Topic Introduction

Today we're diving into one of the most classic interview themes: **state machine dynamic programming for stock trading problems**. If you've ever seen "Best Time to Buy and Sell Stock" on LeetCode, you know the setup: prices go up and down, and your goal is to maximize profit under various constraints.

But here's the twist. Each variation of this problem tweaks the rules—sometimes you can only buy and sell once, sometimes as many times as you like, sometimes you need to take a breather before the next trade. Underneath, these problems are all about managing "states": Are you holding a stock, or not? What actions are allowed next? This is why **state machine DP** is the key.

What is State Machine Dynamic Programming?

State machine DP is an approach where you model a problem as a set of "states" and transitions between them. Each state captures information about what decisions have been made so far. For stock trading, states might include:

- Do you currently hold a stock?
- Have you just sold?
- Are you in a cooldown period?

At each day, you make a decision: buy, sell, or do nothing, transitioning between states to maximize profit.

Why is this useful in interviews?

- These problems are everywhere, in stock trading, scheduling, even gaming.
- They test your ability to model real-world scenarios as code.
- They build your DP intuition for more complex problems.

Example (not one of our target problems):
Suppose you have prices: `[3, 2, 6, 5, 0, 3]` and can only make one transaction (buy once, sell once). The best way is to buy at 2 and sell at 6 for max profit of 4. You'd track the minimum price seen so far and see if selling today beats your best profit so far.

Let’s dive into three classic variations:

- **Best Time to Buy and Sell Stock**: One transaction allowed.
- **Best Time to Buy and Sell Stock II**: Unlimited transactions.
- **Best Time to Buy and Sell Stock with Cooldown**: Unlimited transactions, but after selling, you must wait one day before buying again.

Why are these grouped together?
They all ask you to maximize profit given a list of prices, but each with a twist. The constraint changes force you to tweak your state machine and DP logic, making them great practice for mastering this pattern.

Problem 1: Best Time to Buy and Sell Stock

LeetCode 121

Problem Statement (in plain English):

Given an array `prices` where `prices[i]` is the price of a stock on day `i`, find the maximum profit you can make by buying and selling exactly once (buy before you sell). If you can't make any profit, return 0.

Example:

Input: `[7, 1, 5, 3, 6, 4]`

Output: `5`

(You buy on day 1 at price 1 and sell on day 4 at price 6.)

Thought Process:

This is the simplest version: one buy, one sell. You want the lowest price to buy, and the highest possible price *after* that to sell. At each day, you can ask: If I sold today, what would my profit be, given the lowest price I've seen so far?

Try with pen and paper:

Given `[7, 6, 4, 3, 1]`, what should the answer be?

Brute-force approach:

Try every pair of buy/sell days (nested loops).

- Time complexity: O(N^2)
- Not efficient for large arrays.

Optimal approach:

Track the minimum price so far as you iterate. At each day, compute profit if you sold today and update max profit.

Step-by-step logic:

- Set `min_price` to infinity, `max_profit` to 0.
- For each price:
    - If price < min_price, update min_price.
    - Else, calculate `profit = price - min_price`. If profit > max_profit, update max_profit.

Python Solution

```python
def maxProfit(prices):
    min_price = float('inf')  # Start with a very high min price
    max_profit = 0

    for price in prices:
        if price < min_price:
            min_price = price  # Found a new minimum
        else:
            profit = price - min_price  # Potential profit
            if profit > max_profit:
                max_profit = profit  # Update max profit if better

    return max_profit
```

- **Time complexity:** O(N) (one pass)
- **Space complexity:** O(1)

Explanation:

- `min_price` keeps track of the lowest price seen so far.
- `max_profit` is updated if selling today yields a better profit.
- We only need one pass since, at each step, we keep all info needed.

Trace with `[7, 1, 5, 3, 6, 4]`:

Day	Price	min_price	profit	max_profit
0	7	7	0	0
1	1	1	0	0
2	5	1	4	4
3	3	1	2	4
4	6	1	5	5
5	4	1	3	5

Final answer: `5`

Try this test case yourself:

Input: `[2, 4, 1]`

Expected Output: `2`

Take a moment to solve this on your own before jumping into the solution!

Problem 2: Best Time to Buy and Sell Stock II

LeetCode 122

What's new here?

You can make as many transactions as you want, but you must sell before you can buy again. The goal is still to maximize profit.

Example:

Input: `[7, 1, 5, 3, 6, 4]`

Output: `7`

(You buy at 1, sell at 5, buy at 3, sell at 6.)

How is this different from Problem 1?

Multiple transactions allowed! Now, instead of looking for the largest single rise, we want to capture *every* upward movement.

Brute-force approach:

Try every combination of buy/sell pairs (exponential).

- Not practical.

Optimal approach:

Notice that any time the price goes up from one day to the next, you can profit by "buying yesterday and selling today." So, sum up all increases.

Step-by-step logic:

- Initialize `profit = 0`.
- For each day from 1 to n-1:
    - If `prices[i] > prices[i-1]`, add `prices[i] - prices[i-1]` to profit.

Pseudocode:

```
profit = 0
for i from 1 to n-1:
    if prices[i] > prices[i-1]:
        profit += prices[i] - prices[i-1]
return profit
```

Example trace with `[1, 2, 3, 4, 5]`:

- Day 1: 2 > 1, profit += 1
- Day 2: 3 > 2, profit += 1
- Day 3: 4 > 3, profit += 1
- Day 4: 5 > 4, profit += 1

Total profit: 4

Try this test case:

Input: `[7, 6, 4, 3, 1]`

Expected Output: `0`

(No increases, so no profit.)

Time complexity: O(N)

Space complexity: O(1)

Trace with `[7, 1, 5, 3, 6, 4]`:

- 1 < 7: skip
- 5 > 1: profit += 4 (profit = 4)
- 3 < 5: skip
- 6 > 3: profit += 3 (profit = 7)
- 4 < 6: skip

Final answer: `7`

Try to write this code yourself, or dry-run with `[6, 1, 3, 2, 4, 7]` (Expected: 7).

Problem 3: Best Time to Buy and Sell Stock with Cooldown

LeetCode 309

What's the twist?

You can buy and sell as many times as you want, but after you sell, you must wait one day before buying again (the "cooldown" day).

Example:

Input: `[1, 2, 3, 0, 2]`

Output: `3`

(Buy on day 0 at 1, sell on day 2 at 3, cooldown day 3, buy on day 3 at 0, sell on day 4 at 2.)

How is this harder?

Now you can't just sum all increases—after you sell, you must sit out for a day. This requires DP with explicit state tracking.

State Machine:

At each day, you can be in one of three states:

- **Hold**: You own a stock.
- **Not hold, can buy**: You don't own a stock and are not in cooldown.
- **Cooldown**: You just sold and can't buy today.

Step-by-step logic:

- Let
    - `hold[i]` = max profit on day i if you hold a stock
    - `sold[i]` = max profit on day i if you just sold
    - `rest[i]` = max profit on day i if you do nothing (and are not holding)
- The recurrence:
    - `hold[i] = max(hold[i-1], rest[i-1] - prices[i])`
    - `sold[i] = hold[i-1] + prices[i]`
    - `rest[i] = max(rest[i-1], sold[i-1])`
- Final answer is max of `sold[n-1]` and `rest[n-1]`.

Pseudocode:

```
n = len(prices)
hold = [0] * n
sold = [0] * n
rest = [0] * n

hold[0] = -prices[0]
sold[0] = 0
rest[0] = 0

for i from 1 to n-1:
    hold[i] = max(hold[i-1], rest[i-1] - prices[i])
    sold[i] = hold[i-1] + prices[i]
    rest[i] = max(rest[i-1], sold[i-1])

return max(sold[n-1], rest[n-1])
```

Try this test case:

Input: `[2, 1, 4]`

What should the answer be?

Time complexity: O(N)

Space complexity: O(N) (can be reduced to O(1) with variable optimization)

Hint:

Try to implement this yourself! Notice how each state only depends on the previous day.

If you want a challenge, try reducing space to O(1) by using variables instead of arrays.

Summary and Next Steps

These three problems are grouped because they all use **state machine dynamic programming** to handle stock trading with different constraints. As you move from one to the next, you see how adding rules (unlimited trades, cooldowns) changes the structure of the DP and the states you need to track.

Key patterns and insights:

- Always identify the possible "states" at each step.
- Think about how today's actions depend on yesterday's state.
- For variations, carefully update your transitions and base cases.

Common traps:

- Forgetting that you can't buy and sell on the same day.
- Not updating your state properly after a cooldown.
- Overcomplicating when a greedy approach is enough (as in Problem 2).

Action List:

- Try to solve all three problems on your own—even the one with code provided here.
- For Problem 2 and 3, try to write both the straightforward and optimized (O(1) space) versions.
- Look for other problems that use state machines or DP with "actions" and "states" (e.g., house robber, coin change).
- Review your solutions for edge cases and compare with others' code.
- Practice explaining your logic out loud—the more you can verbalize these patterns, the better you'll do in interviews!

Happy coding!