123. Best Time to Buy and Sell Stock III

Hard Array Dynamic Programming Leetcode Link

Problem Description

We are presented with an array called `prices`, where each element `prices[i]` represents the price of a stock on day `i`. The challenge is to determine the maximum profit that can be made through at most two stock transactions. A transaction consists of buying and then selling one share of the stock. It is important to note that you cannot hold more than one share at a time, which means you must sell the share you hold before buying another one. Therefore, the goal is to strategically choose two periods of time to buy and sell stocks to maximize your profit.

Intuition

The intuition behind the solution involves dynamic programming to keep track of profits across four different states, which represent the four actions you can take:

1. **Buying the first stock** (`f1`): For this action, we want to minimize the cost, so we keep track of the lowest price we've seen so far.
2. **Selling the first stock** (`f2`): Here, we calculate the profit from the first transaction. We aim to maximize the profit by selling at the highest price after buying at the lowest price.
3. **Buying the second stock** (`f3`): For this, we want the net cost to be minimal, which is purchasing a second stock at the lowest effective price after accounting for the profit from the first sale. This means we subtract the price of the second stock from the profit made from selling the first stock.
4. **Selling the second stock** (`f4`): Finally, we want to maximize the total profit, which comes from selling the second stock at the highest possible price.

The algorithm operates by iterating through the price array and updating these four states. Each price offers a potential to update these states—either you get a better buy price (`f1`), a better sell price resulting in higher first transaction profit (`f2`), a lower net buy price (after adjusting for profit from `f2`) for the second transaction (`f3`), or a better final sell price for a higher overall profit (`f4`).

By the end of the iteration, `f4` will represent the maximum profit achievable with up to two trades.

Solution Approach

The solution leverages dynamic programming to break down the problem into smaller subproblems and builds up the answer from those subproblems. Here's how the implementation carries this out:

1. **Initializing Variables**: We initialize four variables, `f1`, `f2`, `f3`, and `f4`, which correspond to the states of buying the first stock, selling the first stock, buying the second stock, and selling the second stock, respectively. Initially, we set `f1` and `f3` to `-prices[0]` since we "buy" the stock on day zero (the maximum profit after buying is the negative value of the stock's price), and `f2` and `f4` to `0` as we have not made any profit yet.
2. **Iterating through Prices**: We iterate through the `prices` array starting from day 1, updating the four states with each new price.
   - Update `f1` by taking the maximum of the current `f1` and the negative current price, indicating the purchase of the first stock at the lowest price seen so far.

     `f1 = max(f1, -price)`
   - Update `f2` by taking the maximum of the current `f2` and the current price plus `f1`. This reflects the sale of the first stock and captures the highest profit possible up to this point after the first transaction.

     `f2 = max(f2, f1 + price)`
   - Update `f3` by taking the maximum of the current `f3` and `f2` minus the current price. This reflects buying the second stock at the lowest effective price by considering the profits from the first sale.

     `f3 = max(f3, f2 - price)`
   - Update `f4` by taking the maximum of the current `f4` and the current price plus `f3`. This updates the total profit after the second sale is the highest we can get.

     `f4 = max(f4, f3 + price)`
3. **Returning Result**: After iterating through all prices, `f4` will hold the maximum profit after at most two transactions, which we return as the final result.

Each of the updates done in the loop effectively simulates all possible buy and sell actions that could be taken on that day and carries forward the best decision. The reason we update in the order `f1` → `f2` → `f3` → `f4` is due to the dependencies between the transactions: selling the first stock must consider its purchase and buying the second stock must consider the sale of the first, etc.

By the end of the loop, we have considered all possible scenarios of one or two transactions through the days, ensuring the maximum profit is calculated. The key aspect of this solution is it avoids the need for nested loops, which would significantly increase the computational complexity.

Example Walkthrough

Let's walk through the solution approach using a small example. Consider the following array for `prices`:

```
prices = [3, 4, 5, 1, 3, 2, 10]
```

In this array, `prices[i]` represents the price of the stock on day `i`. We want to calculate the maximum profit that can be made with at most two transactions.

Following the solution approach:

1. **Initializing Variables**:

- `f1` = -3 (since we "buy" the stock on day 0 at price 3)
- `f2` = 0 (no sale yet)
- `f3` = -3 (after "buying" the first stock, we haven't bought the second one yet)
- `f4` = 0 (no second sale yet)

2. **Iterating through Prices**:

   Day 1 (`price` = 4):

   - `f1` = max(-3, -4) = -3 (we keep the previous buy since it's cheaper)
   - `f2` = max(0, -3 + 4) = 1 (we can sell the first stock bought at -3 for 4)
   - `f3` = max(-3, 1 - 4) = -3 (considering profit from first sale, buying second stock at -3 is still better)
   - `f4` = max(0, -3 + 4) = 1 (after possible second sale)

   Day 2 (`price` = 5):

   - `f1` = max(-3, -5) = -3
   - `f2` = max(1, -3 + 5) = 2
   - `f3` = max(-3, 2 - 5) = -3
   - `f4` = max(1, -3 + 5) = 2

   Day 3 (`price` = 1):

   - `f1` = max(-3, -1) = -3
   - `f2` = max(2, -3 + 1) = 2
   - `f3` = max(-3, 2 - 1) = 1 (we now buy the second stock since we have a net profit after first sale)
   - `f4` = max(2, 1 + 1) = 2

   Day 4 (`price` = 3):

   - `f1` = max(-3, -3) = -3
   - `f2` = max(2, -3 + 3) = 2
   - `f3` = max(1, 2 - 3) = 1
   - `f4` = max(2, 1 + 3) = 4 (we sold the second stock we bought on day 3 for 3, making a profit)

   Day 5 (`price` = 2):

   - `f1` = max(-3, -2) = -3
   - `f2` = max(2, -3 + 2) = 2
   - `f3` = max(1, 2 - 2) = 1
   - `f4` = max(4, 1 + 2) = 4

   Day 6 (`price` = 10):

   - `f1` = max(-3, -10) = -3
   - `f2` = max(2, -3 + 10) = 7
   - `f3` = max(1, 7 - 10) = 1
   - `f4` = max(4, 1 + 10) = 11

3. **Returning Result**: Lastly, after iterating through all prices, `f4` holds the maximum profit of 11 after at most two transactions. Thus, we conclude the maximum profit possible is $11.

By following these steps using dynamic programming, we have efficiently computed the maximum profit possible with up to two transactions without needing to iterate through every possible pair of buy and sell days, which would be far less efficient.

Python Solution

```python
from typing import List

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        # Initialize the four states of profits
        # first_buy represents the profit after the first buy
        # first_sell represents the profit after the first sell
        # second_buy represents the profit after the second buy
        # second_sell represents the profit after the second sell
        first_buy, first_sell = -prices[0], 0
        second_buy, second_sell = -prices[0], 0

        # Loop through each price starting from the second price
        for price in prices[1:]:
            # Update first_buy to be either its previous value
            # or the negative of the current price (representing a buy)
            first_buy = max(first_buy, -price)

            # Update first_sell to be either its previous value
            # or the sum of first_buy and the current price (representing a sell)
            first_sell = max(first_sell, first_buy + price)

            # Update second_buy to be either its previous value
            # or the difference of first_sell and the current price (representing a second buy)
            second_buy = max(second_buy, first_sell - price)

            # Update second_sell to be either its previous value
            # or the sum of second_buy and the current price (representing a second sell)
            second_sell = max(second_sell, second_buy + price)

        # The maximum profit after two transactions is located in second_sell,
        # and that's the result to return
        return second_sell

# Example usage:
# sol = Solution()
# max_profit = sol.maxProfit([3,3,5,0,0,3,1,4])
# print(max_profit)  # Output would be 6, representing the maximum profit possible with two transactions

```

Java Solution

```java
public class Solution {
    public int maxProfit(int[] prices) {
        // f1 represents the maximum profit after the first buy
        int firstBuyProfit = -prices[0];

        // f2 represents the maximum profit after the first sell
        int firstSellProfit = 0;

        // f3 represents the maximum profit after the second buy
        int secondBuyProfit = -prices[0];

        // f4 represents the maximum profit after the second sell
        int secondSellProfit = 0;

        // Iterate through the list of prices starting from the second price
        for (int i = 1; i < prices.length; ++i) {
            // Update the maximum profit after the first buy
            firstBuyProfit = Math.max(firstBuyProfit, -prices[i]);

            // Update the maximum profit after the first sell
            // Equivalent to buying at 'firstBuyProfit' and selling at prices[i]
            firstSellProfit = Math.max(firstSellProfit, firstBuyProfit + prices[i]);

            // Update the maximum profit after the second buy
            // We subtract 'prices[i]' because it's the price we are buying at after the first sell
            secondBuyProfit = Math.max(secondBuyProfit, firstSellProfit - prices[i]);

            // Update the maximum profit after the second sell
            // Equivalent to buying at 'secondBuyProfit' and selling at prices[i]
            secondSellProfit = Math.max(secondSellProfit, secondBuyProfit + prices[i]);
        }

        // Return the maximum profit after the second sell
        // This is the maximum profit we can make with at most two transactions
        return secondSellProfit;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // for std::max function

class Solution {
public:
    int maxProfit(vector<int>& prices) {
        // Initialization of buy1 and sell1 corresponds to the first transaction.
        // buy1 is the maximum profit we can achieve by buying a stock on day 0,
        // which is simply the negative value of the stock price on that day.
        int buy1 = -prices[0];
        // sell1 is the maximum profit we can achieve by selling a stock after buying on day 0,
        // initially this would be zero because we have just bought a stock and yet to sell it.
        int sell1 = 0;

        // Initialization of buy2 and sell2 corresponds to the second transaction.
        // buy2 is the maximum profit we can potentially hold for a second buy,
        // initially it's the same as buy1 because we have not executed the first sell yet.
        int buy2 = -prices[0];
        // sell2 is the maximum profit we can get after completing the second sell,
        // initially, this would also be zero because we haven't sold any stock twice.
        int sell2 = 0;

        for (int i = 1; i < prices.size(); ++i) {
            // For each day, calculate the maximum profit if we were to buy the stock
            // for the first transaction, by comparing the previous buy profit and
            // the negative of today's price (as if we bought the stock today)
            buy1 = std::max(buy1, -prices[i]);

            // Calculate the maximum profit if we were to sell the stock
            // for the first transaction. Compare the previous sell profit and
            // the profit from selling today (today's price plus the current buy profit).
            sell1 = std::max(sell1, buy1 + prices[i]);

            // For the second buy, calculate the maximum profit by comparing the previous second buy profit
            // and the current overall profit minus today's price (buying the second stock today).
            buy2 = std::max(buy2, sell1 - prices[i]);

            // Finally, calculate the profit if we were to make the second sell today.
            // It compares the current second sell profit and the profit from selling today,
            // which includes the potential second buy profit and today's price.
            sell2 = std::max(sell2, buy2 + prices[i]);
        }

        // The result should be the maximum profit after two sales.
        return sell2;
    }
};

```

Typescript Solution

```typescript
function maxProfit(prices: number[]): number {
    // Initialize profit states for the two transactions
    // firstBuyProfit assumes the first transaction hasn't happened yet, hence negative value
    let firstBuyProfit = -prices[0];

    // firstSellProfit assumes no transaction has been made, hence 0 profit
    let firstSellProfit = 0;

    // secondBuyProfit takes into account profit from the first transaction
    let secondBuyProfit = -prices[0];

    // secondSellProfit represents the cumulative profit from both transactions
    // At start, no transaction has happened, hence 0 profit
    let secondSellProfit = 0;

    // Loop to calculate profits at each price point
    for (let i = 1; i < prices.length; ++i) {
        // The maximum profit of the first buy is either the previous value or
        // the negative of the current price (which means buying at the current price)
        firstBuyProfit = Math.max(firstBuyProfit, -prices[i]);

        // The maximum profit of the first sell is either the previous value or
        // the profit after selling at the current price
        firstSellProfit = Math.max(firstSellProfit, firstBuyProfit + prices[i]);

        // The maximum profit of the second buy is the greater of the previous value or
        // the current profit from the first sell minus the current price
        secondBuyProfit = Math.max(secondBuyProfit, firstSellProfit - prices[i]);

        // The maximum profit of the second sell is the greater of the previous value or
        // the profit after selling at the current price (including the profit from the first sell)
        secondSellProfit = Math.max(secondSellProfit, secondBuyProfit + prices[i]);
    }

    // Return the cumulative maximum profit from both allowed transactions
    return secondSellProfit;
}

```

Time and Space Complexity

The given Python code represents a dynamic programming approach to solve a stock trading problem, where an individual is allowed to make at most two transactions to maximize profit. The variables `f1`, `f2`, `f3`, and `f4` represent the four states of profits after each transaction or waiting period.

Time Complexity:

The time complexity of the code is determined by:

- The number of iterations in the loop, which is `n-1` where `n` is the length of the `prices` list.
- The constant time operations within each iteration, which don't depend on the size of the input.

There are four assignments in each iteration that happen in `O(1)` time each.

Thus, the total time complexity is `O(n-1) * O(1)`, which simplifies to `O(n)`.

Space Complexity:

The space complexity of the code is determined by:

- The space taken by the four variables `f1`, `f2`, `f3`, `f4`, which is constant and does not scale with input size.
- No additional data structures are used that grow with input size.

Hence, the space complexity of the code is `O(1)`, representing the constant space used by the variables.