Topic Introduction

Today’s spotlight is on **Dynamic Programming (DP) for Optimal Selection with Constraints** — a powerful technique that helps you make the best choices when each decision limits your options for the next. Whether you’re choosing which houses to rob, which jobs to schedule, or which non-overlapping intervals to pick, DP shines when you want the maximum (or minimum) result while following certain rules.

What is Dynamic Programming for Selection?

At its core, DP breaks complex problems into overlapping subproblems, solves each just once, and stores the results for reuse. In selection problems, you’re often deciding: “Do I pick this item, or skip it for possibly better options?” The trick is to define a recurrence: the best choice at position `i` depends on previous optimal choices, factoring in any constraints (like "don’t pick adjacent items").

When and Why is This Useful?

Interviewers love these problems because they test your ability to:

- Break down a problem into subproblems
- Identify overlapping subproblems
- Write efficient, bug-resistant code

Simple Example: Picking Non-Adjacent Numbers for Maximum Sum

Given an array `[2, 7, 9, 3, 1]`, pick numbers (not adjacent) to maximize the sum.

- If you pick `2`, you must skip `7`.
- If you skip `2`, you can pick `7` or whatever comes next.

This is the classic “maximum sum of non-adjacent elements” problem, and it’s where today’s trio of problems all begin.

Let's look at three classic DP problems that explore this pattern in different settings:

- House Robber
- House Robber II
- House Robber III

Why these three?
All use DP to help you select the most loot, but with different constraints: a straight row of houses, a circular neighborhood, and a neighborhood represented as a binary tree. Each one asks: "How do I pick houses to rob for the most money, without ever robbing two adjacent houses?" The principle is the same, but the structure and constraints evolve!

Problem 1: House Robber

House Robber - LeetCode

Problem in Simple Terms:

You’re a robber, looking at a row of houses. Each house has a certain amount of cash. You can’t rob two adjacent houses (security systems go off). What’s the maximum money you can rob?

Example:

```
Input:  [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (2), skip 7, rob 9, skip 3, rob 1. 2 + 9 + 1 = 12.
```

Thought Process:

- At each house, ask:
    - If I rob this house, I must skip the previous one.
    - If I skip this house, I can take the best up to the previous house.
- DP lets us define:
    - `dp[i] = max(dp[i-1], dp[i-2] + nums[i])`
- Try pen and paper: For small arrays, write each choice and see how the pattern emerges.

Try This Test Case:

```
Input: [1, 2, 3, 1]
Expected Output: 4
```

(Rob 1 and 3 → 1 + 3 = 4)

Brute Force:

Try every combination: O(2^n). Not efficient!

Optimal Approach:

- Use DP.
- For each house, store the max loot up to that house.
- Only need to remember the last two results at any time (space optimization).

Python Solution

```python
def rob(nums):
    # Handle base cases
    if not nums:
        return 0
    if len(nums) == 1:
        return nums[0]

    # Initialize two variables to store results for the last two houses
    prev2 = 0  # Max loot up to house i-2
    prev1 = 0  # Max loot up to house i-1

    for amount in nums:
        # Decide: rob current (prev2 + amount) or skip (prev1)
        current = max(prev1, prev2 + amount)
        prev2 = prev1
        prev1 = current

    return prev1
```

Time Complexity: O(n) (one pass through the array)

Space Complexity: O(1) (just two variables, not an array)

Explanation and Step-By-Step Trace:

- `prev2` and `prev1` hold the best results so far.
- For each house, calculate:
    - If I rob it: `prev2 + amount`
    - If I skip it: `prev1`
    - Take the max.
- Shift the results forward for the next iteration.

Trace for `[2,7,9,3,1]`:

House	prev2	prev1	amount	current (max)
-	0	0	-	-
2	0	0	2	max(0, 0+2) = 2
7	0	2	7	max(2, 0+7) = 7
9	2	7	9	max(7, 2+9) = 11
3	7	11	3	max(11, 7+3) = 11
1	11	11	1	max(11, 11+1) = 12

Result: 12

Try this yourself:
Input: `[2, 1, 1, 2]`
What's the output?

Self-Attempt Encouragement:
Try coding or writing out the solution for a couple of arrays before peeking at the code!

Reflection:
Did you know? This same pattern solves the "maximum sum of non-adjacent numbers" in any array. Try adapting it!

Problem 2: House Robber II

House Robber II - LeetCode

How is it Different?

Now the houses are **in a circle**. The first and last houses are neighbors. You can't rob both!

Example:

```
Input: [2, 3, 2]
Output: 3
Explanation: Rob house 2 (3), since robbing 2 and 2 would mean robbing first and last
(not allowed).
```

Brute Force:

Try every valid combination while skipping either the first or last house. Still exponential.

Optimal Approach:

- Realization: You can't rob both first and last.
- So, split the problem:
    - Case 1: Rob houses 0 to n-2 (skip last)
    - Case 2: Rob houses 1 to n-1 (skip first)
    - Take the maximum of the two.

Step-by-Step Plan:

- If only one house, just take it.
- Otherwise, run the standard House Robber DP twice:
    - On `nums[0 : n-1]`
    - On `nums[1 : n]`
- Return the max of the two.

Pseudocode:

```
function rob(nums):
    if length(nums) == 1:
        return nums[0]
    return max(
        rob_linear(nums[0 : n-1]),
        rob_linear(nums[1 : n])
    )

function rob_linear(nums):
    prev2, prev1 = 0, 0
    for amount in nums:
        current = max(prev1, prev2 + amount)
        prev2 = prev1
        prev1 = current
    return prev1
```

Example Trace:

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

- Run DP on `[1,2,3]`: result is 4 (rob 1 and 3)
- Run DP on `[2,3,1]`: result is 3 (rob 2 and 1)
- Final answer: max(4, 3) = 4

Another Test Case:

Input: `[2, 7, 9, 3, 1]`

- DP on `[2,7,9,3]`: max loot is 11
- DP on `[7,9,3,1]`: max loot is 12
- Output: 12

Complexity:

- Time: O(n) (each DP run is O(n), run twice)
- Space: O(1)

Self-Attempt:
Try implementing this! Reuse your code from Problem 1 for the `rob_linear` helper.

Problem 3: House Robber III

House Robber III - LeetCode

What’s the Twist?

Now, houses are arranged as a **binary tree**. You can’t rob two directly-linked houses (parent-child). What’s the most you can rob?

Example:

```
    3
   / \
  2   3
   \   \
    3   1
```

You can rob 3 (root), and 3 (left child’s right), and 1 (right child’s right): 3 + 3 + 1 = 7

Brute Force:

Try every combination of robbing/skipping each node with the constraint. Time: O(2^n)

Optimal Approach:

- At each node, decide:
    - Rob it: Then you can't rob its children, but can rob its grandchildren.
    - Skip it: You can rob its children.
- Use recursion with memoization: For each node, return two values:
    - Max if you rob this node
    - Max if you skip this node

Pseudocode:

```
function rob_tree(node):
    if node is None:
        return (0, 0)  # (rob, skip)
    left = rob_tree(node.left)
    right = rob_tree(node.right)
    # If we rob this node, can't rob children
    rob = node.val + left[1] + right[1]
    # If we skip this node, can take max of rob/skip for children
    skip = max(left) + max(right)
    return (rob, skip)

function rob(root):
    return max(rob_tree(root))
```

Step-by-Step Example:

For the tree above:

- At leaf nodes: return (val, 0)
- At each parent, compute both choices based on children.

Try This Test Case:

```
    4
   / \
  1   5
 /   / \
2   1   3
```

What’s the max loot? (Hint: Try both robbing and skipping each node.)

Complexity:

- Time: O(n), each node is visited once.
- Space: O(h), for recursion stack (h = height of tree).

Your Turn:
Try implementing this DP. Can you find a way to do it iteratively? Or test with larger trees?

Summary and Next Steps

Why These Problems?

You just explored three levels of the same DP selection challenge:

- Linear (array)
- Circular (array, but with first/last linked)

- Tree (arbitrary structure)

Key Patterns to Remember:

- When you can't pick adjacent elements, DP is your friend.
- Break the problem into "pick or skip" at each step.
- For trees, track two values at each node (rob, skip).

Common Traps:

- Forgetting the edge case for one house or empty input.
- Not handling the circular condition in House Robber II.
- In House Robber III, mixing up which nodes you can rob.

Action List

- Try coding all three problems yourself, even if you saw the full code above!
- For House Robber III, experiment with memoization or iterative tree traversal.
- Explore similar problems: e.g., "Maximum Sum of Non-Adjacent Numbers," "Paint House," "Maximum Weight Independent Set."
- Compare your code to others for edge cases and style tips.
- If you get stuck, walk through with pen and paper — it’s a DP superpower!
- Keep practicing. Each variation strengthens your DP intuition!

Happy robbing (of problems, not houses)!