https://leetcode.com/problems/subsets/

Given an integer array nums of **unique** elements, 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 = [1,2,3]

Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

Example 2:

Input: nums = [0]

Output: [[],[0]]

Constraints:

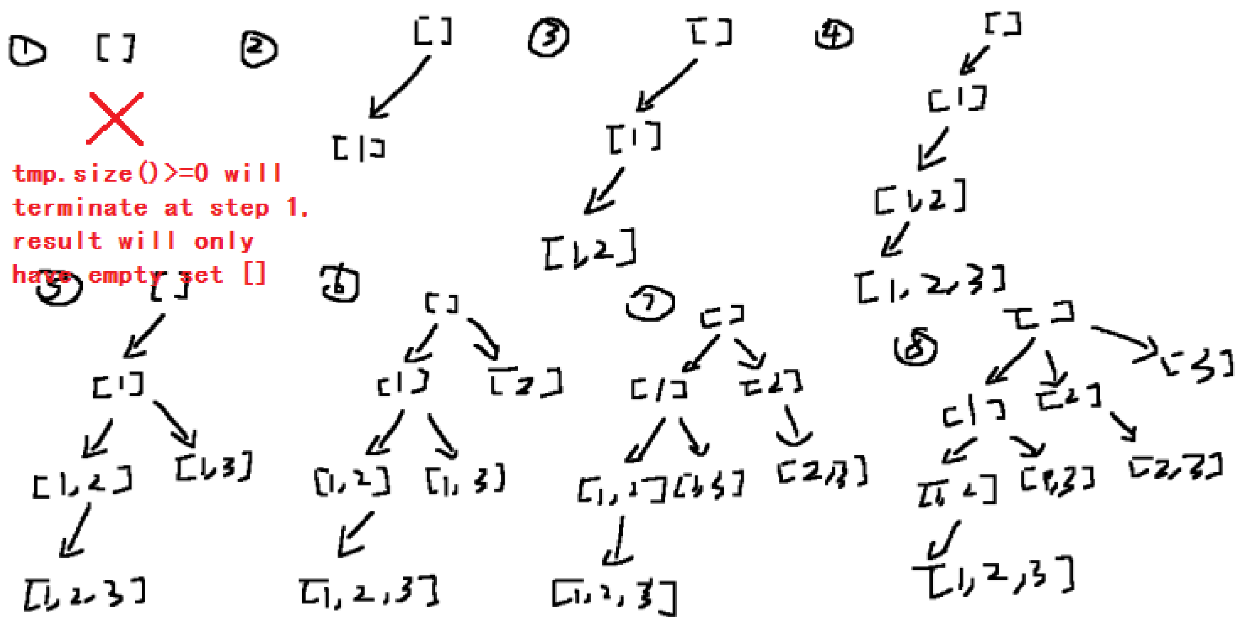
- 1 <= nums.length <= 10
- -10 <= nums[i] <= 10
- All the numbers of nums are **unique**.

Attempt 1: 2022-10-7

Wrong Solution:

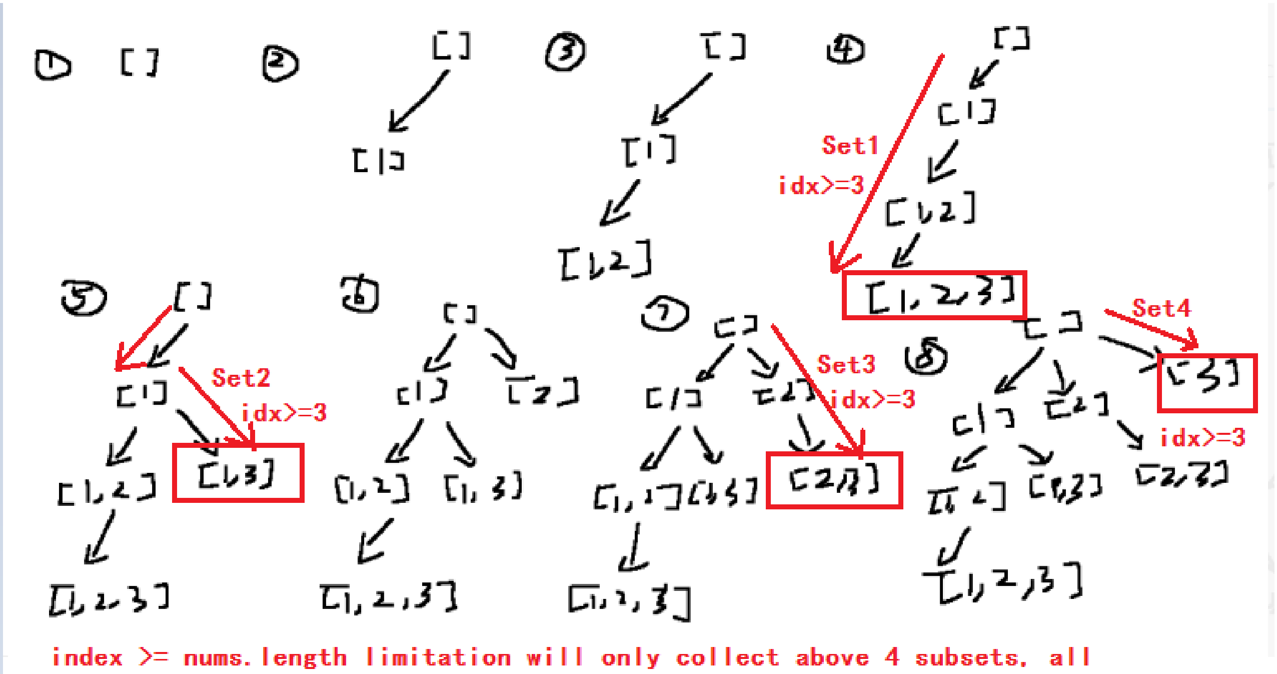
Adding unnecessary limitation for "result.add(new ArrayList<Integer>(tmp));" and then direct return, no limitation required because we have to collect all subsets, the direct return will terminate the search at early stage, the recursion will not continue and miss subsets

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


①
②
③
④
⑤
⑥
⑦
⑧
tmp.size()>=0 will terminate at step 1, result will only have empty set []

```
Input: [1,2,3]
Wrong output: [[]]
Expect output: [[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]

class Solution {
    public List<List<Integer>> subsets(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) {
        // 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++) {
            tmp.add(nums[i]);
            helper(nums, result, tmp, i + 1);
            tmp.remove(tmp.size() - 1);
        }
    }
}
```

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

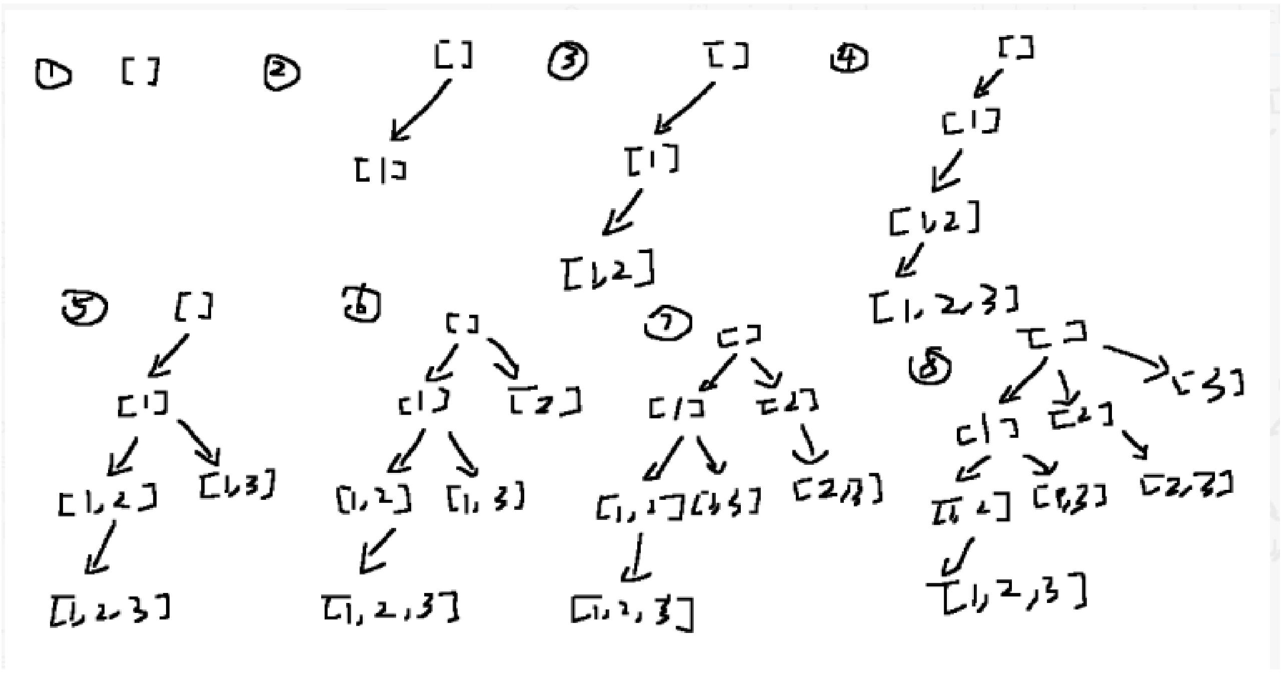


index >= nums.length limitation will only collect above 4 subsets, all includes element 3 because only when includes element 3 will match index >= nums.length, but this limitation will cause miss for all other subsets

```
Input: [1,2,3]
Wrong output: [[1, 2, 3], [1, 3], [2, 3], [3]]
Expect output: [[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]

class Solution {
    public List<List<Integer>> subsets(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) {
        // 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++) {
            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>> subsets(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) {
        result.add(new ArrayList<Integer>(tmp));
        for(int i = index; i < nums.length; i++) {
                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.
```

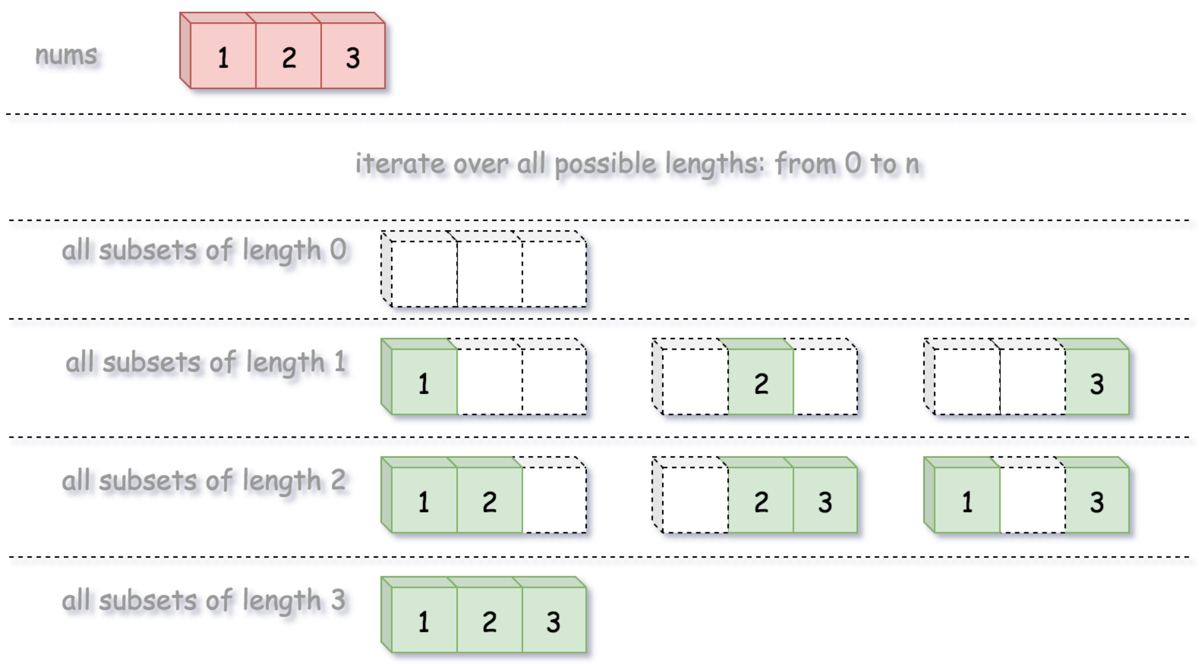
Refer to

https://leetcode.com/problems/subsets/solution/

Power set is all possible combinations of all possible *lengths*, from 0 to n.

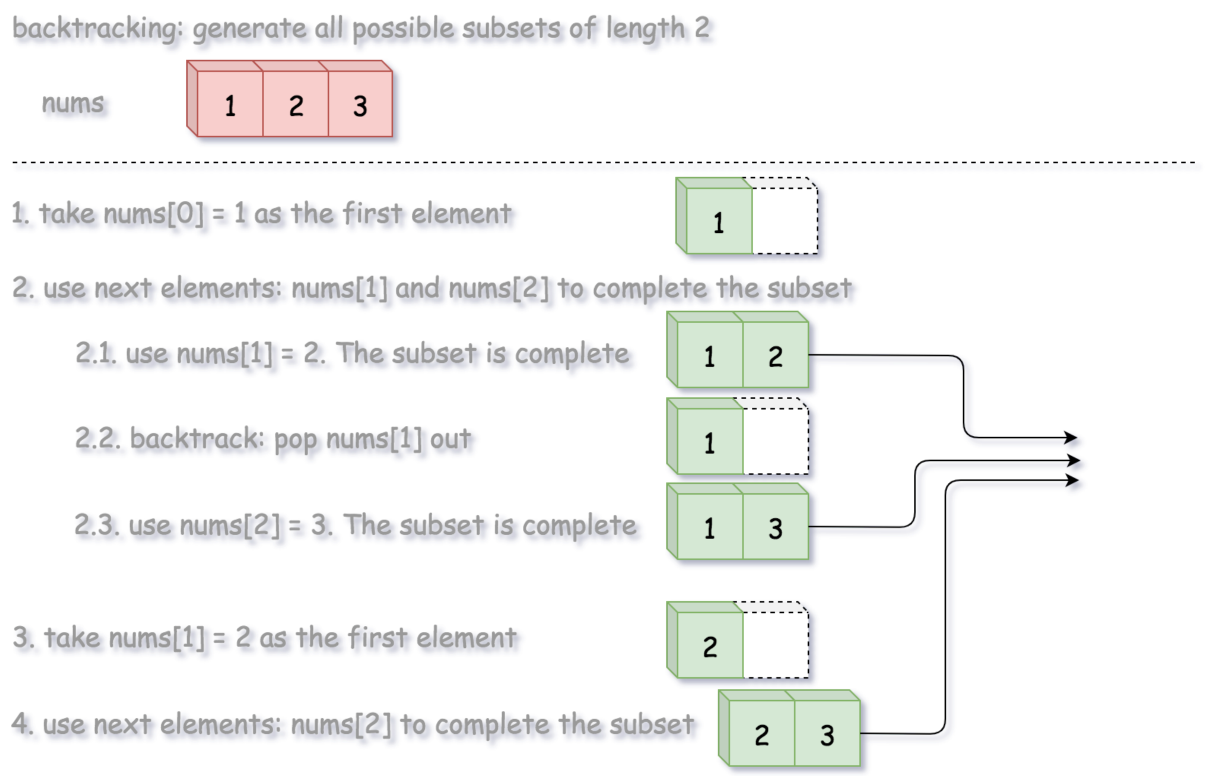
Given the definition, the problem can also be interpreted as finding the *power set* from a sequence.

So, this time let us loop over *the length of combination*, rather than the candidate numbers, and generate all combinations for a given length with the help of *backtracking* technique.



Backtracking is an algorithm for finding all solutions by exploring all potential candidates. If the solution candidate turns to be *not* a solution (or at least not the *last* one), backtracking

algorithm discards it by making some changes on the previous step, *i.e. backtracks* and then try again.


backtracking: generate all possible subsets of length 2
nums
1
2
3
1. take nums[0] = 1 as the first element
1
2. use next elements: nums[1] and nums[2] to complete the subset
2.1. use nums[1] = 2. The subset is complete
1
2
2.2. backtrack: pop nums[1] out
1
2.3. use nums[2] = 3. The subset is complete
1
3
3. take nums[1] = 2 as the first element
2
4. use next elements: nums[2] to complete the subset
2
3

Algorithm

We define a backtrack function named backtrack(first, curr) which takes the index of first element to add and a current combination as arguments.

- If the current combination is done, we add the combination to the final output.
- Otherwise, we iterate over the indexes i from first to the length of the entire sequence n.
    - Add integer nums[i] into the current combination curr.
    - Proceed to add more integers into the combination : backtrack(i + 1, curr).
    - Backtrack by removing nums[i] from curr.

Implementation

```
class Solution {
  List<List<Integer>> output = new ArrayList();
  int n, k;
  public void backtrack(int first, ArrayList<Integer> curr, int[] nums) {
    // if the combination is done
    if (curr.size() == k) {
      output.add(new ArrayList(curr));
      return;
    }
    for (int i = first; i < n; ++i) {
      // add i into the current combination
      curr.add(nums[i]);
      // use next integers to complete the combination
      backtrack(i + 1, curr, nums);
      // backtrack
      curr.remove(curr.size() - 1);
    }
  }
  public List<List<Integer>> subsets(int[] nums) {
    n = nums.length;
    for (k = 0; k < n + 1; ++k) {
      backtrack(0, new ArrayList<Integer>(), nums);
    }
    return output;
  }
}

Complexity Analysis
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 (10min)

```
class Solution {
  public List<List<Integer>> subsets(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) {
        // Base Case
        if(index >= nums.length) {
            result.add(new ArrayList<Integer>(tmp));
            return;
        }
        // Not pick element / Pick element order can switch
        // Not pick element at index
        helper(nums, result, tmp, index + 1);
        // Pick element at index
        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.
```

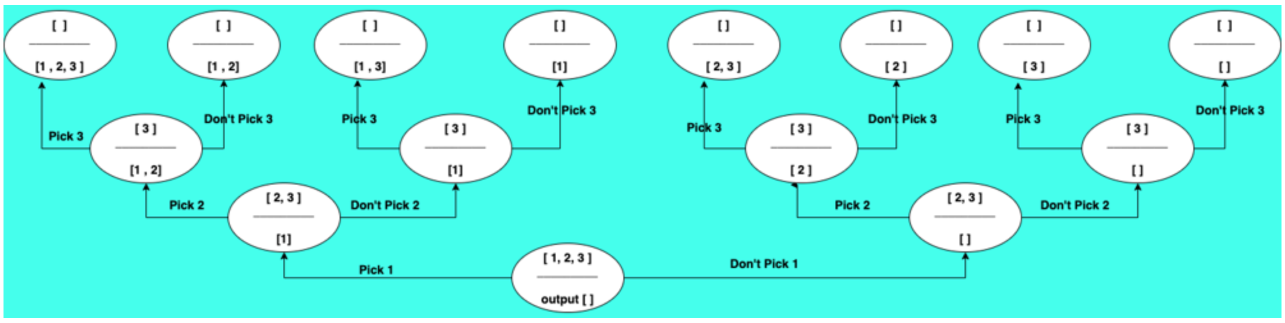
```
We need to create a copy of the current set because we reuse the original one to build all the
valid subsets. This copy costs O(n) and it is performed at each call of the recursive function,
which is called 2^n times as mentioned in above. So total time complexity is O(n x 2^n).
```

Refer to

https://leetcode.com/problems/subsets/discuss/1766675/Java-Intuition-of-Approach-or-Backtracking

Intuition

This falls into a set of classic problems that can be solved using "pick"- "don't pick" approach.

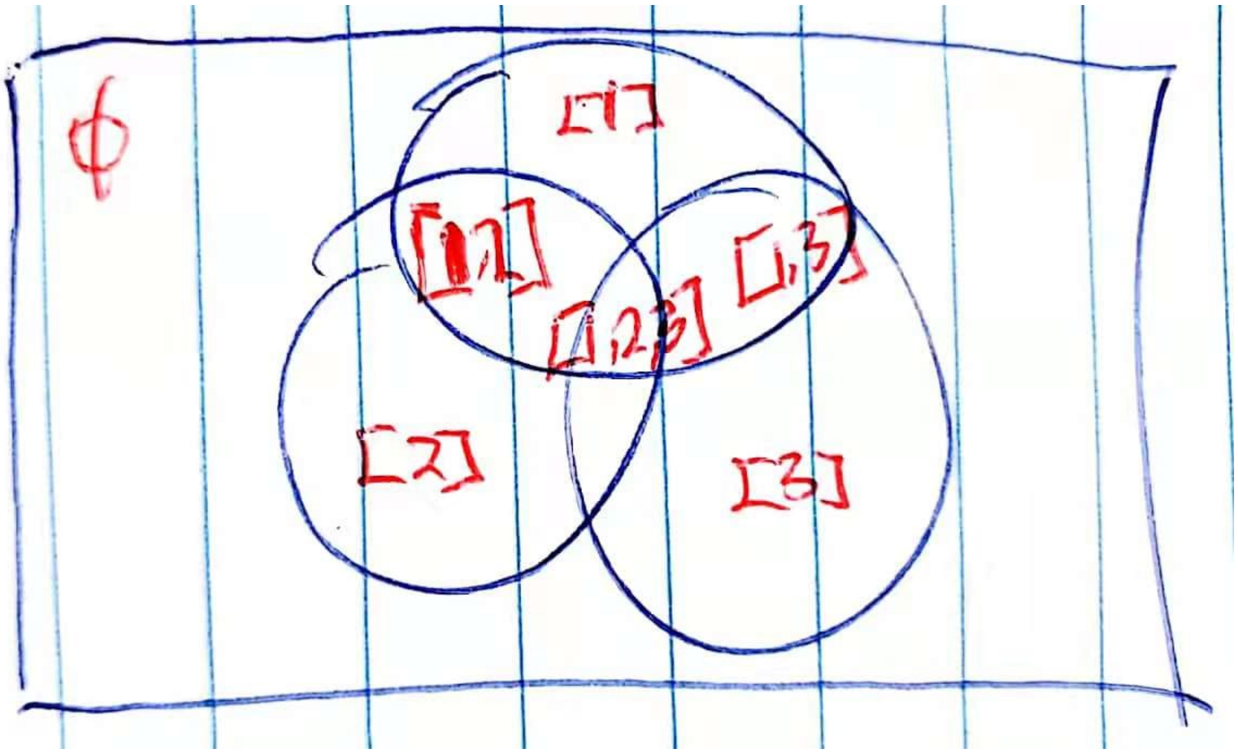


As through observation, it is clear that we start with having a choice to pick or not pick from the first element and then, we propagate this choice down the tree, So

1. we either pick the element and move ahead (increment the index) or,
2. we don't pick the element and move ahead

```java
class Solution {
    List<List<Integer>> result;
    public List<List<Integer>> subsets(int[] nums) {
        result = new ArrayList();
        if(nums==null || nums.length==0) return result;
        subsets(nums,new ArrayList<Integer>(), 0);
        return result;
    }

    private void subsets(int[] nums, ArrayList<Integer> temp, int index) {
        // base condition
        if(index >= nums.length) {
            result.add(new ArrayList<>(temp));
            return;
        }
        // main logic
        // case 1 : we pick the element
        temp.add(nums[index]);
        subsets(nums, temp, index+1); // move ahead
        temp.remove(temp.size()-1);
        // case 2 : we don't pick the element ( notice, we did not add the current element in our temporary list
        subsets(nums, temp, index+1); // move ahead
    }
}
```

Refer to
https://leetcode.com/problems/subsets/discuss/388568/Subsets-I-and-II-Java-Solution-with-Detailed-Explanation-and-Comments-(Recursion-and-Iteration)
The power set of a set S is the set of all subsets of S, including both the empty set emptyset and S itself. The power set of {1, 2, 3} is graphically illustrated as follows.



Backtracking
The idea is that we loop over the number list. For each number, we have two choices: pick it, or not. For example, in [1, 2, 3], we pick 1 and then do the same thing for the subproblem [2, 3]; and we

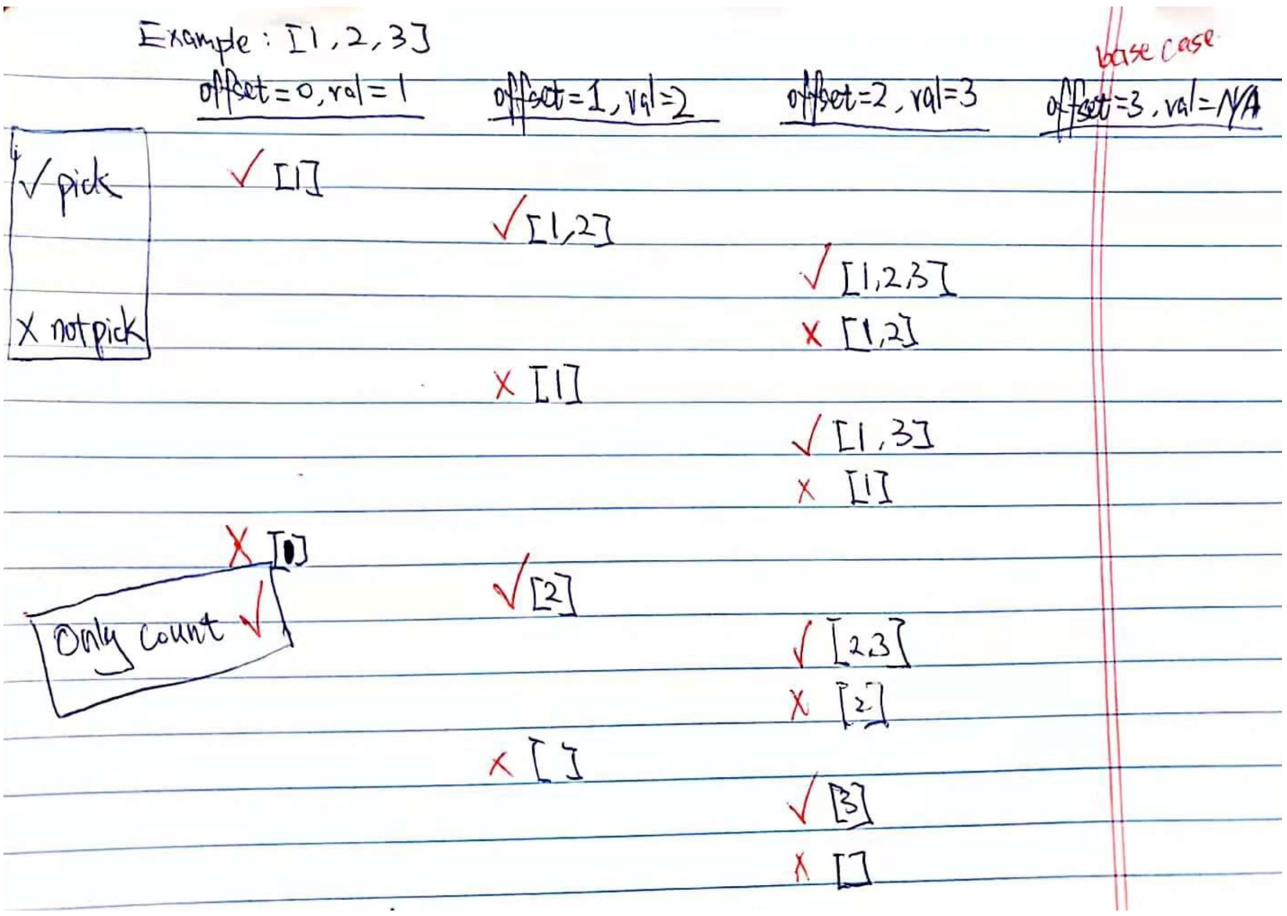
don't pick 1 and then do the same thing for the subproblem [2, 3].

The size of subproblems is decreasing. When picking 2, the subproblem becomes [3] instead of [1, 3].

Consider the following questions:

- What is the base case?
- When do we add the list to the result?

Here is an illustration of recursive process on [1, 2, 3].



Note: Remember to add empty set manually.

```java
public List<List<Integer>> subsets(int[] nums) {
  List<List<Integer>> result = new ArrayList<>();
  List<Integer> numList = new ArrayList<>();

  result.add(new ArrayList<>()); // empty set
  subsets(0, nums, numList, result);
  return result;
}
private void subsets(int offset, int[] nums, List<Integer> numList, List<List<Integer>>
 result) {
  if (offset >= nums.length) {
    return;
  }
  int val = nums[offset];
  // pick
  numList.add(val);
  subsets(offset + 1, nums, numList, result);
  // add to result
  result.add(new ArrayList<>(numList));
  // not pick
  numList.remove(numList.size() - 1);
  subsets(offset + 1, nums, numList, result);
}
```

Solution 3: No Backtracking but deep copy required to avoid modify original object (10min)

```java
class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        // Manually add empty set
        result.add(new ArrayList<Integer>());
        helper(nums, result, new ArrayList<Integer>(), 0);
        return result;
    }

    private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, int index) {
        // Base Case
        if(index >= nums.length) {
            return;
        }
        // Not pick element / Pick element order can switch
        // Not pick element at index
        helper(nums, result, tmp, index + 1);
        // Pick element at index
        // Create a deep copy instead of modifying 'tmp' then no need backtrack
        List<Integer> copy = new ArrayList<Integer>(tmp);
        copy.add(nums[index]);
```

```
            result.add(copy);
            helper(nums, result, copy, index + 1);
        }
}
```

Refer to
https://segmentfault.com/a/1190000003498803

深度优先搜索

复杂度

时间 O(NlogN) 空间 O(N) 递归栈空间

思路

这道题可以转化为一个类似二叉树的深度优先搜索。二叉树的根是个空集，它的左子节点是加上第一个元素产生的集合，右子节点不加上第一个元素所产生的集合。以此类推，左子节点的左子节点是加上第二个元素，左子节点的右子节点是不加上第二个元素。而解就是这个二叉树所有的路径，我们要做的就是根据加，或者不加下一元素，来产生一个新的集合，然后继续递归直到终点。另外需要先排序以满足题目要求。

注意

- 不需要先排序
- 如果递归之前先将空集加入结果，那么递归尽头就不需要再加一次空集。反之则需要。后面详细叙述在

How to avoid manually add empty set ?

- LinkedList在这里效率要高于ArrayList。
- 新的集合要new一个新的list，防止修改引用。

代码

```
public class Solution {
    public List<List<Integer>> subsets(int[] S) {
        Arrays.sort(S);
```

```java
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        List<Integer> tmp = new ArrayList<Integer>();
        //先加入空集
        res.add(tmp);
        helper(S, 0, res, tmp);
        return res;
    }

    private void helper(int[] S,int index,List<List<Integer>> res, List<Integer> tmp){
        if(index>=S.length) return;
        // 不加入当前元素产生的集合，然后继续递归
        helper(S, index+1, res, tmp);
        List<Integer> tmp2 = new ArrayList<Integer>(tmp);
        tmp2.add(S[index]);
        res.add(tmp2);
        // 加入当前元素产生的集合，然后继续递归
        helper(S, index+1, res, tmp2);
    }
}
```

Important Tips

1. How to avoid manually add empty set ?

The critical statement added to avoid manually add empty set before recursion is happening in recursion base case, when first condition match happening (index == nums.length), the empty 'tmp' list we pass through the whole recursion from // Not pick element at index helper(nums, result, tmp, index + 1) statement will keep the empty 'tmp' list as is (no change still keep as empty list) and then result.add(new ArrayList<Integer>(tmp)) will add a deep copy of 'tmp' into result

```java
        // Base Case
        if(index >= nums.length) {
            // Here is the critical statement added to avoid manually add empty set before recursion
            result.add(new ArrayList<Integer>(tmp));
            return;
        }
```

Full code based on Solution 3 without manually add empty set

```java
class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        // Manually add empty set not required
        //result.add(new ArrayList<Integer>());
        helper(nums, result, new ArrayList<Integer>(), 0);
        return result;
    }

    private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, int index) {
        // Base Case
        if(index >= nums.length) {
            // The critical statement added to avoid manually add empty set before recursion
            result.add(new ArrayList<Integer>(tmp));
            return;
        }
        // Not pick element / Pick element order can switch
        // Not pick element at index
        helper(nums, result, tmp, index + 1);
        // Pick element at index
        // Create a deep copy instead of modifying 'tmp' then no need backtrack
        List<Integer> copy = new ArrayList<Integer>(tmp);
        copy.add(nums[index]);
        // When add critical statement in base case, have to remove below statement to avoid
        // duplicate results
        //result.add(copy);
        helper(nums, result, copy, index + 1);
    }
}
```

2. The difference between Backtracking style 1 with for loop and Backtracking style 2 without for loop

Progress from Solution 1: Backtracking style 1

```
class Solution {
    public List<List<Integer>> subsets(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) {
        result.add(new ArrayList<Integer>(tmp));
        for(int i = index; i < nums.length; i++) {
            tmp.add(nums[i]);
            helper(nums, result, tmp, i + 1);
            tmp.remove(tmp.size() - 1);
        }
    }
}

nums={1,2,3}
---------------------------------------------
Round 1:
index=0,tmp={}
helper(nums,result,{},0)
result={{}}
i=index=0
tmp.add(nums[0]) -> tmp={1}
---------------------------------------------
Round 2:
index=0+1=1,tmp={1}
helper(nums,result,{1},1)
result={{},{1}}
i=index=1
tmp.add(nums[1]) -> tmp={1,2}
---------------------------------------------
Round 3:
index=1+1=2,tmp={1,2}
helper(nums,result,{1,2},2)
result={{},{1},{1,2}}
i=index=2
```

```
tmp.add(nums[2]) -> tmp={1,2,3}
---------------------------------------------
Round 4:
index=2+1=3,tmp={1,2,3}
helper(nums,result,{1,2,3},3)
result={{},{1},{1,2},{1,2,3}}
i=2 -> i++=3 == nums.length skip for loop
---------------------------------------------
Round 5: Backtrack to Round 3 statistics
index=2,tmp={1,2,3} -> tmp.size()-1=3-1=2
tmp.remove(2) -> tmp={1,2}
i=2 -> i++=3 == nums.length skip for loop
---------------------------------------------
Round 6: Backtrack to Round 2 statistics
index=1,tmp={1,2} -> tmp.size()-1=2-1=1
tmp.remove(1) -> tmp={1}
i=1 -> i++=2
tmp.add(nums[2]) -> tmp={1,3}
---------------------------------------------
Round 7:
index=3,tmp={1,3}
helper(nums,result,{1,3},3)
result={{},{1},{1,2},{1,2,3},{1,3}}
i=index=3 -> i == nums.length skip for loop
---------------------------------------------
Round 8: Backtrack to Round 2 statistics
index=1,tmp={1,3} -> tmp.size()-1=2-1=1
tmp.remove(1) -> tmp={1}
i=2 -> i++=3 == nums.length skip for loop
---------------------------------------------
Round 9: Backtrack to Round 1 statistics
index=0,tmp={1} -> tmp.size()-1=1-1=0
tmp.remove(0) -> tmp={}
i=0 -> i++=1
tmp.add(nums[1]) -> tmp={2}
---------------------------------------------
Round 10:
index=2,tmp={2}
helper(nums,result,{2},2)
result={{},{1},{1,2},{1,2,3},{1,3},{2}}
```

```
i=index=2
tmp.add(nums[2]) -> tmp={2,3}
---------------------------------------------
Round 11:
index=3,tmp={2,3}
helper(nums,result,{2,3},3)
result={{},{1},{1,2},{1,2,3},{1,3},{2},{2,3}}
i=index=3 == nums.length skip for loop
---------------------------------------------
Round 12: Backtrack to Round 2 statistics
index=2,tmp={2,3} -> tmp.size()-1=2-1=1
tmp.remove(1) -> tmp={2}
i=2 -> i++=3 == nums.length skip for loop
---------------------------------------------
Round 13: Backtrack to Round 1 statistics
index=0,tmp={2} -> tmp.size()-1=1-1=0
tmp.remove(0) -> tmp={}
i=1 -> i++=2
tmp.add(nums[2]) -> tmp={3}
---------------------------------------------
Round 14:
index=3,tmp={3}
helper(nums,result,{3},3)
result={{},{1},{1,2},{1,2,3},{1,3},{2},{2,3},{3}}
i=index=3 == nums.length skip for loop
---------------------------------------------
Round 15: Backtrack to Round 1 statistics
index=0,tmp={3} -> tmp.size()-1=1-1=0
tmp.remove(0) -> tmp={}
i=2 -> i++=3 == nums.length skip for loop
---------------------------------------------
End
=============================================
Result time elapsed statistics:
result={{}}
result={{},{1}}
result={{},{1},{1,2}}
result={{},{1},{1,2},{1,2,3}}
result={{},{1},{1,2},{1,2,3},{1,3}}
result={{},{1},{1,2},{1,2,3},{1,3},{2}}
```

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

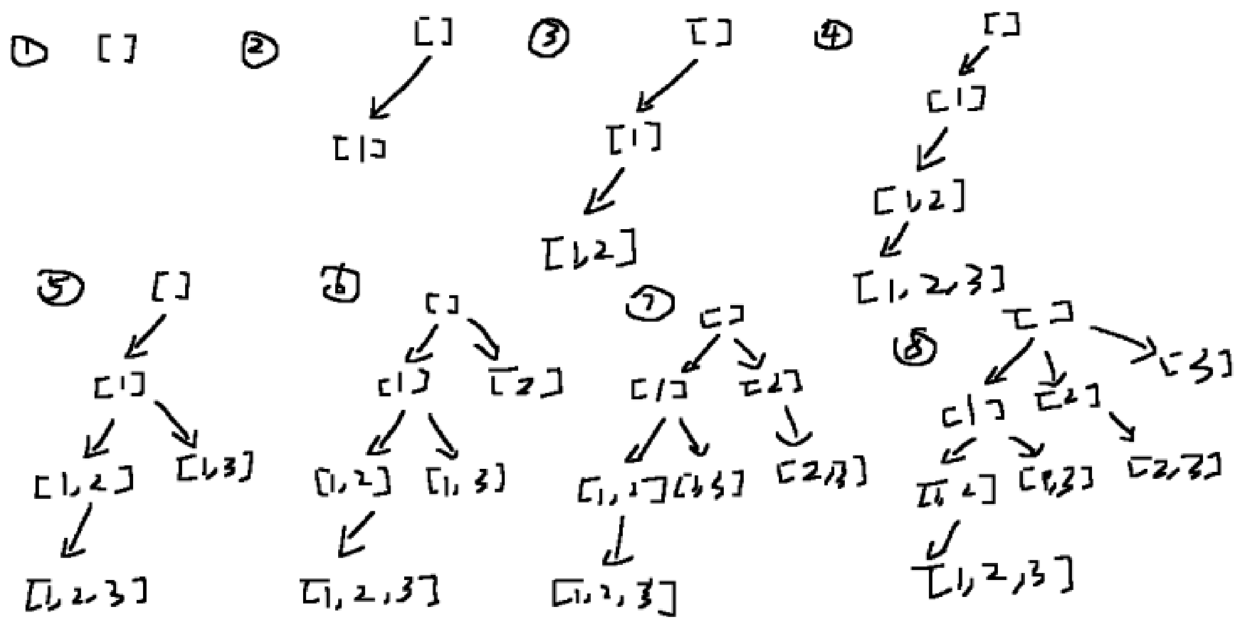
How Backtracking style 1 works ?

Each recursion level focuses on all the following elements. We scan through all the following elements and decide whether to choose or not choose that element. (Every level split into N branches.)

Below two pictures showing how Backtracking style 1 traverse step by step:

Picture 1

Once nums[i] arrived at end, remove the last element and go back to last level until all possible solutions are stored



Progress from Solution 2: Backtracking style 2

```
class Solution {
    public List<List<Integer>> subsets(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) {
        // Base Case
        if(index >= nums.length) {
            result.add(new ArrayList<Integer>(tmp));
            return;
        }
        // Not pick element / Pick element order can switch
        // Not pick element at index
        helper(nums, result, tmp, index + 1);
        // Pick element at index
        tmp.add(nums[index]);
        helper(nums, result, tmp, index + 1);
        tmp.remove(tmp.size() - 1);
    }
}

nums={1,2,3}
---------------------------------------------
Round 1:
index=0,tmp={}
helper(nums,result,{},0)
index=0 < nums.length=3
---------------------------------------------
Round 2:
index=1,tmp={}
helper(nums,result,{},0+1)
index=1 < nums.length=3
---------------------------------------------
Round 3:
index=2,tmp={}
helper(nums,result,{},1+1)
index=2 < nums.length=3
---------------------------------------------
Round 4:
index=3,tmp={}
helper(nums,result,{},2+1)
index=3 == nums.length=3
result={{}}
```

```
Return to Round 3 statistics
index=2,tmp={}
tmp.add(nums[2]) -> tmp={3}
---------------------------------------------
Round 5:
helper(nums,result,{3},2+1)
index=3 == nums.length=3
result={{},{3}}
Return to Round 3 statistics
tmp.remove(tmp.size()-1)=tmp.remove(1-1)=tmp.remove(0)={}
Finish last statement in helper and return to Round 2 statistics
---------------------------------------------
Round 6:
index=1,tmp={}
tmp.add(nums[1]) -> tmp={2}
helper(nums,result,{2},1+1)
index=2 < nums.length=3
---------------------------------------------
Round 7:
helper(nums,result,{2},2+1)
index=3 == nums.length=3
result={{},{3},{2}}
Return to Round 6 statistics
---------------------------------------------
Round 8:
index=2,tmp={2}
tmp.add(nums[2]) -> tmp={2,3}
helper(nums,result,{2,3},2+1)
---------------------------------------------
Round 9:
index=3,tmp={2,3}
index=3 == nums.length=3
result={{},{3},{2},{2,3}}
Return to Round 8 statistics
tmp.remove(tmp.size()-1)=tmp.remove(2-1)=tmp.remove(1)={2}
Finish last statement in helper and return to Round 6 statistics
---------------------------------------------
Round 10:
index=1,tmp={2}
tmp.remove(tmp.size()-1)=tmp.remove(1-1)=tmp.remove(0)={}
```

```
Finish last statement in helper and return to Round 1 statistics
---------------------------------------------
Round 11:
index=0,tmp={}
tmp.add(nums[0]) -> tmp={1}
helper(nums,result,{1},0+1)
index=1 < nums.length=3
---------------------------------------------
Round 12:
index=2,tmp={1}
helper(nums,result,{1},1+1)
index=2 < nums.length=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}}
---------------------------------------------
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}}
```

How Backtracking style 2 works ?

Each recursion level focuses on one element, we need to decide choose or not choose this element (Every level split into 2 branches), which means ALL GENERATED SUBSETs based on current set either include this element or NOT include this element

There is one video explains how we distribute problem into choose / not choose two branches very well

Subsets II - Backtracking - Leetcode 90 - Python **[timeline = 11:25]**

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

Below two pictures showing how Backtracking style 2 traverse step by step

Picture 1

A simple visualization of the recursion tree helps understanding the subset concept. We go from left to right in the nums array. At each iteration we can either include or exclude the element at nums[i]. This gives 2 possible choices at each node in the tree. When i reaches the end of the nums array, we found a valid subset, so we add it to our output list. We have 2^n possible subsets where 2 is the number of possible choices at each node (each node) and n is the number of elements in nums.

```
//                       {  }
//               /                     \
//             { }                     {1}  - - - - - - - - - - - - - - > [(1),2,3] : i = 0 (level 1)
//          /       \              /          \
//       { }        {2}         {1}          {1,2} - - - - - - - - - - - > [1,(2),3] : i = 1 (level 2)
//      /   \      /   \       /   \        /   \
//    { }   {3}  {2}  {2,3}  {1}  {1,3}  {1,2}  {1,2,3} - - - - - - - - > [1,2,(3)] : i = 2 (level 3)
//
// e.g
// On level 1, left node {} means not choose element 1, all generated subsets based on level 1 left node {} which
// not choose element 1 will also NOT include element 1 (e.g all nodes in left subtree not include element 1)
//             { }
//          /       \
//       { }        {2}
//      /   \      /   \
//    { }   {3}  {2}  {2,3}
// Oppositely, right node {1} means choose element 1, all generated subsets based on level 1 right node {1} which
// choose element 1 will also include element 1 (e.g all nodes in right subtree include element 1 on each node)
//             {1}
//        /          \
//      {1}         {1,2}
//     /   \        /   \
//   {1}  {1,3}  {1,2}  {1,2,3}
// On level 2, two left nodes {} and {1} as child of level 1's two nodes respectively which not choose element 2
// have all their subsets not include {2}
//       { }              {1}
//      /   \     and    /   \
//    { }   {3}        {1}  {1,3}
// Oppositely, two right nodes {2} and {1,2} as child of level 1's two nodes respectively which choose element 2
// have all their subsets include {2}
//       {2}              {1,2}
//      /   \     and     /   \
//    {2}  {2,3}       {1,2}  {1,2,3}
// Same logic for level 3
```

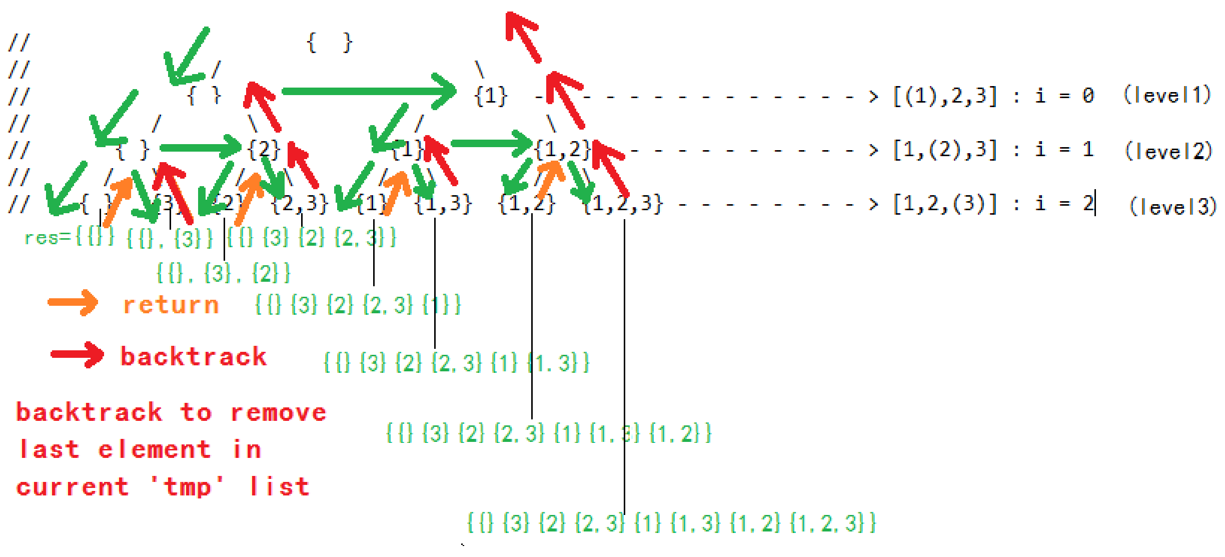
Why besides choose or not choose one element the "Backtracking" technic still necessary in style 2 ?

Because the problem requires to get all subset combinations, and only "Backtracking" technic able to modify traversal object back to previous status and prepare for new modification (for every drill down, we should clear the last path's state, acknowledge as backtracking), only choose or not choose one element cannot do this

If no backtracking to remove last element in current 'tmp' list during traversal, not able to append different element based on either include or exclude current element to get all subset combinations e.g.

For left subtree start from level 1 which means not include element 1, if no backtracking, 'tmp' list not able to change from {3} to {2} by removing last element 3 from 'tmp' and append different element 2, then will miss {2} and {2, 3} subsets in final result.

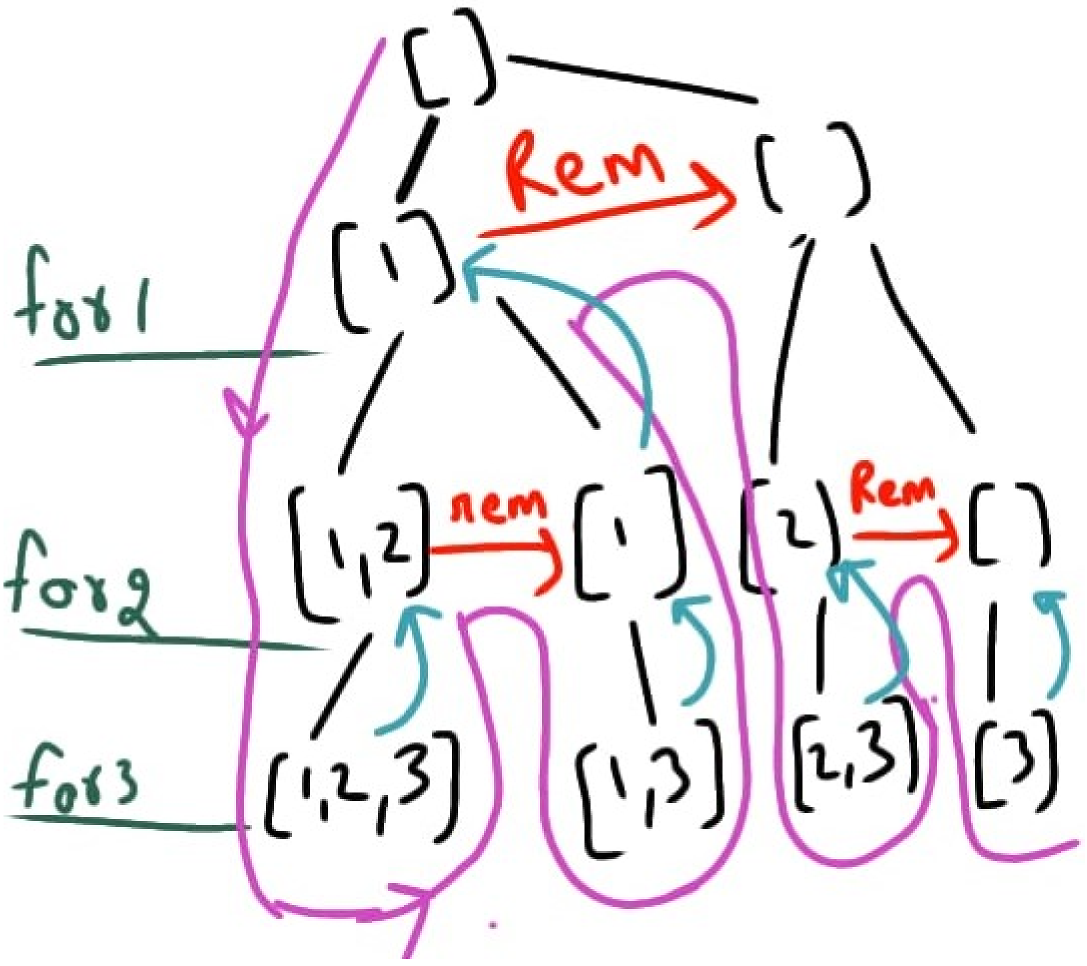
For right subtree start from level 1 which means include element 1, if no backtracking, 'tmp' list not able to change from {1, 3} to {1, 2} by removing last element 3 from 'tmp' and append different element 2, then will miss {1, 2} and {1, 2, 3} subsets in final result



Picture 2

How it traversal from {{}} to {{},{1},{1,2},{1,2,3},{1,3},{2},{2,3},{3}}, order matters, each element fill into result array follow purple line one by one, the only difference than **Solution 2: Backtracking style 2** is switching Not pick element / Pick element order, in **Solution 2: Backtracking style 2** Not pick element first, Pick element second, below snapshot explain how Pick element first, Not pick element second looks like, **Picture 3** below also same.

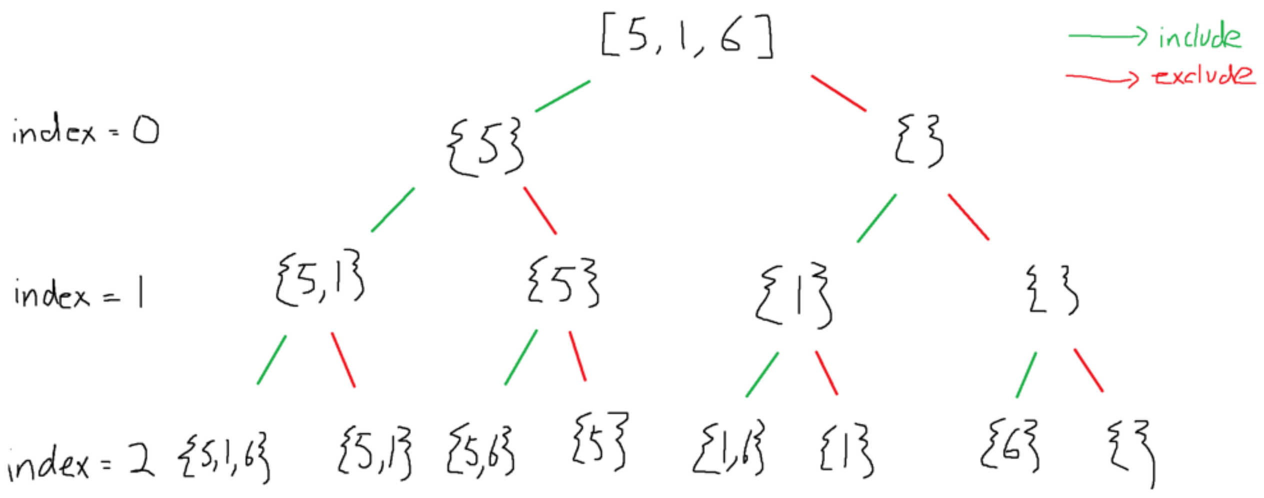
```java
class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        helper(nums, result, new ArrayList<Integer>(), 0);
        return result;
    }

    private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, int
index) {
        // Base Case
        if(index >= nums.length) {
            result.add(new ArrayList<Integer>(tmp));
            return;
        }
        // Not pick element / Pick element order can switch
        // Pick element at index
        tmp.add(nums[index]);
        helper(nums, result, tmp, index + 1);
        tmp.remove(tmp.size() - 1);
        // Not pick element at index
        helper(nums, result, tmp, index + 1);
    }
}
```



Picture 3

We can recursively generate all possible subsets given an array. For each element in the array, you will have two choices: Include the current element in the array in the generated subset, or withhold the current element from the generated subset.

Note: the input array is different than {1,2,3} but the idea exactly same, solution picture related problem L1863



Superficially, we can see the result element order is quite different, for style 1 it start from {1}, but for style 2 it start from {3}, the reason behind is for style 1 we don't have if (index == nums.length) condition when attempt to add 'tmp' list into result, but in style 2 we have if (index == nums.length) as condition, and because of this condition, only when the DFS go into deepest node (leave node), it will match the condition and able to add into result first, that's why result first add {3} instead of {1}.

One step forward, refer to here, above superficial difference is caused by traversal pattern difference between **the Backtracking style 1** and **style 2**. For **the Backtracking style 1 solution**, we are adding the list to the result directly in each recursion function(result.add(new ArrayList<>(tmp))) instead the one like in **Backtracking style 2 solution** which will add only when it reaches the last index. **The reason is that in the 1st solution we are choosing the next element to add, so we must choose one during for loop. So the case where we do not add next element we choose is right before the for loop. So we have to add to result each time to include this case.**

And if we check more on style 2, it a bit close to "Divide and Conquer" style:

1. base case (only when index == nums.length means current recursion combination done, add into result)
2. recursive into smallest problem (pick current element + not pick current element)
3. handle smallest problem (handle pick + handle not pick)

But strictly saying neither style 1 or style 2 is "Divide and Conquer" style of recursion, it is kind of "Traversal" style of recursion:

1. For "Divide and Conquer" always comes into return type for final combination instead of void return, but style 2 is void return for both pick / not pick current element branch
2. We pass a "storage" object as "result" to store all combinations through recursion process

Refer to

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