Example 188. Best Time to Buy and Sell Stock IV

Report Issue

> For this one, we're back to starting with top-down.

In this article, we'll be using the framework to solve Best Time to Buy and Sell Stock IV. This problem is rated as "hard" and may seem daunting at first, but with the framework, the logic behind solving this problem is very intuitive. We'll also make use of the pattern of "doing nothing". Like usual, let's use the framework to develop an algorithm:

1. A **function** that answers the problem for a given state

What information do we need at each state/decision?

We need to know what day it is (so we can look up the current price of the stock), and we need to know how many transactions we have left. These two are directly related to the input.

The note in the problem description says that we cannot engage in multiple transactions at the same time. This means that at any moment, we are either holding one unit of stock or not holding any stock. We should have a state variable that indicates if we are currently holding stock. This variable is fine as a boolean, but for caching purposes, let's use an integer alternating between 0 and 1 (0 means not holding, 1 means holding).

To summarize, we have 3 state variables:

1. i, which represents we are on the i^{th} day. The current price of the stock is $prices[i]$.
2. $transactionsRemaining$, which represents how many transactions we have left. This number goes down by 1 whenever we sell a stock.
3. $holding$, which is equal to 0 if we are not holding a stock, and 1 if we are holding a stock. If $holding$ is 0, we have the option to buy a stock. Otherwise, we have the option to sell a stock.

The problem is asking for a maximum achievable profit. Therefore, let's have a function dp where $dp(i, transactionsRemaining, holding)$ returns the maximum achievable profit starting from the i^{th} day with $transactionsRemaining$ transactions remaining, and $holding$ indicating if we start with a stock or not. To answer the original problem, we would return $dp(0, k, 0)$, as we start on day 0 with k transactions remaining and not holding a stock.

2. A **recurrence relation** to transition between states

At each state, we need to make a decision that depends on what $holding$ is. Let's split it up and look at our options one at a time:

- If we are holding stock, we have two options. We can sell, or not sell. If we choose to sell, we gain $prices[i]$ money, and the next state will be $(i + 1, transactionsRemaining - 1, 0)$. This is because it is the next day $(i + 1)$, we lose a transaction as we completed one by selling (

transactionsRemaining - 1), and we are no longer holding a stock (0). In total, our profit is prices[i] + dp(i + 1, transactionsRemaining - 1, 0). If we choose not to sell and **do nothing**, then we just move onto the next day with the same number of transactions, while still holding the stock. Our profit is dp(i + 1, transactionsRemaining, holding).

- If we are not holding stock, we have two options. We can buy, or not buy. If we choose to buy, we lose prices[i] money, and the next state will be (i + 1, transactionsRemaining, 1). This is because it is the next day, we have the same number of transactions because transactions are only completed on selling, and we now hold a stock. In total, our profit is -prices[i] + dp(i + 1, transactionsRemaining, 1). If we choose not to buy and **do nothing**, then we just move onto the next day with the same number of transactions, while still not having stock. Our profit is dp(i + 1, transactionsRemaining, holding).

> Note that you could also set up the solution so that transactions are completed upon buying a stock instead.

Of course, we always want to make the best decision. We can see that in both scenarios, **doing nothing** is the same - dp(i + 1, transactionsRemaining, holding). Therefore, we have a recurrence relation of:

`dp(i, transactionsRemaining, holding) = max(doNothing, sellStock)` if `holding == 1` otherwise `max(doNothing, buyStock)`

Where,
`doNothing = dp(i + 1, transactionsRemaining, holding)`,
`sellStock = prices[i] + dp(i + 1, transactionsRemaining - 1, 0)`, and
`buyStock = -prices[i] + dp(i + 1, transactionsRemaining, 1)`.

Recurrence relation for Best Time to Buy and Sell Stock IV

State variables:
i: represents day *i*
k: represents remaining transactions allowed
holding: 0 if not holding a stock, 1 if holding a stock

Next state	Buy a stock (holding = 0)	Sell a stock (holding = 1)	Do nothing (both cases)
i	i + 1	i + 1	i + 1
k	k	k - 1	k
holding	1	0	holding
profit	-prices[i] + dp(i+1, k, 1)	prices[i] + dp(i+1, k-1, 0)	dp(i+1, k, holding)

Always choose the option that maximizes profit.

3. **Base cases**

Both base cases are very simple for this problem. If we are out of transactions ($transactionsRemaining = 0$), then we should immediately return 0 as we cannot make any more money. If the stock is no longer on the market ($i = prices.length$), then we should also return 0, as we cannot make any more money.

Top-down Implementation

Bottom-up Implementation

Again, the recurrence relation is the same with top-down, but we need to be careful about how we configure our for loops. The base cases are automatically handled because the dp array is initialized with all values set to 0. For iteration direction and order, remember with bottom-up we start at the base cases. Therefore we will start iterating from the end of the input and with only 1 transaction remaining.

The time and space complexity of this problem for both implementations is the number of states since the recurrence relation is just a constant time formula. If $n = prices.length$, then this means the time and space complexity is $O(n \cdot k \cdot 2) = O(n \cdot k)$.