<https://leetcode.com/problems/increasing-subsequences/>

Given an integer array nums, return all the different possible **non-decreasing** subsequences of the given array with **at least two elements**. You may return the answer in **any order**.

The given array may contain duplicates, and two equal integers should also be considered a special case of increasing sequence.

**Example 1:**

Input: nums = [4,6,7,7]

Output: [[4,6],[4,6,7],[4,6,7,7],[4,7],[4,7,7],[6,7],[6,7,7],[7,7]]

**Example 2:**

Input: nums = [4,4,3,2,1]

Output: [[4,4]]

**Constraints:**

1 <= nums.length <= 15

-100 <= nums[i] <= 100

**Attempt 1: 2022-10-30**

**Wrong answer:**

**1.Don't sort the input**

**Its not L40.Combination Sum II, because not able to sort to make the input monotonic increasing, we have to keep the order of input.**

**e.g  Input nums = [4,4,3,2,1], after sort the input will be [1,2,3,4,4] -> the output will be [[1,2],[1,2,3],[1,2,3,4],[1,2,3,4,4],[1,2,4],[1,2,4,4],[1,3],[1,3,4],[1,3,4,4],[1,4],[1,4,4],[2,3],[2,3,4],[2,3,4,4],[2,4],[2,4,4],[3,4],[3,4,4],[4,4]], the expected output should be [4,4]**

Input: [4,4,3,2,1]

Wrong output: [[1,2],[1,2,3],[1,2,3,4],[1,2,3,4,4],[1,2,4],[1,2,4,4],[1,3],[1,3,4],[1,3,4,4],[1,4],[1,4,4],[2,3],[2,3,4],[2,3,4,4],[2,4],[2,4,4],[3,4],[3,4,4],[4,4]]

Expect output: [[4,4]]

class Solution {

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

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

        // We cannot sort input

        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(tmp.size() > 1) {

            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(tmp.size() > 1) {... return}**

class Solution {

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

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

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

        return result;

    }

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

        if(tmp.size() > 1) {

            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);

        }

    }

}

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

class Solution {

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

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

        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;

        }

        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:  Recursive traversal (360min, too long to figure out two new conditions to filter out elements rather than L90.Subsets II)**

class Solution {

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

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

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

        return result;

    }

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

        // Condition to limit subset size more than 1 (no single element)

        if(tmp.size() >= 2) {

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

        }

        Set<Integer> set = new HashSet<Integer>();

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

            // Condition 1: set.contains(nums[i])

            // Create a local set to filter possible duplicates

            // Note: we introduce a new local set instead of use existing properties like

            // "i > index && nums[i] == nums[i - 1]" to compare because we cannot sort input

            // array into monotonic increasing format, same elements are not necessary adjacent

            // e.g

            // If not adding local set to filter out duplicates, duplicate combination will

            // generate as 4 with first 7 is [4,7], 4 with second 7 is also [4,7], [4,7]

            // happen twice

            // Input: [4,6,7,7]

            // Output: [[4,6],[4,6,7],[4,6,7,7],[4,6,7],[4,7],[4,7,7],[4,7],[6,7],[6,7,7],[6,7],[7,7]]

            // Expected: [[4,6],[4,6,7],[4,6,7,7],[4,7],[4,7,7],[6,7],[6,7,7],[7,7]]

            // After adding set, in each level, we will filter duplicate values after first present,

            // like here, we will only add first 7 for combination [4,7], the second 7 will filter out,

            // but this is just a local set, when process to next level, we will create a new set and

            // it will not block to add same value again but in next level, like here after [4,7] the

            // next level we can add 7 again as [4,7,7]

            // ------------------------------------------------------------------------

            // Condition 2: nums[i] < tmp.get(tmp.size() - 1)

            // Newly added element should not less than last element on current combination list(path)

            // e.g

            // Input: [4,4,3,2,1]

            // Expected: [4,4]

            // After second 4, all elements as 3,2,1 cannot be appended on the current combination list

            if(set.contains(nums[i]) || (tmp.size() > 0 && nums[i] < tmp.get(tmp.size() - 1))) {

                continue;

            }

            set.add(nums[i]);

            tmp.add(nums[i]);

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

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

        }

    }

}

Time Complexity: O(2^n)

Space Complexity: O(2^n)

**Two new conditions to filter out elements rather than L90.Subsets II**

**Condition 1: set.contains(nums[i])**

**Create a local set to filter possible duplicates**

**Note: we introduce a new local set instead of use existing properties like "i > index && nums[i] == nums[i - 1]" to compare because we cannot sort input array into monotonic increasing format, same elements are not necessary adjacent**

e.g

If not adding local set to filter out duplicates, duplicate combination will generate as 4 with first 7 is [4,7], 4 with second 7 is also [4,7],[4,7] happen twice

Input: [4,6,7,7]

Output: [[4,6],[4,6,7],[4,6,7,7],[4,6,7],[4,7],[4,7,7],[4,7],[6,7],[6,7,7],[6,7],[7,7]]

Expected: [[4,6],[4,6,7],[4,6,7,7],[4,7],[4,7,7],[6,7],[6,7,7],[7,7]]

**After adding set, in each level, we will filter duplicate values after first present, like here, we will only add first 7 for combination [4,7], the second 7 will filter out, but this is just a local set, when process to next level, we will create a new set and it will not block to add same value again but in next level, like here after [4,7] the next level we can add 7 again as [4,7,7]**

**Condition 2: nums[i] < tmp.get(tmp.size() - 1)**

Newly added element should not less than last element on current combination list (path)

e.g

Input: [4,4,3,2,1]

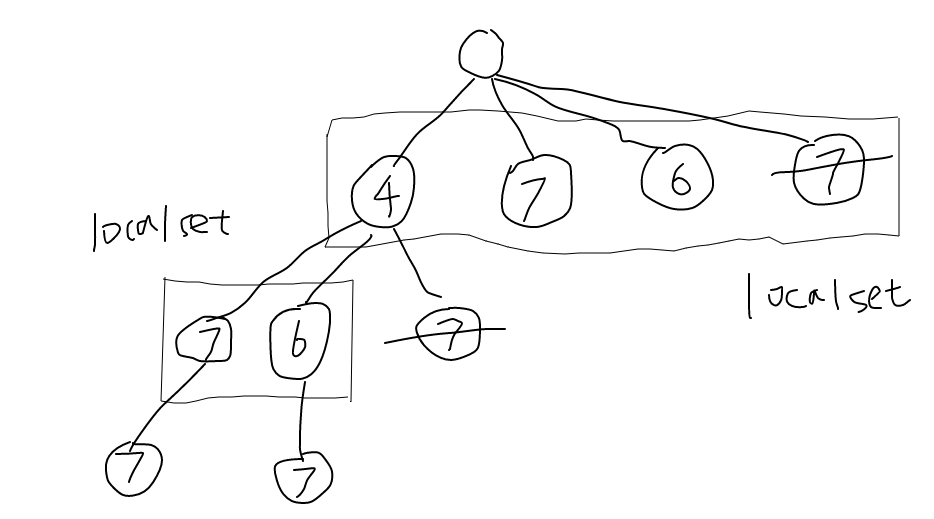
Expected: [4,4]

After second 4, all elements as 3,2,1 cannot be appended on the current combination list

**How the local set on each recursion level works ?**

<https://leetcode.com/problems/increasing-subsequences/discuss/97130/Java-20-lines-backtracking-solution-using-set-beats-100./101613>

consider the following case: [4, 7, 6, 7], we can draw recursion tree like this:



Java implementation:

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

        // we cannot sort array first, sequence matters

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

        search(nums, res, new ArrayList<Integer>(), 0);

        return res;

    }

    private void search(int[] nums, List<List<Integer>> res, List<Integer> list, int pos) {

        if(list.size() >= 2) {

            res.add(new ArrayList<Integer>(list));

        }

        Set<Integer> visited = new HashSet<>(); // local set to de-duplicate

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

            // if(i > pos && nums[i] == nums[i - 1]) continue; // WRONG

            if(visited.contains(nums[i])) continue;

            visited.add(nums[i]);

            if(list.size() == 0 || nums[i] >= list.get(list.size() - 1)) {

                list.add(nums[i]);

                search(nums, res, list, i + 1);

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

            }

        }

    }

**Also refer to**

<https://leetcode.com/problems/increasing-subsequences/discuss/97134/Evolve-from-intuitive-solution-to-optimal>

Solution 4: Duplicates can also be avoided in recursion. Starting from a given number, we pick the next number. We cache the numbers already tried to avoid duplicates.

vector<vector<int>> findSubsequences(vector<int>& nums) {

        vector<vector<int>> res;

        vector<int> one;

        find(0,nums,one,res);

        return res;

    }

        int n = nums.size();

        if(one.size()>1) res.push\_back(one);

        unordered\_set<int> ht;

        for(int i=p;i<n;i++) {

            if((!one.empty() && nums[i] < one.back()) || ht.count(nums[i])) continue;

            ht.insert(nums[i]);

            one.push\_back(nums[i]);

            find(i+1,nums,one,res);

            one.pop\_back();

        }

    }

<https://leetcode.com/problems/increasing-subsequences/discuss/97147/Java-solution-beats-100>

A bit different than use ArrayList for 'tmp', here use a Deque for 'tmp', which has "peekLast" method to find the last element

public class Solution {

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

        List<List<Integer>> res = new LinkedList<>();

        helper(new LinkedList<Integer>(), 0, nums, res);

        return res;

    }

    private void helper(LinkedList<Integer> list, int index, int[] nums, List<List<Integer>> res){

        if(list.size()>1) res.add(new LinkedList<Integer>(list));

        Set<Integer> used = new HashSet<>();

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

            if(used.contains(nums[i])) continue;

            if(list.size()==0 || nums[i]>=list.peekLast()){

                used.add(nums[i]);

                list.add(nums[i]);

                helper(list, i+1, nums, res);

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

            }

        }

    }

}

**Refer to chatGPT**

LeetCode problem 491, "Increasing Subsequences," requires you to find all the different increasing subsequences of a given array with a length of at least two. Below is a Java native DFS (Depth-First Search) solution for this problem:

**Problem Statement**

Given an integer array nums, return all the different possible increasing subsequences of the given array with at least two elements. You may return the answer in any order.

**DFS Approach**

Here's the Java code that uses DFS to solve the problem:

public class Solution {

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

Set<List<Integer>> result = new HashSet<>();

dfs(nums, 0, new ArrayList<>(), result);

return new ArrayList<>(result);

}

private void dfs(int[] nums, int index, List<Integer> current, Set<List<Integer>> result) {

if (current.size() >= 2) {

result.add(new ArrayList<>(current));

}

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

if (current.isEmpty() || nums[i] >= current.get(current.size() - 1)) {

current.add(nums[i]);

dfs(nums, i + 1, current, result);

current.remove(current.size() - 1); // backtrack

}

}

}

}

**Explanation**

Set<List<Integer>> result: This stores the final result of unique subsequences.

dfs: The DFS function explores all possible subsequences. The current subsequence is stored in current, and we add to result if its length is at least 2.

Backtracking: After exploring with a particular number in the subsequence, the number is removed to explore other possibilities.

**Example Execution**

For the input nums = {4, 6, 7, 7}, this code will produce the following subsequences:

[4, 6]

[4, 6, 7]

[4, 6, 7, 7]

[4, 7]

[4, 7, 7]

[6, 7]

[6, 7, 7]

[7, 7]

This approach ensures that all possible subsequences are explored and only valid, increasing subsequences are included in the result.

**I see 'result' is a hashset used to filter out all duplicate subsequences, but if handling large input, the initial generated duplicate subsequences will be too much, is there another way to filter out duplicate subsequences earlier ?**

Yes, you can filter out duplicates earlier by avoiding the generation of duplicate subsequences in the first place, rather than generating all subsequences and then removing the duplicates afterward. This can be done by tracking the numbers that have been used at each level of the recursion, which will prevent you from adding the same number multiple times within the same recursive call.

You can use a **set** to keep track of the numbers that have already been used at the current recursion depth, ensuring that duplicates are not generated at each level. Here's how you can modify the existing solution to filter out duplicate subsequences earlier:

**Optimized DFS Solution (with duplicate filtering during recursion)**

public class Solution {

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

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

dfs(nums, 0, new ArrayList<>(), result);

return result;

}

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

if (current.size() >= 2) {

result.add(new ArrayList<>(current));

}

Set<Integer> used = new HashSet<>(); // Track numbers used at this level

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

// Skip duplicates or if the current number doesn't form an increasing sequence

if (used.contains(nums[i]) || (!current.isEmpty() && nums[i] < current.get(current.size() - 1))) {

continue;

}

used.add(nums[i]); // Mark this number as used at this level

current.add(nums[i]); // Add current number to the subsequence

dfs(nums, i + 1, current, result); // Recursive call to explore further

current.remove(current.size() - 1); // Backtrack

}

}

}

**Key Improvements**

used **Set:** Inside the DFS function, we use a set used to track the numbers that have been used at the current recursion depth. This ensures that we do not explore the same number multiple times from the same starting index, which prevents generating duplicate subsequences.

Early Filtering: By skipping numbers that have already been used in the current recursion call, we avoid generating duplicate subsequences early, reducing the overall space and time complexity compared to the original approach where duplicates are filtered out afterward.

**Example Walkthrough**

Given nums = {4, 6, 7, 7}, here's what happens:

At index 0, 4 is added.

At index 1, 6 is added, forming [4, 6].

At index 2, 7 is added, forming [4, 6, 7].

At index 3, the second 7 is skipped because it has already been used at this recursion depth, preventing duplicates like [4, 6, 7, 7].

Similarly, the used set prevents exploring unnecessary branches at each recursion depth.

This solution is more efficient for large inputs, as it avoids unnecessary subsequence generation.

**Solution 2: Backtracking style 2 (720min, too long to sort out different conditions for "Not pick" and "Pick" branch, especially for "Not pick" branch, more complicate than L90.Subsets II)**

**1. For "Pick" branch condition:  if (tmp.size() == 0 || nums[index] >= tmp.get(tmp.size() - 1) {...}**

**2. For "Not pick" branch condition: if(index == 0 || tmp.size() == 0 || tmp.get(tmp.size() - 1) != nums[index]) {...}**

**Correct solution 2.1 "Pick" before "Not pick" style 1**

class Solution {

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

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

        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) {

            if(tmp.size() > 1) {

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

            }

            return;

        }

        // Pick

        if(tmp.size() == 0 || nums[index] >= tmp.get(tmp.size() - 1)) {

            tmp.add(nums[index]);

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

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

        }

// Skip "Not pick" branch happen when coming element is duplicate than existing

// last element on 'tmp' list => "tmp.get(tmp.size() - 1) == nums[index]"

// Note: graph explain below with "blue" color font

        if(index > 0 && tmp.size() > 0 && tmp.get(tmp.size() - 1) == nums[index]) {

            return;

        }

        // Not pick

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

    }

}

Time Complexity: O(2^n)

Space Complexity: O(2^n)

**Refer to**

<https://leetcode.com/problems/increasing-subsequences/discuss/97147/Java-solution-beats-100/363291>

The set is needless:

class Solution {

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

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

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

        return result;

    }

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

        if (index > nums.length - 1) {

            if (ans.size() > 1) result.add(new ArrayList<>(ans));

            return;

        }

        if (ans.isEmpty() || nums[index] >= ans.get(ans.size() - 1)) {

            ans.add(nums[index]);

            helper(nums, index + 1, ans);

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

        }

        // repeated value, so don't need to drill down.

        if (index > 0

            && ans.size() > 0

            && nums[index] == ans.get(ans.size() - 1)) {

            return;

        }

        helper(nums, index + 1, ans);

    }

}

**Correct solution 2.2 "Pick" before "Not pick" style 2, just merge 'return' condition with "Not pick" branch**

class Solution {

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

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

        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) {

            if(tmp.size() > 1) {

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

            }

            return;

        }

        // Pick

        if(tmp.size() == 0 || nums[index] >= tmp.get(tmp.size() - 1)) {

            tmp.add(nums[index]);

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

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

        }

// Skip "Not pick" branch happen when coming element is duplicate than existing

// last element on 'tmp' list => "tmp.get(tmp.size() - 1) == nums[index]"

// Note: graph explain below with "blue" color font

//if(index > 0 && tmp.size() > 0 && tmp.get(tmp.size() - 1) == nums[index]) {

        //    return;

        //}

        // Not pick (merge 'return' condition with "Not pick" branch based on solution 2.1)

        if(index == 0 || tmp.size() == 0 || tmp.get(tmp.size() - 1) != nums[index]) {

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

        }

    }

}

Time Complexity: O(2^n)

Space Complexity: O(2^n)

**Correct solution 2.3 "Not pick" before "Pick", switch the branch will not impact the result, because no local variable used like L90.Subsets II**

class Solution {

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

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

        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) {

            if(tmp.size() > 1) {

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

            }

            return;

        }

        // Not pick

        if(index == 0 || tmp.size() == 0 || tmp.get(tmp.size() - 1) != nums[index]) {

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

        }

        // Pick

        if(tmp.size() == 0 || nums[index] >= tmp.get(tmp.size() - 1)) {

            tmp.add(nums[index]);

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

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

        }

    }

}

Time Complexity: O(2^n)

Space Complexity: O(2^n)

**Refer to**

<https://leetcode.com/problems/increasing-subsequences/discuss/1857460/Java-Backtracking-91-Speed-or-Explained>

Usually when it comes to generating subsets, there is always a way to avoid using a HashSet, and this question is of no exception.

For any element in the array, we can either pick or not pick and we only pick when the current element is no less than the last element in the tmp list, but that along is not enough because we will come across duplicates. Let me elaborate:

**Consider something like 3 -> 5 -> 7 -> 1 -> 7 -> .... Here, we have two 7 in the array, picking the first 7 and skip the second 7 is the exactly same thing as skipping the first 7 and picking the second 7 !**

**It means we have to check the current element with the last element in the tmp list, if they are identical then we disallow not-pick (= only allow pick) current element as an option for the current layer of recursion.** **It works because if the last element in the tmp list is identical as the current element, not-pick current element option will be covered by the previous recursion layer that added that element to the tmp list (i.e. Choose not-pick there, not here), so we don't have to do it again**.

class Solution {

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

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

        gen(0, nums, ans, new ArrayList<>());

        return ans;

    }

    private void gen(int cur, int[] nums, List<List<Integer>> ans, List<Integer> tmp){

        if (cur == nums.length){

            if (tmp.size() > 1){

                ans.add(new ArrayList<>(tmp));

            }

            return;

        }

        if (cur == 0 || tmp.isEmpty() || tmp.get(tmp.size() - 1) != nums[cur]){

            gen(cur + 1, nums, ans, tmp); // not-pick option

        }

        if (tmp.isEmpty() || tmp.get(tmp.size() - 1) <= nums[cur]){

            tmp.add(nums[cur]);

            gen(cur + 1, nums, ans, tmp); // pick option

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

        }

    }

}

**Two step by step examples:**

**Example 1: input = [4,4,6,7]**

                                                  { }

                              /                                          \

                            { }                                          {4}

                      /            \                        /                      \

                    { }              {6}                    {4}                    {4,6}

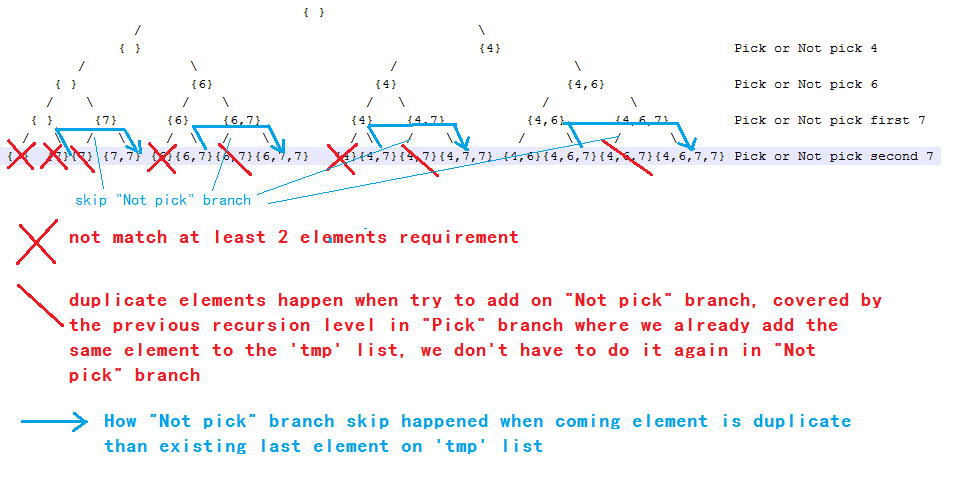
                  /    \          /    \                /  \                /          \

                { }    {7}      {6}    {6,7}          {4}    {4,7}          {4,6}      {4,6,7}

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

              { }  {7}{7} {7,7} {6}{6,7}{6,7}{6,7,7}  {4}{4,7}{4,7}{4,7,7} {4,6}{4,6,7}{4,6,7}{4,6,7,7}

**It works because if the last element in the list is the identical as the current element, not-pick option will be covered by the previous recursion layer that added that element to the tmp list (i.e. Choose not-pick there, not here), so we don't have to do it again --------> A good example below is for removed with "back slash" symboled subsets [6,7], [4,7], [4,6,7] in "Not pick" branch all covered by the previous recursion layer that same subsets in "Pick" branch, blue highlighted below for how "Not pick" branch skip happened when coming element is duplicate than existing last element on 'tmp' list**



**Example 2: input = [4,4,3,2,1]**

                                                                                  { }

                              /                                                                                            \

                              { }                                                                                            {4}

            /                                    \                                              /                                                    \

          { }                                    {4}                                            {4}                                                    {4,4}

      /            \                    /                    \                        /                  \                            /                            \

    { }            {3}                {4}                  {4,3}                    {4}                  {4,3}                      {4,4}                        {4,4,3}

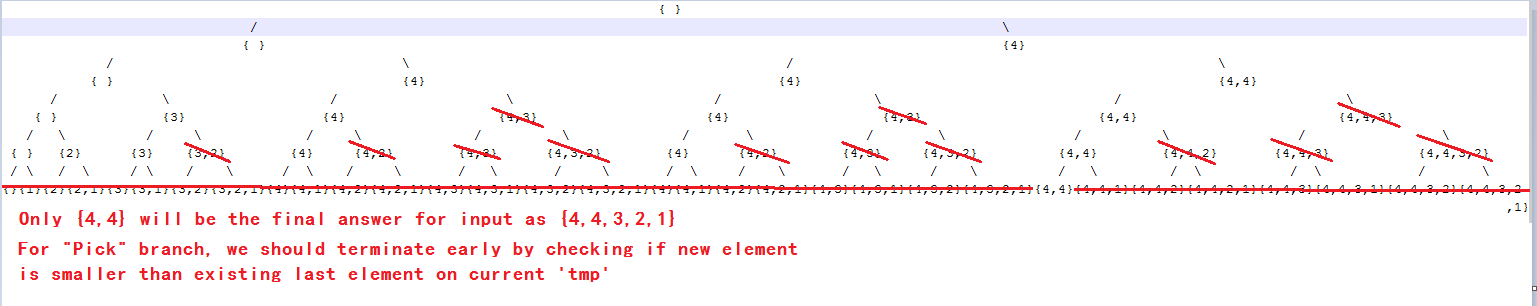
  /  \          /    \            /    \              /          \              /      \              /        \                /          \                /                \

{ }  {2}      {3}    {3,2}        {4}    {4,2}        {4,3}      {4,3,2}        {4}      {4,2}        {4,3}    {4,3,2}          {4,4}        {4,4,2}      {4,4,3}          {4,4,3,2}

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

{}{1}{2}{2,1}{3}{3,1}{3,2}{3,2,1}{4}{4,1}{4,2}{4,2,1}{4,3}{4,3,1}{4,3,2}{4,3,2,1}{4}{4,1}{4,2}{4,2,1}{4,3}{4,3,1}{4,3,2}{4,3,2,1}{4,4}{4,4,1}{4,4,2}{4,4,2,1}{4,4,3}{4,4,3,1}{4,4,3,2}{4,4,3,2

                                                                                                                                                                                            ,1}



**Why in 'Correct solution 2.1 "Pick" before "Not pick" style 1' the statement below have condition requires "index > 0" ?**

To avoid unnecessary comparisons and prevent potential errors when checking the first element. Let me explain why this is necessary.

**Purpose of the Condition**

index > 0**:**This ensures that the comparison tmp.get(tmp.size() - 1) == nums[index] only happens when there is at least one element before nums[index]. If index == 0, the current element is the first element, so there is no "previous" element to compare against. Without this check, trying to access a previous element when index == 0 could result in a comparison against a non-existent element, or worse, an exception like IndexOutOfBoundsException.

tmp.size() > 0**:**This ensures that tmp contains at least one element before trying to access tmp.get(tmp.size() - 1). If tmp is empty, trying to get tmp.get(tmp.size() - 1) would cause an error.

tmp.get(tmp.size() - 1) == nums[index]**:**This checks if the current element nums[index] is the same as the last element in the temporary list tmp. If they are equal, this condition prevents adding the same element again when the recursive call explores the "not pick" branch. This helps to avoid adding duplicate subsequences like [7, 7] multiple times.

**Example Walkthrough**

Consider an input like nums = {4, 6, 7, 7}:

Initially, index = 0, and the first element 4 is considered. There's no previous element (index == 0), so we should not compare it with any prior element, hence the need for index > 0.

When index = 3, the current element is nums[3] = 7. If the previous element (nums[2]) is also 7, we might want to skip this element in the "not pick" branch to avoid duplicate subsequences like [7, 7]. This is where tmp.get(tmp.size() - 1) == nums[index] helps prevent duplicates.

**Summary of the Logic**

**index > 0 ensures that we only compare elements starting from the second index, avoiding unnecessary comparisons for the first element.**

**tmp.size() > 0 ensures that we don't attempt to access an element in tmp if it's empty, preventing potential errors.**

**tmp.get(tmp.size() - 1) == nums[index] checks if the current element is the same as the last picked element, helping to avoid duplicates in the subsequences.**

**This combination prevents duplicate subsequences from being generated and ensures the correctness of the solution.**

**But looks like even remove "index > 0" in condition it still works, just take a bit longer to finish all test cases (e.g take 6ms to finish all test cases rather than 4ms if have "index > 0")**

**Still work which remove "index > 0" condition**

class Solution {

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

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

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) {

if(tmp.size() > 1) {

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

}

return;

}

// Pick

if(tmp.size() == 0 || nums[index] >= tmp.get(tmp.size() - 1)) {

tmp.add(nums[index]);

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

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

}

// Not pick -> but removed condition 'index > 0' still works

if(tmp.size() > 0 && nums[index] == tmp.get(tmp.size() - 1)) {

return;

}

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

}

}

**Why It Still Works**

Even without the index > 0 condition, the logic functions correctly because:

**The second check tmp.size() > 0 && nums[index] == tmp.get(tmp.size() - 1) is sufficient to prevent duplicate subsequences from being generated.**

**When index == 0 (i.e., the first element), the check tmp.size() > 0 ensures that the condition nums[index] == tmp.get(tmp.size() - 1) won't be evaluated, as tmp is empty at that point. Thus, the recursive flow proceeds without errors for the first element.**

**Why It Takes Longer Without index > 0**

**The index > 0 check is a minor optimization. By checking if index is greater than 0, you're skipping the duplicate-check logic for the very first element, which is not necessary since there's no "previous" element to compare it with.**

Without the index > 0 condition, the extra check nums[index] == tmp.get(tmp.size() - 1) is still performed even when index == 0. This isn't harmful but results in a slight overhead that increases the runtime by a small amount, as you've observed (e.g., 6ms vs. 4ms).

**Summary of Differences**

With **index > 0:** Slightly faster (e.g., 4ms) because the comparison nums[index] == tmp.get(tmp.size() - 1) is skipped for the first element.

Without **index > 0:** Works fine but slightly slower (e.g., 6ms) due to the unnecessary comparison being made when index == 0.

So, while removing the index > 0 check doesn't cause any logical errors, keeping it is a minor optimization that saves a bit of processing time.

**Refer to**

<https://algo.monster/liteproblems/491>

**Problem Description**

The problem is asking us to find all the different possible subsequences of a given array of integers nums. A subsequence is a sequence that can be obtained from another sequence by deleting some or no elements without changing the order of the remaining elements. The subsequences we are looking for should be non-decreasing, meaning each element in the subsequence is less than or equal to the subsequent element. Also, each subsequence must contain at least two elements. Unlike combinations or subsets, the order is important here, so sequences with the same elements but in different orders are considered different.

**Intuition**

The intuition behind the solution is to explore all possible subsequences while maintaining the non-decreasing order constraint. We can perform a depth-first search (DFS) to go through all potential subsequences. We'll start with an empty list and at each step, we have two choices:

Include the current element in the subsequence if it's greater than or equal to the last included element. This is to ensure the non-decreasing order.

Skip the current element to consider a subsequence without it.

**However, to avoid duplicates, if the current element is the same as the last element we considered and decided not to include, we skip the current element. This is because including it would result in a subsequence we have already considered.**

Starting from the first element, we will recursively call the DFS function to traverse the array. If we reach the end of the array, and our temporary subsequence has more than one element, we include it in our answer.

The key component of this approach is how we handle duplicates to ensure that we only record unique subsequences while performing our DFS.

**Solution Approach**

The implementation of the solution uses a recursive approach known as Depth-First Search (DFS). Let's break down how the given Python code functions:

The function dfs is a recursive function used to perform the depth-first search, starting from the index u in the nums array. This function has the parameters u, which is the current index in the array; last, which is the last number added to the current subsequence t; and t, which represents the current subsequence being constructed.

At the beginning of the dfs function, we check if u equals the length of nums. If it does, we have reached the end of the array. At this point, if the subsequence t has more than one element (making it a valid subsequence), we append a copy of it to the answer list ans.

If the current element, nums[u], is greater than or equal to the last element (last) included in our temporary subsequence (t), we can choose to include the current element in the subsequence by appending it to t and recursively calling dfs with the next index (u + 1) and the current element as the new last.

After returning from the recursive call, the element added is popped from t to backtrack and consider subsequences that do not include this element.

**Additionally, to avoid duplicates, if the current element is different from the last element, we also make a recursive call to dfs without including the current element in the subsequence t, regardless of whether it could be included under the non-decreasing criterion.**

The ans list collects all valid subsequences. The initial DFS call is made with the first index (0), a value that's lower than any element of the array (-1000 in this case) as the initial last value, and an empty list as the initial subsequence.

At the end of the call to the dfs from the main function, ans will contain all possible non-decreasing subsequences of at least two elements, fulfilling the problem's requirement.

Key elements in this solution are the handling of backtracking by removing the last appended element after the recursive calls, and the checking mechanism to avoid duplicates.

The choice of the initial last value is crucial. It must be less than any element we expect in nums, ensuring that the first element can always be considered for starting a new subsequence.

Data structures:

ans: A list to store all the valid non-decreasing subsequences that have at least two elements.

t: A temporary list used to build each potential subsequence during the depth-first search.

Overall, the solution effectively explores all combinations of non-decreasing subsequences through the depth-first search while ensuring that no duplicates are generated.

**Example Walkthrough**

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

Initialize ans as an empty list to store our subsequences and t as an empty list to represent the current subsequence.

Start with the first element 1. Since 1 is greater than our initial last value -1000, we can include 1 in t (which is now [1]) and proceed to the next index.

At the second element 2, it's greater than the last element in t (which is 1), so we can include 2 in t (now [1, 2]) and proceed to the next element. Now our t is a valid subsequence, so we can add it to ans.

Backtrack by popping 2 from t (now [1]) and proceed without including the second element 2.

**At the third element (also 2), we check if we just skipped an element with the same value (which we did). If so, we do not include this element to avoid a duplicate subsequence. If not, since 2 is equal or greater than the last element in t, we could include it in t, and add the resulting subsequence [1, 2] to ans again. But since we are skipping duplicates, we do not do this.**

Instead, we proceed without including this third element 2. Since we have finished going through the array, and t has less than two elements, we don't add it to ans.

Our final ans list contains [1, 2], representing the valid non-decreasing subsequences with at least two elements.

To summarize, our DFS explores these paths:

[1] -> [1, 2] (added to ans) -> [1] (backtrack) -> [1] (skip the second 2) -> [1, 2] (skipped because it would be a duplicate) -> [1] (end of array, not enough elements).

The end result for ans is [[1, 2]].

The key part of this example is that our DFS allowed us to include the first 2, but by using the duplicate check, we did not include the second 2, ensuring our final ans list only included unique non-decreasing subsequences.

**Solution Implementation**

import java.util.ArrayList;

import java.util.List;

class Solution {

private int[] sequence; // Renamed from 'nums' to 'sequence' for better clarity

private List<List<Integer>> subsequences; // List to store the answer subsequences

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

this.sequence = nums; // Assign the given array to the class variable

subsequences = new ArrayList<>(); // Initialize the list to store subsequences

// Start the Depth-First Search (DFS) from index 0 with the last picked element as the smallest integer value

dfs(0, Integer.MIN\_VALUE, new ArrayList<>());

return subsequences; // Return the list of subsequences

}

// Helper method to perform DFS

private void dfs(int index, int lastPicked, List<Integer> currentSubsequence) {

// Base case: if we've reached the end of the sequence

if (index == sequence.length) {

// Check if the current list is a subsequence with more than one element

if (currentSubsequence.size() > 1) {

// If it is, add a copy of it to the list of subsequences

subsequences.add(new ArrayList<>(currentSubsequence));

}

return; // End the current DFS path

}

// If the current element can be picked (is greater or equal to the last picked element)

if (sequence[index] >= lastPicked) {

// Pick the current element by adding it to the currentSubsequence

currentSubsequence.add(sequence[index]);

// Continue the DFS with the next index and the new lastPicked element

dfs(index + 1, sequence[index], currentSubsequence);

// Backtrack: remove the last element added to the currentSubsequence

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

}

// Perform another DFS to explore the possibility of not picking the current element

// Only if the current element isn't equal to the last picked one to avoid duplicates

if (sequence[index] != lastPicked) {

dfs(index + 1, lastPicked, currentSubsequence);

}

}

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the given code mainly depends on the number of recursive calls it can potentially make. At each step, it has two choices: either include the current element in the subsequence or exclude it (as long as including it does not violate the non-decreasing order constraint).

Given that we have n elements in nums, in the worst case, each element might participate in the recursion twice—once when it is included and once when it is excluded. This gives us an upper bound of O(2^n) on the number of recursive calls. However, the condition if nums[u] != last prevents some recursive calls when the previous number is the same as the current, which could lead to some pruning, but this pruning does not affect the worst-case complexity, which remains exponential.

Therefore, the time complexity of the code is O(2^n).

**Space Complexity**

The space complexity consists of two parts: the space used by the recursion call stack and the space used to store the combination t.

The space used by the recursion stack in the worst case would be O(n) because that's the maximum depth the recursive call stack could reach if you went all the way down including each number one after the other.

The space required for storing the combination t grows as the recursion deepens, but since the elements are only pointers or references to integers and are reused in each recursive call, this does not significantly contribute to the space complexity. However, the temporary arrays formed during the process, which are then copied to ans, could increase the storage requirements. ans itself can grow up to O(2^n) in size, in the case where every possible subsequence is valid.

Thus, the space complexity is dominated by the size of the answer array ans and the recursive call stack, leading to a total space complexity of O(n \* 2^n), with n being the depth of the recursion (call stack) and 2^n being the size of the answer array in the worst case.

**Refer to**

[L90.P11.2.Subsets II (Ref.L491,L78)](note://3BBFC543DFF4475AB12B3EE110E58669)