Subsets Pattern

The **Subsets** (or "Power Set") pattern is used when you need to generate **all possible combinations or subsets** from a set of items (array, string, etc.).
This pattern is the foundation for problems about **combinations, permutations, and generating all possible choices** (including "pick or not pick" scenarios).

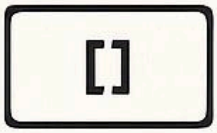
SUBSETS PATTERN

GENERATE ALL SUBSETS (POWER SET)

Input array: [1, 2, 3]

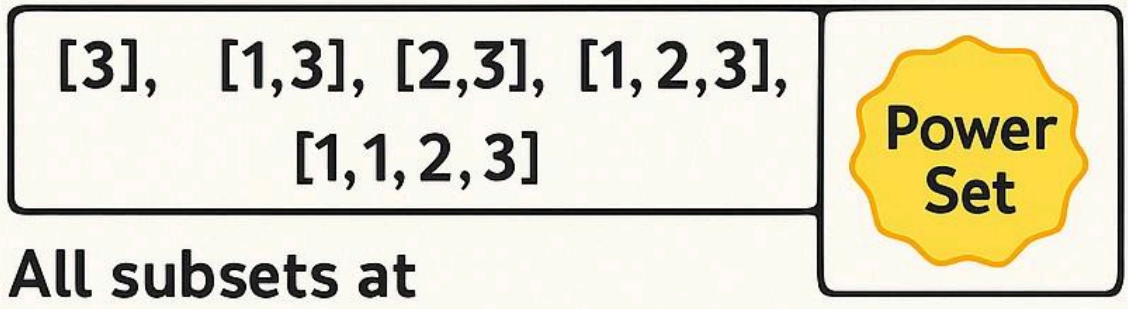
Task: List all possible subsets (including empty set)

1. Start with the empty set.



2. For each element, add it to all existing subsets to create new ones.

4. All possible su



5. All subsets at

Bonus Insight:

This pattern is the foundation for problems about combinations, permutations, and all possible choices!

Classic Problem: Generate All Subsets (Power Set)

Problem Statement:
Given an array of `n` unique numbers, print **all possible subsets**.

Example:
Input: `[1, 2, 3]`
Output:

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

Explanation

There are two popular approaches:

1. **Iterative (using queues or dynamic lists)**
2. **Recursive (using backtracking)**

Let's do both (the iterative is the true "coding pattern," but recursion is classic in C).

1. Iterative Approach (Using Array of Subsets)

Key Idea:
Start with the empty set.
For each number, add it to all existing subsets to create new subsets.

C Code Example: Iterative Subsets

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_SETS 100
#define MAX_SET_SIZE 10

// Function to print a subset
void printSubset(int subset[], int size) {
    printf("[");
    for (int i = 0; i < size; i++) {
        printf("%d", subset[i]);
        if (i < size - 1) printf(", ");
    }
```

```
    printf("]\n");
}

void subsets(int arr[], int n) {
    int result[MAX_SETS][MAX_SET_SIZE]; // Stores all subsets
    int sizes[MAX_SETS]; // Stores sizes of subsets
    int count = 1; // Number of subsets found

    // Start with empty set
    sizes[0] = 0;

    for (int i = 0; i < n; i++) {
        int currNum = arr[i];
        int currCount = count;

        // For every existing subset, add currNum to make a new subset
        for (int j = 0; j < currCount; j++) {
            // Copy existing subset
            for (int k = 0; k < sizes[j]; k++) {
                result[count][k] = result[j][k];
            }
            // Add current number
            result[count][sizes[j]] = currNum;
            sizes[count] = sizes[j] + 1;
            count++;
        }
    }

    // Print all subsets
    for (int i = 0; i < count; i++) {
        printSubset(result[i], sizes[i]);
    }
}

int main() {
    int arr[] = {1, 2, 3};
    int n = sizeof(arr) / sizeof(arr[0]);
    subsets(arr, n);
    return 0;
}
```

How does this work?

- Start with only `[]`.
- For each number, for every existing subset, make a **new subset that includes this number**.
- Add all those new subsets to your collection.

For `[1, 2, 3]`:

- Start: `[ ]`
- Add 1: `[ ]`, `[1]`
- Add 2: `[ ]`, `[1]`, `[2]`, `[1,2]`
- Add 3: `[ ]`, `[1]`, `[2]`, `[1,2]`, `[3]`, `[1,3]`, `[2,3]`, `[1,2,3]`

2. Recursive Approach (Backtracking)

Key Idea:
For each number:

- Include it in the subset and move forward.
- Or skip it and move forward.

C Code Example: Recursive Subsets

```
#include <stdio.h>

void printSubset(int subset[], int size) {
    printf("[");
    for (int i = 0; i < size; i++) {
        printf("%d", subset[i]);
        if (i < size - 1) printf(", ");
    }
    printf("]\n");
}

// Recursive function to generate all subsets
void generateSubsets(int arr[], int n, int index, int subset[], int subsetSize) {
    if (index == n) {
        printSubset(subset, subsetSize);
        return;
    }

    // 1. Include arr[index]
    subset[subsetSize] = arr[index];
    generateSubsets(arr, n, index + 1, subset, subsetSize + 1);

    // 2. Exclude arr[index]
    generateSubsets(arr, n, index + 1, subset, subsetSize);
}

int main() {
    int arr[] = {1, 2, 3};
    int n = sizeof(arr) / sizeof(arr[0]);
    int subset[10];
    generateSubsets(arr, n, 0, subset, 0);
    return 0;
}
```

How does this work?

- At every step, you have **two choices**: pick the current element, or skip it.
- The recursion tree explores all possible combinations.

Where is the Subsets Pattern Useful?

- Generating power sets
- Combination/Permutation problems (pick or skip)
- Partitioning sets (like subset-sum, equal partition)

- Word break, string abbreviations, etc.

Practice Challenge

- Try generating **all subsets of a string** (not an array).
- Try to generate all subsets **with only a specific length** (e.g., all pairs, all triplets).