# **Leetcode 78 – Subsets**

## Problem Understanding

Given an integer array nums of **distinct integers**, return **all possible subsets (the power set)**.

* Each element can be either **included** or **excluded** in a subset.
* Output must include the **empty set** and the **original array** itself.

### Example:

Input: nums = [1, 2]

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

This is a classic **backtracking/recursion** problem, often visualized as a **binary tree of choices**.

## Optimized Java Solution (Backtracking)

class Solution {

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

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

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

return result;

}

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

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

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

current.add(nums[i]); // choose

backtrack(i + 1, nums, current, result); // explore

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

}

}

}

## Dry Run Using Table

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

We build subsets by choosing or not choosing each element.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Depth | Start | Path | Action | Result So Far |
| 0 | 0 | [] | Add to result | [[]] |
| 1 | 0 | [1] | Add to result | [[], [1]] |
| 2 | 1 | [1, 2] | Add to result | [[], [1], [1, 2]] |
| 3 | 2 | [1, 2, 3] | Add to result | [[], [1], [1, 2], [1, 2, 3]] |
| ← | 3 | [1, 2] | Backtrack | - |
| 3 | 2 | [1, 3] | Add to result | [[], [1], [1, 2], [1, 2, 3], [1, 3]] |
| ← | 2 | [1] | Backtrack | - |
| 2 | 1 | [2] | Add to result | [[], [1], [1, 2], [1, 2, 3], [1, 3], [2]] |
| 3 | 2 | [2, 3] | Add to result | [..., [2, 3]] |
| ← | 2 | [2] | Backtrack | - |
| 2 | 1 | [3] | Add to result | [..., [3]] |

✅ Final output:

## [

## [],

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

## [2], [2, 3],

## [3]

## ]

## Time / Space Complexity

|  |  |
| --- | --- |
| Metric | Value |
| Time | O(2ⁿ × n) |
| Space | O(2ⁿ × n) |

* There are **2ⁿ subsets**
* Each subset takes up to O(n) space (for copying)
* Call stack goes up to depth n

## Alternate Approaches

### 1. ****Bitmasking (Iterative)****

Generate integers from 0 to 2ⁿ - 1. Use binary bits to decide inclusion.

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

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

int n = nums.length;

for (int mask = 0; mask < (1 << n); mask++) {

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

for (int i = 0; i < n; i++) {

if ((mask & (1 << i)) != 0)

subset.add(nums[i]);

}

res.add(subset);

}

return res;

}

* Same time/space: O(2ⁿ × n)
* No recursion: useful when recursion isn't allowed

### 2. ****Recursive (No Backtracking)****

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

return helper(nums, 0);

}

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

if (index == nums.length) return Arrays.asList(new ArrayList<>());

List<List<Integer>> subsets = helper(nums, index + 1);

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

for (List<Integer> subset : subsets) {

List<Integer> newSubset = new ArrayList<>(subset);

newSubset.add(nums[index]);

result.add(newSubset);

}

return result;

}