# Dynamic Programming: Top-Down Memoization — Coding Interview Notes (Light Theme)

## General Pattern Template

```
def fn(arr):
    def dp(STATE):
        if BASE_CASE:
            return 0

        if STATE in memo:
            return memo[STATE]

        ans = RECURRENCE_RELATION(STATE)
        memo[STATE] = ans
        return ans

    memo = {}
    return dp(STATE_FOR_WHOLE_INPUT)
```

**Concept:**
**Top-Down Dynamic Programming** (also known as **memoization**) solves problems recursively while caching intermediate results to avoid recomputation. Each subproblem's result is stored in a memo dictionary keyed by its state.

This approach combines the clarity of recursion with the efficiency of dynamic programming.

**Time Complexity:** O(#states × cost_per_state)
**Space Complexity:** O(#states + recursion_depth)

## Key Ideas

1. Identify the problem's **state** — the minimal variables defining a subproblem.
2. Define the **recurrence relation** that expresses the state in terms of smaller states.
3. Use a **memo** dictionary to store already computed subproblem results.
4. The recursion should end with **base cases** (directly computable results).
5. Memoization avoids repeated work by caching results, turning exponential recursion into polynomial time.

## Example 1: Fibonacci Numbers

**Goal:** Compute nth Fibonacci number efficiently.
**Approach:** Recursive formula F(n) = F(n–1) + F(n–2), memoized to avoid recomputation.

```
def fib(n):
    memo = {}
    def dp(i):
        if i <= 1:
            return i
        if i in memo:
            return memo[i]
        memo[i] = dp(i-1) + dp(i-2)
        return memo[i]
    return dp(n)

# Example
print(fib(10))  # Output: 55
```

## Example 2: Coin Change (Minimum Coins)

**Goal:** Find the minimum number of coins to make amount target from given denominations.
**Approach:** Recurse over all coin choices, memoizing subresults.

```
def coin_change(coins, target):
    memo = {}
    def dp(rem):
        if rem == 0:
            return 0
        if rem < 0:
            return float('inf')
        if rem in memo:
            return memo[rem]
        memo[rem] = min(dp(rem - c) + 1 for c in coins)
        return memo[rem]

    ans = dp(target)
    return ans if ans != float('inf') else -1

# Example
print(coin_change([1,2,5], 11))  # Output: 3 (5+5+1)
```

## Example 3: Longest Increasing Subsequence

**Goal:** Return the length of the longest increasing subsequence in nums.
**Approach:** Top-down recursion with memoization over indices.

```
def length_of_LIS(nums):
    memo = {}
    def dp(i, prev):
        if i == len(nums):
            return 0
        key = (i, prev)
        if key in memo:
            return memo[key]
```

```
        take = 0
        if prev == -1 or nums[i] > nums[prev]:
            take = 1 + dp(i + 1, i)
        skip = dp(i + 1, prev)
        memo[key] = max(take, skip)
        return memo[key]

    return dp(0, -1)

# Example
print(length_of_LIS([10,9,2,5,3,7,101,18]))  # Output: 4
```

## Summary Table

ProblemStateRecurrenceResultComplexity Fibonacciidp(i)=dp(i-1)+dp(i-2)nth FibonacciO(n) Coin Changeremaining amountmin(dp(rem-c)+1)min #coinsO(n×target) LIS(i,prev)max(take, skip)longest lengthO(n$^2$)