and is O(2^n) in the worst case, when all elements are unique.

**Space Complexity:**

The space complexity is considered based on the space used by the recursion stack and the space needed to store the subsets (ans).

The maximum depth of the recursive call stack is n, which is the length of nums. Each recursive call uses space for local variables, which is O(n) space in the stack at most.

The list ans will eventually contain all subsets, and thus, it will have 2^n elements, considering each subset as an element. However, the total space taken by all subsets combined is also considerable as each of the 2^n subsets could have up to n elements. This effectively adds up to O(n \* 2^n) space.

Considering these factors, the space complexity of the code can be defined as O(n \* 2^n) because the space used by the algorithm is proportional to the number of subsets generated, and each subset at most can have n elements.

**Solution 2: Backtracking style 2 (720min, too long to sort out why local variable to skip duplicate elements is mandatory)**

**Correct solution 2.1 with local variable 'i' to skip duplicate elements on particular "Not pick" branch**

class Solution {

    public List<List<Integer>> subsetsWithDup(int[] nums) {

        List<List<Integer>> result = new ArrayList<List<Integer>>();

        Arrays.sort(nums);

        helper(nums, result, new ArrayList<Integer>(), 0);

        return result;

    }

    private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, int index) {

        if(index >= nums.length) {

            result.add(new ArrayList<Integer>(tmp));

            return;

        }

        int i = index;

        while(i + 1 < nums.length && nums[i] == nums[i + 1]) {

            i++;

        }

        // Not pick

        helper(nums, result, tmp, i + 1);

        // Pick

        tmp.add(nums[index]);

        helper(nums, result, tmp, index + 1);

        tmp.remove(tmp.size() - 1);

    }

}

Space complexity: O(n + n \* 2^n) = O(n \* 2^n)

For recursion: max depth the call stack is going to reach at any time is length of nums, n.

For output: we're creating 2^n subsets where the average set size is n/2 (for each A[i],

half of the subsets will include A[i], half won't) = n/2 \* 2^n = O(n \* 2^n). Or in a different way,

the total output size is going to be the summation of the binomial coefficients, the total number

of r-combinations we can make at each r size \* r elements from 0..n which evaluates to n\*2^n.

More informally, at size 0, how many empty sets can we make from n elements, then at size 1 how

many subsets of 1 elements can we make from n elements, at size 2, how many subsets of 2 elements

can we make ... at size n, etc.

So total is call stack of n + output of n \* 2^n = O(n \* 2^n). If we don't include the output

(eg if just asked to print in an interview) then just O(n) of course.

Time Complexity: O(n \* 2^n)

The recursive function is called 2^n times. Because we have 2 choices at each iteration in nums array.

Either we include nums[i] in the current set, or we exclude nums[i]. This array nums is of size

n = number of elements in nums.

We need to create a copy of the current set because we reuse the original one to build all the

valid subsets. This copy costs O(n) and it is performed at each call of the recursive function,

which is called 2^n times as mentioned in above. So total time complexity is O(n x 2^n).

**Progress of correct solution 2.1**

Round 1:

nums={1,2,2,3},result={},tmp={},index=0

helper({1,2,2,3},{},{},0)

i=0 -> no skip happening

--------------------------------------------

Round 2:

nums={1,2,2,3},result={},tmp={},index=0

helper({1,2,2,3},{},{},0+1)

i=index=1 -> nums[1]=2 == nums[1+1]=2 -> i++=2 skip happening

--------------------------------------------

Round 3:

nums={1,2,2,3},result={},tmp={},index=2

helper({1,2,2,3},{},{},2+1)

i=index=3 -> no skip happening

--------------------------------------------

Round 4:

nums={1,2,2,3},result={},tmp={},index=3

helper({1,2,2,3},{},{},3+1)

index=4 == nums.length -> result={{}}

return to Round 3 statistics

--------------------------------------------

Round 5: Continue from Round 3

index=3

tmp.add(nums[3])=tmp.add(3)={3}

--------------------------------------------

Round 6:

nums={1,2,2,3},result={{}},tmp={3},index=3

helper({1,2,2,3},{{}},{3},3+1)

index=4 == nums.length -> result={{}{3}}

return to Round 5 statistics

tmp.remove(1-1)={}

End statement back to Round 2 statistics

--------------------------------------------

Round 7:

index=1

tmp.add(nums[1])=tmp.add(2)={2}

--------------------------------------------

Round 8:

nums={1,2,2,3},result={{}{3}},tmp={2},index=1

helper({1,2,2,3},{{}{3}},{2},1+1)

i=index=2

helper({1,2,2,3},{{}{3}},{2},2+1)

i=index=3

helper({1,2,2,3},{{}{3}},{2},3+1)

index=4 == nums.length -> result={{}{3}{2}}

return to index=3 statistics

--------------------------------------------

Round 9:

tmp.add(nums[3])={2,3}

helper({1,2,2,3},{{}{3}{2}},{2,3},3+1)

index=4 == nums.length -> result={{}{3}{2}{2,3}}

return to index=3 statistics

tmp.remove(2-1)={2}

end statement back to Round 8 statistics

--------------------------------------------

Round 10:

tmp={2},index=2

tmp.add(nums[2])={2,2}

helper({1,2,2,3},{{}{3}{2}{2,3}},{2,2},2+1)

i=index=3

--------------------------------------------

Round 11:

helper({1,2,2,3},{{}{3}{2}{2,3}},{2,2},3+1)

index=4 == nums.length -> result={{}{3}{2}{2,3}{2,2}}

return to index=3 statistics

--------------------------------------------

Round 12:

tmp.add(nums[3])={2,2,3}

helper({1,2,2,3},{{}{3}{2}{2,3}{2,2}},{2,2,3},3+1)

index=4 == nums.length -> result={{}{3}{2}{2,3}{2,2}{2,2,3}}

return to index=3 statistics

tmp.remove(3-1)={2,2}

tmp.remove(2-1)={2}

tmp.remove(1-1)={}

--------------------------------------------

Round 13:

tmp={},index=0

tmp.add(nums[0])={1}

helper({1,2,2,3},{{}{3}{2}{2,3}{2,2}{2,2,3}},{1},0+1)

i=index=1 -> nums[1]=2 == nums[1+1]=2 -> i++=2 skip happening

--------------------------------------------

Round 14:

helper({1,2,2,3},{{}{3}{2}{2,3}{2,2}{2,2,3}},{1},2+1)

i=index=3 -> no skip happening

--------------------------------------------

Round 15:

helper({1,2,2,3},{{}{3}{2}{2,3}{2,2}{2,2,3}},{1},3+1)

index=4 == nums.length -> result={{}{3}{2}{2,3}{2,2}{2,2,3}{1}}

return to index=3 statistics

--------------------------------------------

Round 16:

tmp.add(nums[index])=tmp.add(nums[3])={1,3}

index=4 == nums.length -> result={{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}}

return to index=3 statistics

tmp.remove(2-1)={1}

return to index=1 statistics

--------------------------------------------

Round 17:

tmp.add(nums[index])=tmp.add(nums[1])={1,2}

helper({1,2,2,3},{{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}},{1,2},1+1)

i=index=2

helper({1,2,2,3},{{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}},{1,2},2+1)

i=index=3

helper({1,2,2,3},{{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}},{1,2},3+1)

i=index=4 == nums.length -> result={{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}{1,2}}

return to index=3 statistics

--------------------------------------------

Round 18:

tmp.add(nums[3])={1,2,3}

helper({1,2,2,3},{{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}{1,2}},{1,2,3},3+1)

index=4 == nums.length -> result={{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}{1,2}{1,2,3}}

return to index=3 statistics

tmp.remove(3-1)={1,2}

end statement back to index=2 statistics

--------------------------------------------

Round 19:

tmp.add(nums[2])={1,2,2}

helper({1,2,2,3},{{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}{1,2}},{1,2,2},2+1)

i=index=3

helper({1,2,2,3},{{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}{1,2}},{1,2,2},3+1)

index=4 == nums.length -> result={{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}{1,2}{1,2,3}{1,2,2}}

return to index=3 statistics

--------------------------------------------

Round 20:

tmp.add(nums[3])={1,2,2,3}

helper({1,2,2,3},{{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}{1,2}{1,2,2}},{1,2,2,3},3+1)

index=4 == nums.length -> result={{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}{1,2}{1,2,3}{1,2,2}{1,2,2,3}}

return to index=3 statistics

tmp.remove(4-1)={1,2,2}

tmp.remove(3-1)={1,2}

tmp.remove(2-1)={1}

tmp.remove(1-1)={}

--------------------------------------------

End

============================================

Result time elapsed statistics:

result={{}}

result={{}{3}}

result={{}{3}{2}}

result={{}{3}{2}{2,3}}

result={{}{3}{2}{2,3}{2,2}}

result={{}{3}{2}{2,3}{2,2}{2,2,3}}

result={{}{3}{2}{2,3}{2,2}{2,2,3}{1}}

result={{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}}

result={{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}{1,2}}

result={{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}{1,2}{1,2,3}}

result={{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}{1,2}{1,2,3}{1,2,2}}

result={{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}{1,2}{1,2,3}{1,2,2}{1,2,2,3}}

**Correct solution 2.1 recursion step by step picture**

Start from index=1 (level 2) "Not pick first 2 branch" also will not pick second 2, local variable 'i' helps skip happen only on "Not pick first 2 branch" and not impact "Pick first 2 branch"

                                          { }

                      /                                        \

                    { }                                        {1} -> [(1),2,2,3]:index=0(level1)

            /                \                      /                    \

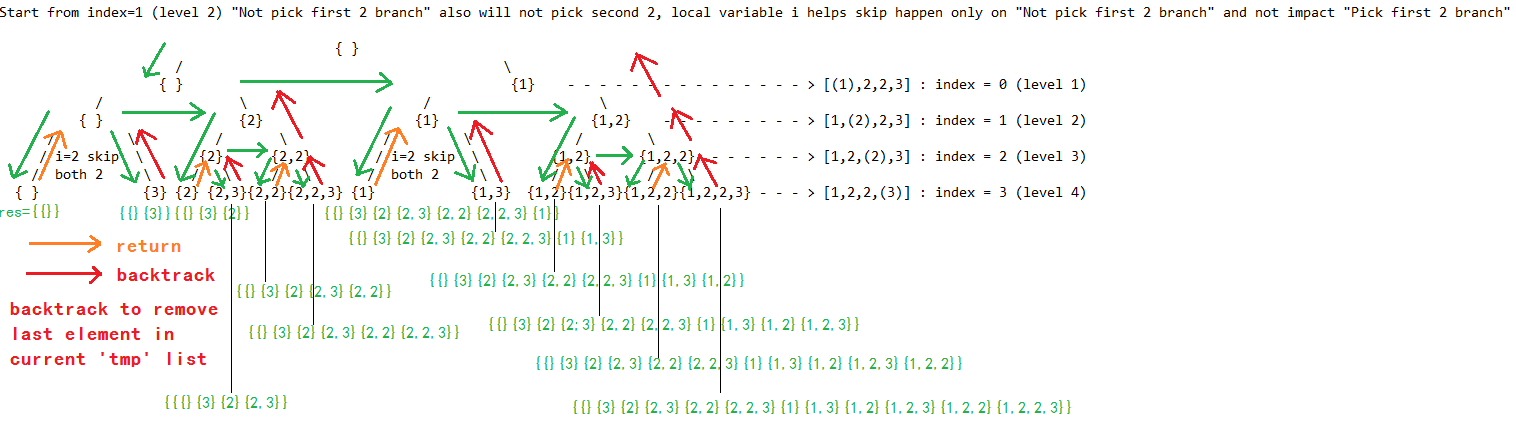
          { }                {2}                  {1}                  {1,2} -> [1,(2),2,3]:index=1(level2)

      /        \          /      \            /        \            /        \

    / i=2 skip  \      {2}      {2,2}        / i=2 skip  \        {1,2}      {1,2,2} -> [.2(2).]:index=2(level3)

    /  both 2    \    /  \    /  \      /  both 2    \        /  \      /    \

  { }            {3} {2} {2,3}{2,2}{2,2,3} {1}            {1,3}  {1,2}{1,2,3}{1,2,2}{1,2,2,3}[..(3)]:idx=3(level4)



**If not skip both 2 in "Not pick first 2 branch", what will happen ?**

                                          { }

                      /                                        \

                    { }                                        {1} -> [(1),2,2,3]:index=0(level1)

            /                \                      /                    \

          { }                {2}                  {1}                  {1,2} -> [1,(2),2,3]:index=1(level2)

      /        \          /      \            /        \            /        \

    { } No skip {2}      {2}      {2,2}      {1} No skip {1,2}      {1,2}      {1,2,2} -> [.2(2).]:index=2(level3)

    /          /  \    /  \    /  \      /          /    \      /  \      /    \

  { }        {2} {2,3} {2} {2,3}{2,2}{2,2,3} {1}      {1,2}{1,2,3}{1,2}{1,2,3}{1,2,2}{1,2,2,3}[..(3)]:idx=3(level4)

            Duplicate                                  Duplicate

**We can see duplicate subsets generated as {2}{2,3}{1,2}{1,2,3} based on second 2 (index=2), which not happen in correct solution because we skip second 2 in "Not pick first 2" branch**

**Correct solution 2.2 with local variable 'i' to skip duplicate elements on particular "Not pick" branch**

class Solution {

    public List<List<Integer>> subsetsWithDup(int[] nums) {

        List<List<Integer>> result = new ArrayList<List<Integer>>();

        Arrays.sort(nums);

        helper(nums, result, new ArrayList<Integer>(), 0);

        return result;

    }

    private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, int index) {

        if(index >= nums.length) {

            result.add(new ArrayList<Integer>(tmp));

            return;

        }

        // Not pick

        // Not add, then we will not add all the following same element, just jump to the index where nums[index] is a different value

        int i = index;

        while(i < nums.length && nums[i] == nums[index]) {

            i++;

        }

        // Be careful, the next "Not pick" recursion start from 'i' not 'i + 1',

        // because nums[i] is the first element different than nums[index] not

        // nums[i + 1]

        // Compare to below style

        // while(i + 1 < nums.length && nums[i] == nums[i + 1]) {i++;}

        // helper(nums, result, tmp, i + 1)

        helper(nums, result, tmp, i);

        // Pick

        tmp.add(nums[index]);

        helper(nums, result, tmp, index + 1);

        tmp.remove(tmp.size() - 1);

    }

}

**Different styles to skip duplicate elements in correct solution 2.1 and 2.2?**

e.g

For sorted array nums={1,2,2,2,5}, index=1, all duplicate '2' stored continuously in array

-------------------------------------

For

int i = index;

while(i < nums.length && nums[i] == nums[index]) {i++;}

helper(nums, result, tmp, i);

=> while loop ending when i=4, nums[4]=5 != nums[1]=2, not pick up branch skip all duplicate 2

and start from 5 requires pass i(=4) to next recursion

-------------------------------------

For

int i = index;

while(i + 1 < nums.length && nums[i] == nums[i + 1]) {i++;}

helper(nums, result, tmp, i + 1);

=> while loop ending when i=3, nums[3]=2 != nums[4]=5, not pick up branch skip all duplicate 2

and start from 5 requires pass i + 1(=4) to next recursion

**Wrong solution without local variable 'i' to skip duplicate elements**

class Solution {

    public List<List<Integer>> subsetsWithDup(int[] nums) {

        List<List<Integer>> result = new ArrayList<List<Integer>>();

        Arrays.sort(nums);

        helper(nums, result, new ArrayList<Integer>(), 0);

        return result;

    }

    private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, int index) {

        if(index >= nums.length) {

            result.add(new ArrayList<Integer>(tmp));

            return;

        }

        while(index + 1 < nums.length && nums[index] == nums[index + 1]) {

            index++;

        }

        // Not pick

        helper(nums, result, tmp, index + 1);

        // Pick

        tmp.add(nums[index]);

        helper(nums, result, tmp, index + 1);

        tmp.remove(tmp.size() - 1);

    }

}

**Progress of wrong solution**

Round 1:

nums={1,2,2,3},result={},tmp={},index=0

helper({1,2,2,3},{},{},0)

--------------------------------------------

Round 2:

helper({1,2,2,3},{},{},0+1)

index=1 -> nums[index]==nums[index+1] -> i++=2 skip happening

--------------------------------------------

Round 3:

helper({1,2,2,3},{},{},2+1)

index=3

--------------------------------------------

Round 4:

helper({1,2,2,3},{},{},3+1)

index=4 == nums.length -> result={{}}

return to index=3 statistics

--------------------------------------------

Round 5:

index=3

tmp.add(nums[3])={3}

helper({1,2,2,3},{{}},{3},3+1)

index=4 == nums.length -> result={{}{3}}

tmp.remove(1-1)={}

--------------------------------------------

1st difference happening, compare to correct solution Round 7 & 8, after end statement, back to previous recursion, the index can rollback to 1, but here the index only able to rollback to 2, why?

Because in correct solution we reserve index=1 status by assigning index=1 to a new local variable 'i' and only loop on 'i' in previous recursion to skip the duplicate elements, so even 'i' change

to other value and used by "Not pick" branch "helper(nums, result, tmp, i + 1)", index=1 kept as is and used by "Pick" branch "helper(nums, result, tmp, index + 1)", the local variable 'i'

prevents the side effect of updating 'index' in all traversal branches and limits the updating only impact the "Not pick" branch. It helps us when return from further recursion back to "Pick"

branch we can still start with index=1 status, not like wrong solution here, since we globally only use 'index' and no local variable 'i' helps to isolate updating 'index' impact, it will wrongly

impact "Pick" branch

Round 6:

index=2

tmp.add(nums[2])={2}

helper({1,2,2,3},{{}{3}},{2},2+1)

index=3

helper({1,2,2,3},{{}{3}},{2},3+1)

index=4 == nums.length -> result={{}{3}{2}}

return to index=3 statistics

--------------------------------------------

Round 7:

index=3

tmp.add(nums[3])={2,3}

helper({1,2,2,3},{{}{3}},{2,3},3+1)

index=4 == nums.length -> result={{}{3}{2}{2,3}}

return to index=3 statistics

tmp.remove(2-1)={2}

tmp.remove(1-1)={}

end statement back to index=0

--------------------------------------------

Round 8:

index=0

tmp.add(nums[0])={1}

helper({1,2,2,3},{{}{3}{2}{2,3}},{1},0+1)

index=1 -> nums[index]==nums[index+1] -> i++=2 skip happening

helper({1,2,2,3},{{}{3}{2}{2,3}},{1},2+1)

index=3

helper({1,2,2,3},{{}{3}{2}{2,3}},{1},3+1)

index=4 == nums.length -> result={{}{3}{2}{2,3}{1}}

return to index=3 statistics

--------------------------------------------

Round 9:

index=3

tmp.add(nums[3])={1,3}

helper({1,2,2,3},{{}{3}{2}{2,3}{1}},{1,3},3+1)

index=4 == nums.length -> result={{}{3}{2}{2,3}{1}{1,3}}

return to index=3 statistics

tmp.remove(2-1)={1}

end statement back to index=2

--------------------------------------------

Round 10:

index=2

tmp.add(nums[2])={1,2}

helper({1,2,2,3},{{}{3}{2}{2,3}{1}{1,3}},{1,2},2+1)

index=3

helper({1,2,2,3},{{}{3}{2}{2,3}{1}{1,3}},{1,2},3+1)

index=4 == nums.length -> result={{}{3}{2}{2,3}{1}{1,3}{1,2}}

return to index=3 statistics

--------------------------------------------

Round 11:

tmp.add(nums[3])={1,2,3}

helper({1,2,2,3},{{}{3}{2}{2,3}{1}{1,3}{1,2}},{1,2,3},3+1)

index=4 == nums.length -> result={{}{3}{2}{2,3}{1}{1,3}{1,2}{1,2,3}}

return to index=3 statistics

tmp.remove(3-1)={1,2}

tmp.remove(2-1)={1}

tmp.remove(1-1)={}

--------------------------------------------

End

============================================

Result time elapsed statistics:

result={{}}

result={{}{3}}

result={{}{3}{2}}

result={{}{3}{2}{2,3}}

result={{}{3}{2}{2,3}{1}}

result={{}{3}{2}{2,3}{1}{1,3}}

result={{}{3}{2}{2,3}{1}{1,3}{1,2}}

result={{}{3}{2}{2,3}{1}{1,3}{1,2}{1,2,3}}

**Wrong solution recursion step by step picture**

Because of no local variable 'i' to inherit 'index' value and used in skip duplicate elements, in Round 6 and 7 we can see after adding 'tmp' list {2,3} into result, in wrong solution it directly start to backtrack 'tmp' list from {2,3} back to empty list {} and index=0(the correct answer only backtrack to {2}, index=2)

                                          { }

                      /                                        \

                    { }                                        {1} -> [(1),2,2,3]:index=0(level 1)

            /                \                      /                    \

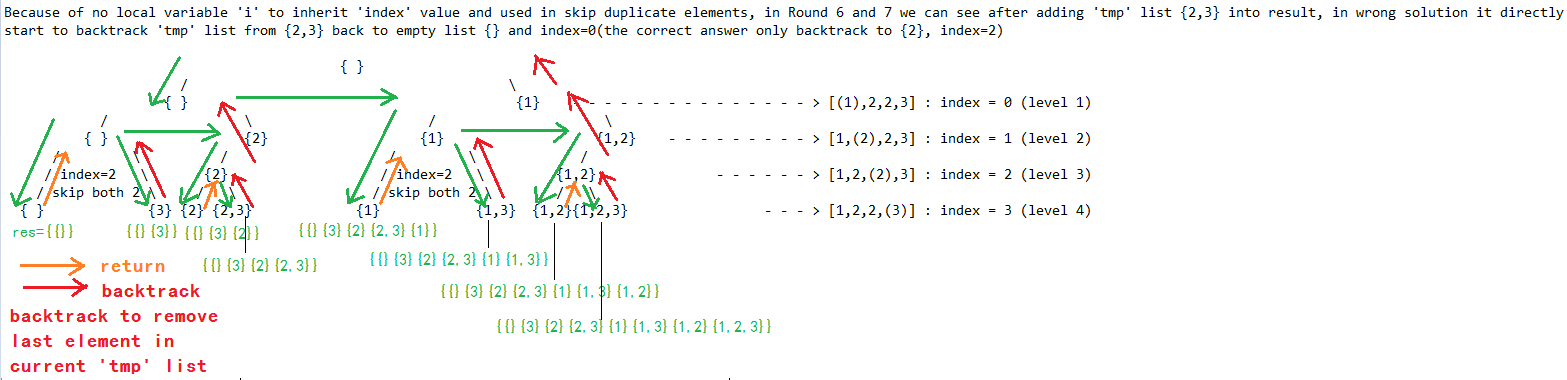
          { }                {2}                  {1}                  {1,2} -> [1,(2),2,3]:index=1(level 2)

      /        \          /                    /        \            /

    / index=2  \      {2}                  / index=2  \        {1,2}      -> [1,2,(2),3]:index=2(level 3)

    / skip both 2 \    /  \                / skip both 2 \        /  \

  { }            {3} {2} {2,3}            {1}            {1,3}  {1,2}{1,2,3}  -> [1,2,2,(3)]:index=3(level 4)



**Alternative correct solution without local variable but switch order of "Pick" or "Not pick" branch**

**Move "Pick first 2 branch" before skip duplicate elements while loop statement, then  even update 'index' directly without local variable 'i' will only impact "Not pick first 2" branch**

class Solution {

    public List<List<Integer>> subsetsWithDup(int[] nums) {

        List<List<Integer>> result = new ArrayList<List<Integer>>();

        Arrays.sort(nums);

        helper(nums, result, new ArrayList<Integer>(), 0);

        return result;

    }

    private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, int index) {

        if(index >= nums.length) {

            result.add(new ArrayList<Integer>(tmp));

            return;

        }

        // Move "Pick first 2 branch" before skip duplicate elements while loop statement,

        // then  even update 'index' directly without local variable 'i' will only impact

        // "Not pick first 2" branch

        // Pick

        tmp.add(nums[index]);

        helper(nums, result, tmp, index + 1);

        tmp.remove(tmp.size() - 1);

        while(index + 1 < nums.length && nums[index] == nums[index + 1]) {

            index++;

        }

        // Not pick

        helper(nums, result, tmp, index + 1);

    }

}

**Why local variable to skip duplicate elements only on particular "Not pick" branch is mandatory ?**

**In wrong solution process Round 6, the 1st difference happening, compare to correct solution Round 7 & 8, after end statement, back to previous recursion, the index can rollback to 1, but here in the wrong solution the index only able to rollback to 2, why? Because in correct solution we reserve index=1 status by assigning index=1 to a new local variable 'i' and only loop on 'i' in previous recursion to skip the duplicate elements, so even 'i' change to other value and used by "Not pick" branch "helper(nums, result, tmp, i + 1)", index=1 kept as is and used by "Pick" branch "helper(nums, result, tmp, index + 1)", the local variable 'i' prevents the side effect of updating 'index' in all traversal branches and limits the updating only impact the "Not pick" branch. It helps us when return from further recursion back to "Pick" branch we can still start with index=1 status, not like wrong solution here, since we globally only use 'index' and no local variable 'i' helps to isolate updating 'index' impact, it will wrongly impact "Pick" branch, the wrong solution above is the negative impact.**

**Video explain why and how to skip duplicate elements only on particular "Not pick" branch**

[Subsets II - Backtracking - Leetcode 90 - Python](https://www.youtube.com/watch?v=Vn2v6ajA7U0)

<https://www.youtube.com/watch?v=Vn2v6ajA7U0>

**Refer to**

[L491.Non-decreasing Subsequences (Ref.L90)](note://6519624114A34866BC5B26A6BE4E32F5)

[L78.11.1.Subsets (Ref.L90)](note://CA8092FEC5984E1EB27E26C85FB3B887)