LeetCode 78. Subsets

1. Problem Title & Link

- **78. Subsets**
- 🔗 https://leetcode.com/problems/subsets/

2. Problem Statement (Short Summary)

Given an integer array nums containing **unique elements**, return *all possible subsets (the power set)*.

The solution set **must not contain duplicates**, and can be returned in **any order**.

3. Examples (Input → Output)

Input: nums = [1,2,3]

Output:

```
[
  [],
  [1],
  [2],
  [3],
  [1,2],
  [1,3],
  [2,3],
  [1,2,3]
]
```

4. Constraints

- 1 <= nums.length <= 10
- -10 <= nums[i] <= 10
- All elements are unique.

5. Thought Process (Step by Step)

This is **backtracking** in its purest form ❤️

We have a list of numbers, and for each one, we can either:

- **Include** it in the subset, or
- **Exclude** it.

This creates a *binary decision tree* of depth n.

Hence total subsets = 2^n 🌈

🧠 Step 1: Recursive Exploration

At each index:

1. Add the current element → explore.

2. Skip it → explore.

Whenever you reach the end of the array → add the built subset to the result.

🧩 Step 2: Ensure Copying

When adding a subset to results, always use a copy (path[:])
to avoid mutation issues due to Python list references.

6. Pseudocode

```
result = []

def backtrack(start, path):
    add copy(path) to result

    for i in range(start, len(nums)):
        path.append(nums[i])
        backtrack(i + 1, path)
        path.pop()

backtrack(0, [])
return result
```

7. Code Implementation

✅ **Python**

```python
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        res = []

        def backtrack(start, path):
            res.append(path[:])  # add the current subset

            for i in range(start, len(nums)):
                path.append(nums[i])
                backtrack(i + 1, path)
                path.pop()

        backtrack(0, [])
        return res
```

✅ **Java**

```java
class Solution {
    public List<List<Integer>> subsets(int[] nums) {
```

```
        List<List<Integer>> res = new ArrayList<>();
        backtrack(0, nums, new ArrayList<>(), res);
        return res;
    }

    private void backtrack(int start, int[] nums, List<Integer> path,
List<List<Integer>> res) {
        res.add(new ArrayList<>(path));
        for (int i = start; i < nums.length; i++) {
            path.add(nums[i]);
            backtrack(i + 1, nums, path, res);
            path.remove(path.size() - 1);
        }
    }
}
```

8. Time & Space Complexity

Metric	Complexity	Reason
Time	O(2^n * n)	Each subset takes O(n) to copy.
Space	O(n)	Recursion stack depth.

9. Dry Run (Step-by-Step Execution)

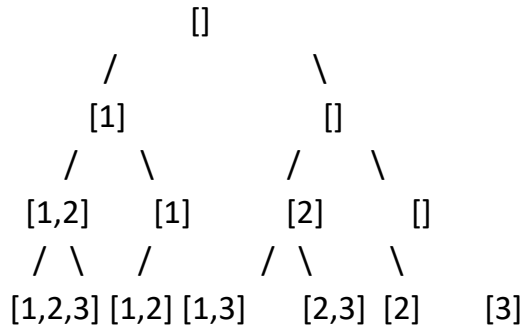
👉 Input: nums = [1, 2, 3]

Step	path	start	Added to res	Action
1	[]	0	✅ []	begin
2	[1]	1	✅ [1]	include 1
3	[1,2]	2	✅ [1,2]	include 2
4	[1,2,3]	3	✅ [1,2,3]	include 3
5	[1,3]	3	✅ [1,3]	skip 2
6	[2]	2	✅ [2]	skip 1
7	[2,3]	3	✅ [2,3]	include 3
8	[3]	3	✅ [3]	skip 1,2

✅ Output:

[
 [], [1], [2], [3],
 [1,2], [1,3], [2,3], [1,2,3]
]

10. Visual Tree Diagram (For Class ❤️)



Each node = one subset

Each branch = “Include” or “Exclude” decision 🌳

11. Concept Insight Table

Core Concept	Common Use Cases	Common Traps	Builds / Next Steps
Backtracking (Decision Tree Generation) — explore all combinations by recursive inclusion/ exclusion.	- Subsets / Power sets - Combinatorial generation - State-space exploration	- Forgetting to copy list before adding - Not popping after recursion - Using same `start` (causing duplicates)	🔷 Builds to **LC 90 (Subsets II)** (handle duplicates) 🔷 Connects to **LC 46 (Permutations)** 🔷 Forms base for **Combination Sum**, **N-Queens**, etc.

12. Common Mistakes / Edge Cases

- Forgetting to path.pop() → duplicates or incorrect results.
- Modifying path in place without copying.
- Assuming order matters — subsets are **order-independent**.

13. Variations / Follow-Ups

- **LC 90:** Subsets II (handles duplicates).
- **LC 46:** Permutations.
- **LC 77:** Combinations (limit subset size = k).
- **LC 39/40:** Combination Sum problems.