

■ Subset Sums

Generate all possible sums from all possible subsets

● Recursion

■ MEDIUM

■ What You Will Learn

- The 'include or exclude' recursion pattern — the foundation of backtracking
- How 2^n subsets arise from n binary choices
- Building a result list through recursive exploration
- Sorting the output for clean results

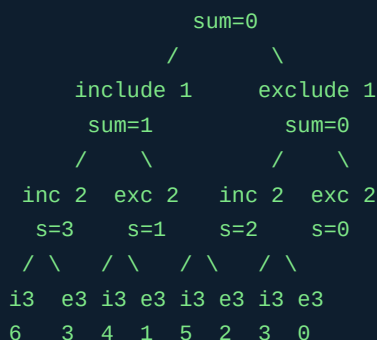
■ Intuition First

For every element, you make a binary choice: INCLUDE it in the current subset, or EXCLUDE it. If you have $[1,2,3]$, that's 8 possible subsets (2^3). The sum of $\{\}$ is 0, sum of $\{1\}=1$, $\{2\}=2$, $\{3\}=3$, $\{1,2\}=3$, $\{1,3\}=4$, $\{2,3\}=5$, $\{1,2,3\}=6$. We want all these sums.

■ Include/Exclude Pattern

At each index, you have exactly 2 choices. This creates a binary tree of decisions. Left branch = include $arr[i]$, Right branch = exclude $arr[i]$. Every path from root to leaf gives one subset sum. This is the MOST important recursion pattern in DSA!

■ Visual — Decision Tree for $[1,2,3]$



All 8 sums: $[0, 1, 2, 3, 3, 4, 5, 6]$

Sorted: $[0, 1, 2, 3, 3, 4, 5, 6]$

■■ How It Works — Step by Step

1. Start at index 0 with current sum = 0
2. At each index, make two recursive calls:
 - a. INCLUDE: add $arr[index]$ to sum, move to $index+1$
 - b. EXCLUDE: don't change sum, move to $index+1$

3. Base case: when `index == arr.length`, add current sum to result
4. After all calls return, sort the result list

■ Java Code

```
1  class Solution {
2      public List<Integer> subsetSums(int[] arr) {
3          List<Integer> result = new ArrayList<>();
4          solve(arr, 0, 0, result);
5          Collections.sort(result);
6          return result;
7      }
8
9      private void solve(int[] arr, int index, int currSum, List<Integer> result) {
10         // BASE CASE: processed all elements
11         if (index == arr.length) {
12             result.add(currSum); // This is one valid subset sum
13             return;
14         }
15
16         // CHOICE 1: Include arr[index] in current subset
17         solve(arr, index + 1, currSum + arr[index], result);
18
19         // CHOICE 2: Exclude arr[index] from current subset
20         solve(arr, index + 1, currSum, result);
21     }
22 }
```

■■ Time & Space Complexity

Type	Value	What It Means
Time	$O(2^n)$	2^n subsets, each explored exactly once
Space	$O(2^n)$	2^n sums stored in result list

■■ Common Mistakes

■■ Mistake

Trying to generate the subsets themselves before computing sums. Skip the subsets entirely — just carry the running sum as a parameter.

■■ Both Branches

You MUST make BOTH recursive calls (include AND exclude). Missing one branch means missing half the subsets.

■ Pro Tips

■ This IS Backtracking

The include/exclude pattern IS the backtracking template. Once you internalize it here, problems like Subsets, Combination Sum, Permutations all become variations of the same idea.

■ Practice Problems

#	Problem	Diff
1	Subset Sums (GFG)	Medium
2	Subsets (LeetCode 78)	Medium
3	Sum of All Subsets	Easy
4	Target Sum (LeetCode 494)	Medium

■ Quick Revision

✓	At each index: 2 choices — include ($\text{currSum} + \text{arr}[i]$) or exclude (currSum)
✓	Base case: $\text{index} == n \rightarrow$ add currSum to result
✓	Total subsets = 2^n , total sums = 2^n
✓	Sort result at the end for ordered output
✓	Time $O(2^n)$, Space $O(2^n)$