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.
```

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}}
```

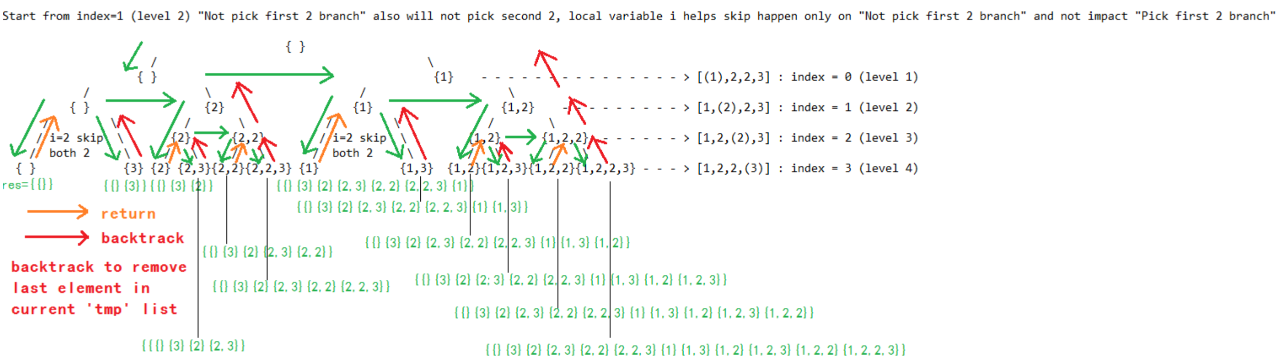
```
result={{}{3}{2}{2,3}{2,2}{2,2,3}{1}}
result={{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}}
result={{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}{1,2}}
result={{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}{1,2}{1,2,3}}
result={{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}{1,2}{1,2,3}{1,2,2}}
result={{}{3}{2}{2,3}{2,2}{2,2,3}{1}{1,3}{1,2}{1,2,3}{1,2,2}{1,2,2,3}}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

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

```java
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

```java
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

Refer to

L491.Increasing Subsequences (Ref.L90)

L78.11.1.Subsets (Ref.L90)