714. Best Time to Buy and Sell Stock with Transaction Fee

Medium | Greedy | Array | Dynamic Programming — Leetcode Link

Problem Description

You are provided with an array called `prices`, where each element `prices[i]` represents the price of a particular stock on the `i-th` day. In addition, you are given an integer `fee` that represents a transaction fee. Your goal is to calculate the maximum profit you can achieve from trading the stock. You can make as many trades as you like, but for every trade you make (i.e., whenever you buy and then sell a stock), you must pay a transaction fee.

There are a couple of rules that you must follow as part of your trading strategy:

- You cannot hold more than one transaction at a time. Essentially, you must sell the stock before you can buy again.
- The transaction fee is charged once per round-trip (buy and then sell) of trading a stock.

Your task is to determine the strategy that maximizes your profit given these constraints.

Intuition

The core of solving this problem lies in understanding the state of each day: either you have stock or you don't. The maximum profit for each day depends on the actions you could take on that day, which in turn depend on whether you are holding a stock at the end of the previous day.

The intuition for the provided solution involves keeping track of two variables as we iterate through the array `prices`:

- `f0`: The maximum profit we can have up to the current day if we do not hold stock by the end of the day.
- `f1`: The maximum profit we can have if we do hold stock by the end of the day.

As we iterate through the prices:

1. For each day, we calculate the new `f0` as the maximum of its previous value or the value of `f1` plus the sell price of the stock minus the fee. Essentially, this represents not making a transaction or selling the stock we hold.
2. Concurrently, we calculate the new `f1` as the maximum of its previous value or the value of `f0` minus the price of buying the stock. This represents either continuing to hold the stock we have or buying new stock.

The base case for `f1` is `-prices[0]` because we consider that we buy a stock on the first day and hence, the initial profit is negative by the price of the stock. For `f0`, the base case is `0`, which means we start with no stock and no profit.

So in a nutshell, on any day, your decision to sell or hold the stock reflects on the maximum profit you could get by that day. This pattern keeps track of profits efficiently by reducing the original problem into smaller subproblems and builds on their results, which is a key idea in dynamic programming.

Solution Approach

The solution uses dynamic programming to optimize the process of finding the maximum profit. It builds up the solution by breaking the problem down into smaller subproblems and storing their results to avoid re-computation. The reference solution approaches give us two perspectives on the problem: memoization (top-down approach) and tabulation (bottom-up approach).

Memoization (Top-Down Approach)

In this approach, we define a recursive function `dfs(i, j)` representing the maximum profit from the `i-th` day with state `j`, where `j` can be either `0` (not holding a stock) or `1` (holding a stock).

- If `i >= n`, where `n` is the length of the `prices` array, it means there are no more days left for trading, and hence the profit is `0`.
- If `j = 0`, we have two options: either do not engage in any trade, which means the profit remains the same as the next day without a stock (`dfs(i + 1, 0)`), or sell our stock (if we have one) and add the price to our profit minus the fee (`prices[i] + dfs(i + 1, 0) - fee`).
- If `j = 1`, we again have two options: either keep holding the stock, so there's no change in profit (`dfs(i + 1, 1)`), or buy a stock, costing us the current price (`-prices[i] + dfs(i + 1, 1)`).

A memoization array `f` is used to store the results of `dfs(i, j)` to avoid recalculating the same state. The time complexity is `O(n)`, and the space complexity is also `O(n)`.

Dynamic Programming (Bottom-Up Approach)

The bottom-up dynamic programming approach defines a 2-dimensional array `f[i][j]`, where `i` represents the day, and `j` represents whether or not we are holding a stock.

- We initialize `f[0][0]` to `0` because on the first day, if we don't hold any stock, the profit is zero.
- We initialize `f[0][1]` to `-prices[0]` as if we buy the stock on the first day, our profit is negative (the cost of the stock).

For subsequent days (`i >= 1`), we iterate through the prices array and determine:

- `f[i][0]`: The maximum profit for not holding a stock, which comes from either not doing any transaction the previous day (`f[i - 1][0]`) or selling the stock we had (`f[i - 1][1]` plus `prices[i]` minus `fee`).
- `f[i][1]`: The maximum profit for holding a stock, which comes from either keeping the stock we had the previous day (`f[i - 1][1]`) or buying a new stock (`f[i - 1][0]` minus `prices[i]`).

Finally, the answer is obtained by looking at `f[n - 1][0]`, where `n` is the length of `prices`, which gives us the maximum profit on the last day when we are not holding any stock. The time complexity is `O(n)`, and the space complexity can be optimized to `O(1)` if we only keep track of the last state because each state only depends on the previous state.

The provided solution code implements the second approach in a space-optimized manner, collapsing the 2-dimensional array to just two variables because the state for each day only depends on the previous day:

```python
class Solution:
    def maxProfit(self, prices: List[int], fee: int) -> int:
        f0, f1 = 0, -prices[0]
        for x in prices[1:]:
            f0, f1 = max(f0, f1 + x - fee), max(f1, f0 - x)
        return f0
```

Each iteration updates `f0` and `f1` to reflect the current day's best choices for stock trading.

Example Walkthrough

Let's consider a small example to illustrate the solution approach with `prices = [1,3,2,8,4,9]` and `fee = 2`.

Initially, we have `f0 = 0`, as we hold no stock and therefore have no profit, and `f1 = -prices[0]`, meaning we buy the stock on the first day which gives us `f1 = -1`.

On day 1 (prices[1] = 3), we have two options:

- Sell the stock we bought on day 0 for `3` and pay a fee of `2`. So, `f0 = max(f0, f1 + prices[1] - fee) = max(0, -1 + 3 - 2) = 0`.
- Keep holding the stock we bought on day 0. So, `f1 = max(f1, f0 - prices[1]) = max(-1, 0 - 3) = -1`.

After day 1, `f0 = 0`, `f1 = -1`.

On day 2 (prices[2] = 2), the choices are:

- Again, don't sell any stock. So, `f0 = max(f0, f1 + prices[2] - fee) = max(0, -1 + 2 - 2) = 0`.
- Buy a stock (if we have no stock), which will cost us `2`. So, `f1 = max(f1, f0 - prices[2]) = max(-1, 0 - 2) = -1`.

After day 2, `f0 = 0`, `f1 = -1`.

We then repeat this process for each day:

On day 3 (prices[3] = 8), we have:

- `f0 = max(f0, f1 + prices[3] - fee) = max(0, -1 + 8 - 2) = 5`.
- `f1 = max(f1, f0 - prices[3]) = max(-1, 0 - 8) = -1`.

After day 3, `f0 = 5`, `f1 = -1`.

On day 4 (prices[4] = 4), we have:

- `f0 = max(f0, f1 + prices[4] - fee) = max(5, -1 + 4 - 2) = 5`.
- `f1 = max(f1, f0 - prices[4]) = max(-1, 5 - 4) = 1`.

After day 4, `f0 = 5`, `f1 = 1`.

On day 5 (prices[5] = 9), we have:

- `f0 = max(f0, f1 + prices[5] - fee) = max(5, 1 + 9 - 2) = 8`.
- `f1 = max(f1, f0 - prices[5]) = max(1, 5 - 9) = 1`.

After day 5, `f0 = 8`, `f1 = 1`.

At the end of the trading period, the maximum profit with no stock in hand is `f0 = 8`, which is our answer.

Throughout this process, we dynamically choose whether to hold or sell the stock each day based on which option will give us a higher profit, factoring in the fee for each sale.

Python Solution

```python
# The class Solution contains a method to calculate the maximum profit from trading stocks,
# given an array that represents the price of a stock on different days, and a fixed transaction fee.
class Solution:
    def maxProfit(self, prices: List[int], fee: int) -> int:
        # Initialize cash and hold variables:
        # cash represents the max profit achievable without holding any stock
        # hold represents the max profit achievable while holding a stock
        cash, hold = 0, -prices[0]

        # Iterate through the list of prices, starting from the second price
        for price in prices[1:]:
            # Update cash to the max of itself or the profit from selling a stock at the current price minus the fee
            # Update hold to the max of itself or the value of the cash after buying a stock at the current price
            cash, hold = max(cash, hold + price - fee), max(hold, cash - price)

        # The value of cash at the end of iteration will represent the maximum profit achievable
        return cash

```

Java Solution

```java
class Solution {
    public int maxProfit(int[] prices, int fee) {
        // Initialize cash (f0) to represent max profit with 0 stocks on hand
        // Initialize hold (f1) to represent max profit with 1 stock on hand - bought on the first day
        int cash = 0, hold = -prices[0];

        for (int i = 1; i < prices.length; ++i) {
            // Calculate the new cash by selling the stock held today, if it's a better option than holding cash
            int newCash = Math.max(cash, hold + prices[i] - fee);
            // Calculate the new hold by buying the stock today, if it's a better option than holding the current stock
            hold = Math.max(hold, cash - prices[i]);

            // Update cash to the newly calculated max profit with 0 stocks
            cash = newCash;
        }

        // Finally, return the cash, which represents the maximum profit with 0 stocks on hand after all transactions
        return cash;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <cstring>
using namespace std;

class Solution {
public:
    int maxProfit(vector<int>& prices, int fee) {
        // Determine the number of days in the given price array
        int numberOfDays = prices.size();

        // f[i][0] represents the maximum profit at day i when we do not have a stock
        // f[i][1] represents the maximum profit at day i when we have a stock
        int profit[numberOfDays][2];

        // Initializing the profit array with zeros
        memset(profit, 0, sizeof(profit));

        // Base case: On day 0, if we buy a stock,
        // the profit is negative because of the stock price.
        profit[0][1] = -prices[0];

        // Iterate over each day starting from day 1
        for (int i = 1; i < numberOfDays; ++i) {
            // Either keep the maximum profit without stock from the previous day,
            // or sell the stock bought on a previous day for today's price minus
            // the transaction fee to maximize profit.
            profit[i][0] = max(profit[i - 1][0], profit[i - 1][1] + prices[i] - fee);

            // Either keep the maximum profit with a stock from the previous day,
            // or buy a stock today using the maximum profit without a stock
            // from the previous day minus today's stock price to maximize profit.
            profit[i][1] = max(profit[i - 1][1], profit[i - 1][0] - prices[i]);
        }

        // The answer will be the maximum profit at the last day when we do not have a stock,
        // because that represents the maximum profit we can earn.
        return profit[numberOfDays - 1][0];
    }
};

```

Typescript Solution

```typescript
function maxProfit(prices: number[], fee: number): number {
    const numPrices = prices.length; // Total number of prices
    let noStockProfit = 0; // Maximum profit when not holding any stock
    let inHandProfit = -prices[0]; // Maximum profit when holding stock, initially after buying first stock

    // Starting from the second price, determine the max profit by either keeping/selling the stock or buying a stock
    for (const currentPrice of prices.slice(1)) {
        // Calculate the profit if we sell the stock at the current price (-fee) or keep the profit as is
        noStockProfit = Math.max(noStockProfit, inHandProfit + currentPrice - fee); // Max profit after selling the stock
        // Calculate the profit if we buy the stock at the current price or keep the profit as is
        inHandProfit = Math.max(inHandProfit, noStockProfit - currentPrice); // Max profit of holding the stock
    }

    // The max profit is when we don't hold any stock at the end
    return noStockProfit;
}

```

Time and Space Complexity

The given code achieves the objective of finding the maximum profit with a transaction fee by using dynamic programming with state compression.

The time complexity is `O(n)`, where `n` is the length of the `prices` array. This is because the code iterates through the `prices` array once, and within each iteration, it performs a constant number of computations.

The space complexity is `O(1)`. Instead of using an `n x 2` array to hold the state of the maximum profit on each day, two variables, `f0` and `f1`, are used, maintaining the state of the system at the current and previous steps. The reference answer explains that previously a larger array `f[i][]` was used and that space complexity has been reduced by only keeping track of the necessary states for calculation.