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

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**.

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
```

```java
        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}

```java
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}

```java
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)

```java
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);
        }
    }
}
```

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]

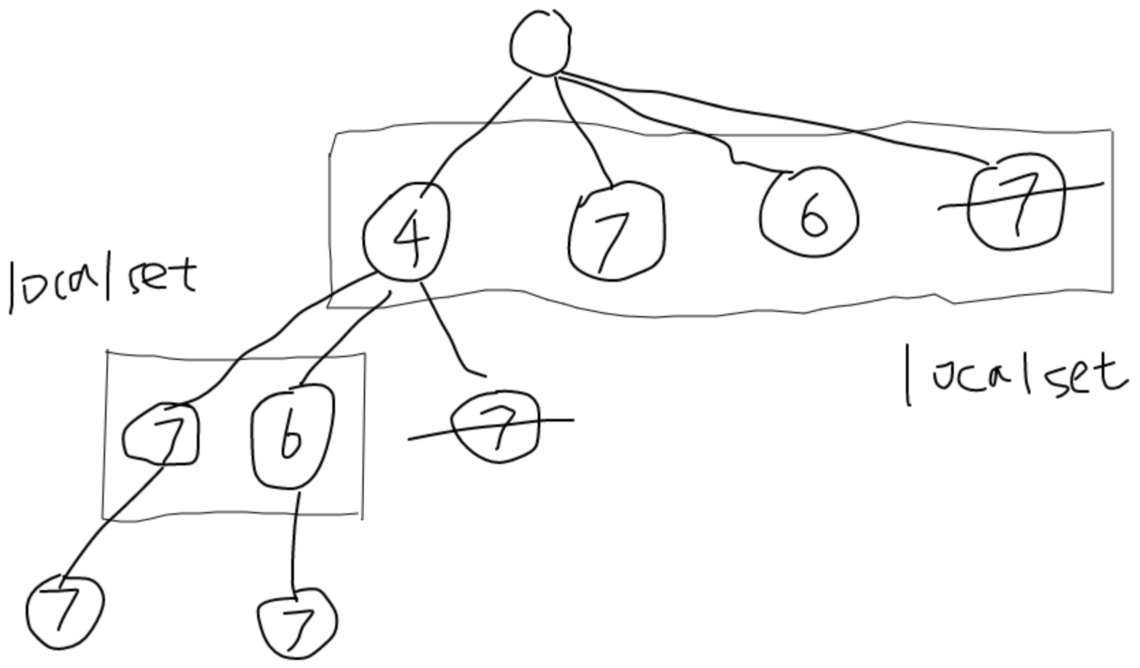
Expected: [4,4]

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:

```java
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;
    }
    void find(int p, vector<int>& nums, vector<int>& one, vector<vector<int>>& 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

```java
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);
            }
        }
    }
}
```

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

```java
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);
        }
        if(index > 0 && tmp.size() > 0 && tmp.get(tmp.size() - 1) == nums[index]) {
            return;
        }
        // Not pick
        helper(nums, result, tmp, index + 1);
    }
}
```

Refer to

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

The set is needless:

```java
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

```java
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);
        }
        // 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);
        }
    }
}
```

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

```java
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);
        }
    }
}
```

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!

This means that we have to check the last element in the tmp list and if they are identical, we disallow not-pick as an option for the current layer of recursion. 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.

```java
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);
        }
    }
}
```

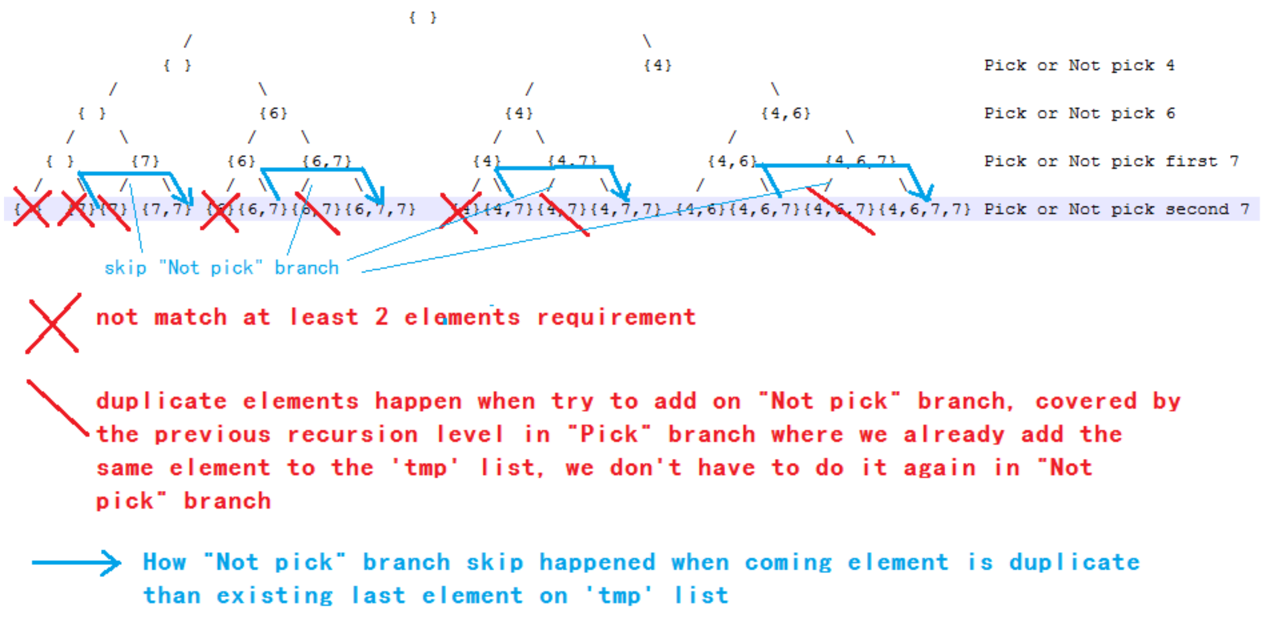
Two step by step examples:

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 symbol 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

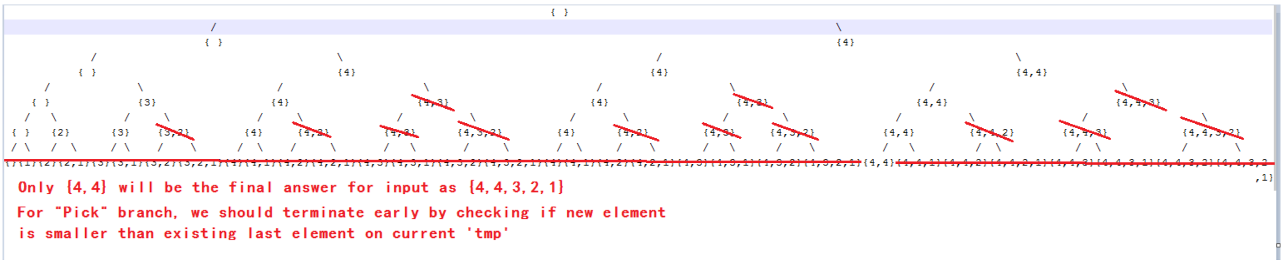


{ }
{ }
{4}
Pick or Not pick 4
{ }
{6}
{4}
{4,6}
Pick or Not pick 6
{ }
{7}
{6}
{6,7}
{4}
{4,7}
{4,6}
{4,6,7}
Pick or Not pick first 7
{ }{7}{7}{7,7}{6}{6,7}{6,7}{6,7,7}{4}{4,7}{4,7}{4,7,7}{4,6}{4,6,7}{4,6,7}{4,6,7,7}
Pick or Not pick second 7
skip "Not pick" branch
not match at least 2 elements requirement
duplicate elements happen when try to add on "Not pick" branch, covered by the previous recursion level in "Pick" branch where we already add the same element to the 'tmp' list, we don't have to do it again in "Not pick" branch
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}
```


Only {4,4} will be the final answer for input as {4,4,3,2,1}

For "Pick" branch, we should terminate early by checking if new element
is smaller than existing last element on current 'tmp'

Refer to

L90.P11.2.Subsets II (Ref.L491,L78)