Greedy vs. Dynamic Programming: The Ultimate Guide

*Master the Art of Choosing the Right Approach*

# Core Concepts Demystified

## 1. Greedy Algorithms: The "Live in the Moment" Strategy

**What It Is**:

* **Make the best local choice** at every step, assuming it will lead to the best global outcome.
* *Analogy*: Picking the juiciest apple from the basket first, hoping it gives the best overall snack.

**Key Properties**:

* **Greedy Choice Property**: A local optimal choice is part of the global optimal solution.
* **Optimal Substructure**: The problem can be broken into smaller subproblems with optimal solutions.

**When to Use**:

* Problems where **past decisions don’t restrict future choices** (e.g., scheduling, shortest path).
* *Example*: **Fractional Knapsack** (take fractions of items) → Always pick the highest value/weight ratio.

**Pitfalls**:

* Greedy fails if the problem requires **revisiting past choices** (e.g., 0/1 Knapsack).

## 2. Dynamic Programming: The "Strategic Planner"

**What It Is**:

* **Solve subproblems once, reuse their solutions** to build up to the final answer.
* *Analogy*: Planning a road trip by precomputing the best route between every pair of cities.

**Key Properties**:

* **Overlapping Subproblems**: The same subproblem is solved multiple times.
* **Optimal Substructure**: Optimal solution can be constructed from optimal solutions of subproblems.

**When to Use**:

* Problems with **interdependent decisions** (e.g., sequence alignment, resource allocation).
* *Example*: **0/1 Knapsack** → Track all combinations of items and weights.

**Pitfalls**:

* Overkill for simple problems; high space/time complexity if not optimized.

# Greedy vs. DP: Head-to-Head

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Greedy** | **Dynamic Programming** |
| **Decision Style** | "What’s best now?" | "What if I try all options?" |
| **Time Complexity** | Often *O(n log n)* (sorting + iteration) | Usually *O(n²)* or *O(nW)* |
| **Space** | *O(1)* or *O(n)* (for sorting) | *O(nW)* (knapsack) or *O(n)* (LIS) |
| **Proof Required** | Must **prove correctness** (no guarantees!) | Correct by construction (recurrence) |
| **Use Case** | Scheduling, Shortest Path (Dijkstra) | Knapsack, LCS, Grid Path Counting |

# Problem Spotting Guide

### **Greedy Flags** 🚩

1. Keywords: *"Maximize count," "Shortest time," "Earliest deadline."*
2. **Sorted Inputs**: Problems where sorting unlocks a clear optimal path (e.g., activity selection).
3. **No Backtracking Needed**: Once a choice is made, it’s final (e.g., Huffman coding).

## **DP Flags** 🚩

1. Keywords: *"Number of ways," "Minimum/Maximum cost with constraints," "Subsequence."*
2. **State Dependency**: Decisions affect future states (e.g., 0/1 Knapsack weight limits).
3. **Recursive Relationships**: Fibonacci, grid paths with obstacles.

# Classic LeetCode Problems

## 1. Greedy: [435. Non-Overlapping Intervals](https://leetcode.com/problems/non-overlapping-intervals/)

**Problem**: Given intervals, remove the minimum number to make the rest non-overlapping.  
**Intuition**: Sort by **end time** → Greedily pick the interval that ends earliest to maximize remaining space.

**Why Greedy Works**: Overlaps are resolved by favoring intervals that "free up" the timeline fastest.

## 2. DP: [322. Coin Change](https://leetcode.com/problems/coin-change/)

**Problem**: Find the minimum coins needed to make an amount (coins can be reused).  
**Intuition**:

* **Greedy Trap**: Works only for canonical systems (e.g., US coins). Fails for coins = [1, 3, 4], amount = 6 (greedy picks 4+1+1=3 coins, DP picks 3+3=2).
* **DP Approach**: Track dp[i] = min coins for amount i. Update for each coin.

## 3. Hybrid (Greedy + DP): [300. Longest Increasing Subsequence](https://leetcode.com/problems/longest-increasing-subsequence/)

**Problem**: Find the length of the longest strictly increasing subsequence.  
**Intuition**:

* **DP (O(n²))**: dp[i] = LIS ending at i. Compare all previous elements.
* **Greedy-like (O(n log n))**: Maintain a list of smallest tail elements for increasing sequences of length i+1. Use binary search to update.

**Example**:  
nums = [10, 9, 2, 5, 3, 7, 101] → LIS is [2, 5, 7, 101] (length 4).

## When to Choose Greedy vs. DP: A Decision Tree

1. **Can I make a choice that’s *provably* optimal at this step?**
   1. **Yes** → Greedy (e.g., activity selection).
   2. **No** → Move to DP.
2. **Do I need to track multiple states or revisit decisions?**
   1. **Yes** → DP (e.g., 0/1 Knapsack).
   2. **No** → Greedy (e.g. Fractional Knapsack)

# Pro Tips for Interviews

1. **Greedy**:
   1. **Always ask**: "Can I prove this works for all cases?"
   2. **Sort first**: Many greedy solutions start with sorting (e.g., interval problems).
2. **DP**:
   1. **Start with recursion**: Write a brute-force recursive solution, then memoize.
   2. **Visualize the table**: Draw the DP table for small examples to spot patterns.
3. **Hybrid Problems**:
   1. Look for optimizations (e.g., LIS with binary search).

# Real-World Applications

* **Greedy**:
  + **Dijkstra’s Algorithm**: Shortest path in graphs with non-negative weights.
  + **Huffman Coding**: Data compression by prioritizing frequent characters.
* **DP**:
  + **Autocorrect**: Edit distance to find the closest valid word.
  + **Stock Trading**: Maximize profit with buy/sell constraints.

# Practice Makes Perfect

**Recommended Problems**:

1. **Greedy**:
   1. [763. Partition Labels](https://leetcode.com/problems/partition-labels/): Merge overlapping intervals greedily.
   2. [121. Best Time to Buy/Sell Stock](https://leetcode.com/problems/best-time-to-buy-and-sell-stock/): Track min price.
2. **DP**:
   1. [1143. Longest Common Subsequence](https://leetcode.com/problems/longest-common-subsequence/): Classic 2D DP.
   2. [70. Climbing Stairs](https://leetcode.com/problems/climbing-stairs/): Fibonacci-style recurrence.

**Final Takeaway**:

* **Greedy** = Speed + Simplicity (but needs proof!).
* **DP** = Flexibility + Power (but needs careful state design!).
* **Hybrids** = Best of both worlds (e.g., LIS).