2 Optimal stock market strategy

When evaluating stock market trading strategies, it is useful to determine the maximum possible profit that can be made by trading a certain stock. Write an algorithm that, given the daily price of a stock, computes the maximum profit that can be made by buying and selling that stock. Assume that you are allowed to own no more than 1 share at any time, and that you have an unlimited budget.

Example 1: The stock price over several days is `[2, 5, 1]`. The best strategy is to buy a share on the first day for price 2, then sell it on the second day for price 5, obtaining a profit of 3.

Example 2: The stock price over several days is `[2, 5, 1, 3]`. The best strategy is to buy a share on the first day for price 2, then sell it on the second day for price 5, obtaining a profit of 3; then buy it again on the third day for price 1, and sell it on the fourth day for price 3, obtaining an overall profit of 5.

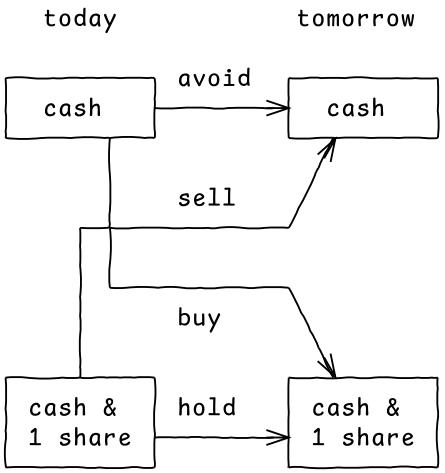
Solution 1: dynamic programming, top-down, $O(n)$ time

The first idea that comes to mind while approaching this problem is using a state machine. This is because on any day, our state can be described by:

- whether we own the share or not;
- the amount of money we have.

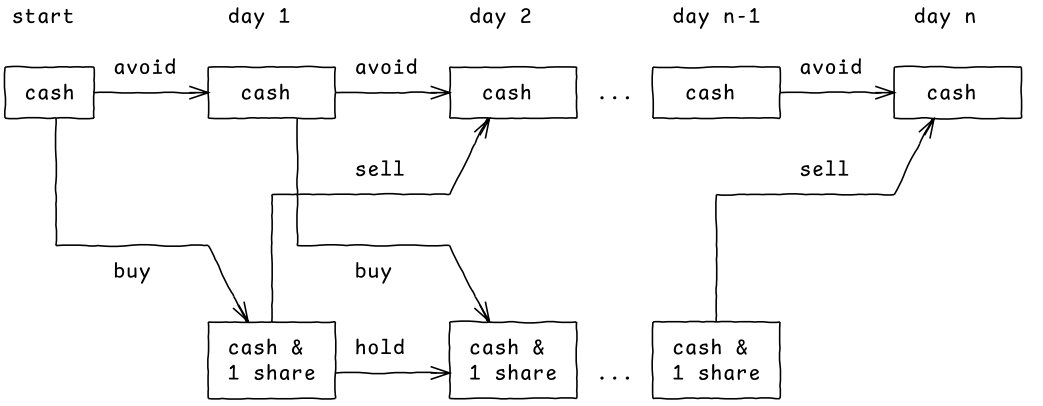
Between the states of consecutive days, we have only four possible transitions:

- If at the end of the previous day we did not own the share:
  - buying the stock, so we now own it, but we have less money;
  - avoiding the stock, so we keep our money unchanged;
- If at the end of the previous day we owned the share:
  - selling the stock, so we no longer own it, and have more money;
  - holding the stock, so we keep both the stock and our money unchanged.



Knowing this, we can model the entire problem using a state machine. In our initial state,

we have some amount of cash and no shares. In the final state, we have some other amount of cash (ideally higher), and no shares. In between, we have state transitions:



Solving the original problem can be reduced to finding a chain of transitions through this state machine, that yields the maximum profit.

Notice how our state during any day only depends on the state from the previous day. This is excellent: we can express our problem using a simple recurrence relation, just as we did for the Fibonacci sequence problem.

The structure of the solution using a recursive algorithm looks like this:

```
def max_profit(daily_price):
    def get_best_profit(day, have_stock):
        """
        Returns the best profit that can be obtained by the end of the day.
        At the end of the day:
        * if have_stock is True, the trader must own the stock;
        * if have_stock is False, the trader must not own the stock.
        """
        # TODO ...
    # Final state: end of last day, no shares owned.
    last_day = len(daily_price) - 1
    no_stock = False
    return get_best_profit(last_day, no_stock)
```

Note that we defined a helper function `get_best_profit` which takes as parameters the identifiers of a state: the day number and whether we own the stock or not at the end of the day. We use `get_best_profit` to compute the profit for a specific state in the state machine.

Let's now implement the helper using a recurrence relation. We need to consider the previous states that can transition into the current state, and choose the best one:

```
@lru_cache(maxsize=None)
def get_best_profit(day, have_stock):
    """
```

```
    Returns the best profit that can be obtained by the end of the day.
    At the end of the day:
    * if have_stock is True, the trader must own the stock;
    * if have_stock is False, the trader must not own the stock.
    """
    if day < 0:
        if not have_stock:
            # Initial state: no stock and no profit.
            return 0
        else:
            # We are not allowed to have initial stock.
            # Add a very large penalty to eliminate this option.
            return -float('inf')
    price = daily_price[day]
    if have_stock:
        # We can reach this state by buying or holding.
        strategy_buy = get_best_profit(day - 1, False) - price
        strategy_hold = get_best_profit(day - 1, True)
        return max(strategy_buy, strategy_hold)
    else:
        # We can reach this state by selling or avoiding.
        strategy_sell = get_best_profit(day - 1, True) + price
        strategy_avoid = get_best_profit(day - 1, False)
        return max(strategy_sell, strategy_avoid)
```

The first part of the helper implements the termination condition, i.e. handling the initial state, while the second part implements the recurrence. To simplify the logic of the recurrence we allow selling on any day including the first, but we ensure that selling on the first day would yield a negative profit, so it's an option that cannot be chosen as optimal.

> **Handle the termination condition early**
>
> It is preferrable to handle the termination condition of a recursive function in a single place, as opposed to wrapping each call of the function with a check like `if day < 0` . . . . Handling it early simplifies greatly the logic and makes the code easier to read.

Both the time and space complexity of this solution are $O(n)$. Note that it is important to cache the results of the helper function, otherwise the time complexity becomes exponential instead of linear.

Solution 2: dynamic programming, bottom-up, $O(n)$ time

Once we have implemented the top-down solution, it is easy to rewrite it as bottom-up: we start from the initial state, and iterate day by day until we reach the final state:

```
def max_profit(daily_price):
    # Initial state: start from a reference cash amount.
    # It can be any value.
    # We use 0 and allow our cash to go below 0 if we need to buy a share.
    cash_not_owning_share = 0
    # High penalty for owning a stock initially:
    # ensures this option is never chosen.
    cash_owning_share = -float('inf')
    for price in daily_price:
        # Transitions to the current day, owning the stock:
        strategy_buy = cash_not_owning_share - price
        strategy_hold = cash_owning_share
        # Transitions to the current day, not owning the stock:
        strategy_sell = cash_owning_share + price
        strategy_avoid = cash_not_owning_share
        # Compute the new states.
        cash_owning_share = max(strategy_buy, strategy_hold)
        cash_not_owning_share = max(strategy_sell, strategy_avoid)
    # The profit is the final cash amount, since we start from
    # a reference of 0.
    return cash_not_owning_share
```

At each step, we only need to store the profit corresponding to the two states of that day. This is due to the state machine not having any transitions between non-consecutive days: we could say that at any time, the state machine does not "remember" anything from the days before yesterday.

The time complexity is $O(n)$, but the space complexity has been reduced to $O(1)$, since we only need to store the result for the previous day.

> **Bottom-up solutions often have smaller space complexity than top-down**
>
> It is very common (with some exceptions) that bottom-up solutions have lower memory requirements than top-down. This is due to the ability to control precisely what data we store and what data we discard: instead of a LRU cache policy, we can keep only the data we need. In addition, they do not suffer from the overhead of storing stack frames due to recursion, which is a hidden cost of the top-down solutions.

Variation: limited investment budget

In a variation of the problem, the investment budget is limited: we start with a fixed amount of money, and we are not allowed to buy a share if we cannot afford it (we cannot borrow money).

We can adjust the solution for this constraint:

```
def max_profit(daily_price, budget):
    # Initial state.
    cash_not_owning_share = budget
    # High penalty for owning a stock initially:
    # ensures this option is never chosen.
    cash_owning_share = -float('inf')
    for price in daily_price:
        # Transitions to the current day, owning the stock:
        strategy_buy = cash_not_owning_share - price
        strategy_hold = cash_owning_share
        # Transitions to the current day, not owning the stock:
        strategy_sell = cash_owning_share + price
        strategy_avoid = cash_not_owning_share
        # Compute the new states.
        cash_owning_share = max(strategy_buy, strategy_hold)
        if cash_owning_share < 0:
            # We cannot afford to buy the share at this time.
            # Add a high penalty to ensure we never choose this option.
            cash_owning_share = -float('inf')
        cash_not_owning_share = max(strategy_sell, strategy_avoid)
    return cash_not_owning_share - budget
```

Any time the optimal cash amount in a given state goes below zero, we replace it with negative infinity. This ensures that this path through the state machine will not be chosen. We only have to do this for the states where we own stock. In the states where we do not own stock, our cash amount never decreases from the previous day, so this check is not needed.

> **Expect follow-up questions**
>
> One of the most common mistakes candidates make is to assume that they have to solve a single problem during a time slot of the interview. Taking too long to solve it does not leave any time for follow-up questions, which puts the candidate at a disadvantage compared to others. An old Italian proverb applies here: *perfect is the enemy of good*. Do not get lost in too many details trying to make your solution perfect; reach an agreement with your interviewer on when it is good enough, so that you have time to take 1-2 follow-up questions such as this one.

Variation: limited number of transactions

In anover variation of the problem, the total number of transactions that can be performed is bounded: the stock can only be sold up to a certain number of times `tx_limit`.

In this variation, the state machine needs to be adjusted so that it keeps track of multiple pieces of information:

- whether we own the stock or not at the end of the day;

- how many times we have sold the stock so far.

Instead of having only 2 states each day (for owning or not owning the stock), we now have up to 2 * `tx_limit` depending on how many times we have sold, per day. For each of these states, we have to compute the best amount of money we can earn.

The transitions between the states need to take into account the operation (buying, holding, selling, avoiding) and whether it leads to the next day state with the same transaction count or one higher.

We can write the following implementation:

```
def max_profit(daily_price, tx_limit):
    # cash_not_owning_share[k] = amount of cash at the end of the day,
    # if we do not own the share, and we have sold k times so far.
    # Initially we have sold 0 times and we start from a reference
    # budget of 0. Any other state is invalid.
    cash_not_owning_share = [-float('inf')] * (tx_limit + 1)
    cash_not_owning_share[0] = 0
    # cash_owning_share[k] = amount of cash at the end of the day,
    # if we own the share, and we have sold k times so far.
    # Initially we do not own any stock, so set the state to invalid.
    cash_owning_share = [-float('inf')] * (tx_limit + 1)
    for price in daily_price:
        # Initialize the next day's states with -Infinity,
        # then update them with the best possible transition.
        cash_not_owning_share_next = [-float('inf')] * (tx_limit + 1)
        cash_owning_share_next = [-float('inf')] * (tx_limit + 1)
        for prev_tx_count in range(tx_limit):
            # Transition to the current day, owning the stock:
            strategy_buy = cash_not_owning_share[prev_tx_count] - price
            strategy_hold = cash_owning_share[prev_tx_count]
            # Transitions to the current day, not owning the stock:
            strategy_sell = cash_owning_share[prev_tx_count] + price
            strategy_avoid = cash_not_owning_share[prev_tx_count]
            # Compute the new states.
            if prev_tx_count < tx_limit:
                # Selling increases the tx_count by 1.
                cash_not_owning_share_next[prev_tx_count + 1] = max(
                        cash_not_owning_share_next[prev_tx_count + 1],
                        strategy_sell)
            # All other transitions keep tx_count the same.
            cash_not_owning_share_next[prev_tx_count] = max(
                    cash_not_owning_share_next[prev_tx_count],
                    strategy_avoid)
            cash_owning_share_next[prev_tx_count] = max(
                    cash_owning_share_next[prev_tx_count],
```

```
                strategy_buy,
                strategy_hold)
        cash_not_owning_share = cash_not_owning_share_next
        cash_owning_share = cash_owning_share_next
    # We have multiple final states, depending on how many times we sold.
    # The transaction limit may not have been reached.
    # Choose the most profitable final state.
    return max(cash_not_owning_share)
```

> **Master state machines**
>
> While you are reading this book, take your time to understand well how to model problems using state machines. Almost all dynamic programming problems can be solved this way. This skill is useful not just for interviews, but also in general in your software engineer career.