188. Best Time to Buy and Sell Stock IV

Hard | Array | Dynamic Programming | Leetcode Link

Problem Description

You are given an integer array `prices`, which represents the price of a given stock on different days, where `prices[i]` is the price on day `i`. Additionally, you're given an integer `k`. Your task is to calculate the maximum profit you can achieve from at most `k` transactions. A transaction consists of buying and then selling one share of the stock — You can only hold one share of the stock at a time, meaning you need to sell it before buying another.

To summarize, the goal is to find the best days to buy and sell the stock so that the total profit, after at most `k` transactions, is as high as possible without conducting simultaneous transactions.

Intuition

Given the constraint of a maximum of `k` transactions, it's clear that this problem cannot be solved by a simple greedy strategy that tends to work when there's no limit on the number of transactions. Efficient solutions to this problem rely on dynamic programming, where we track the state of our transactions at each step.

The intuition behind the solution is the following:

- Every day, you have the option either to do nothing or to take an action (buy or sell), depending on whether you're currently holding a stock.
- If you're holding a stock, you can either sell it or continue holding it for potentially better future profits.
- If you're not holding a stock, you can either buy a stock (provided you have remaining transactions) or wait.

Thus, a state is defined by three parameters: the current day, the number of transactions left, and whether you're holding a stock.

The dynamic programming approach typically involves breaking down the problem into smaller subproblems and building up a solution from there, using a table (or, in memory-efficient solutions, two arrays) to keep track of the state transitions—namely, the maximum profit from each state. The transition rules govern how you can move from one state to another and thereby form the essence of the solution.

For any given day, the maximum profit depends on the maximum profits of the previous days and the current stock price. The solution code simplifies the space complexity by maintaining only two columns for holding and not holding states, running through the prices, and updating the state values in place. This way, we derive our maximum profit without needing to store the entire table's worth of data.

Solution Approach

The implementation for finding the maximum profit with at most `k` transactions uses dynamic programming. Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable when the problem can be divided into overlapping subproblems that can be solved independently.

Here's the approach:

1. **State Definition:**
   - Define `f` with the dimensions `[k + 1]` by `2` to represent the state table where `f[j][0]` is the maximum profit up to day `i` with `j` transactions and not holding a stock, while `f[j][1]` is the maximum profit up to day `i` with `j` transactions and holding a stock.
2. **Base Case Initialization:**
   - Initialize the first day's state because we know that on the first day (before any transactions and stock prices come into play), if we are not holding a stock, our profit is 0, and if we are holding a stock, then the profit is negative because it's an expense: `-prices[0]`.
3. **State Transition:**
   - Loop through each price `x` in `prices[1:]`, starting from the second day, since the first day is used for initialization.
   - For each day and for transaction counts from `k` down to `1` (to ensure we reset the holding state before using it for the next transaction) we update our states:
     - `f[j][0]` is updated to the maximum of either selling the stock we're holding at current price `x` or not selling it (keeping the previous profit without holding a stock).
     - `f[j][1]` is updated to the maximum of either buying the stock at current price `x` using one of our transactions or not buying it (keeping the previous profit with holding a stock).
4. **State Transitions Equations:** The transition function is based on the choices we have on each day `i` for transaction `j`:

```
// Not holding a stock
f[j][0] = max(f[j][1] + price[i], f[j][0])
// Holding a stock
f[j][1] = max(f[j - 1][0] - price[i], f[j][1])
```

   This also adheres to the constraints that you must sell the stock before you can buy again.

5. **Answer Retrieval:**
   - The final answer is the profit at the last transaction having sold any stock, which is `f[k][0]`.

The critical understanding in the dynamic programming approach involves deciding the state dimensions and transitions that capture the essence of the problem while optimizing the space used by re-using the state information of the previous day as we move to the next day.

In terms of complexity:

- The time complexity of this algorithm is `O(n * k)` because we go through all the stock prices (n steps) and for each price, we go through `k` transactions.
- The space complexity of this algorithm is `O(k)` since we only keep track of the most recent `2 * (k + 1)` states.

This presented approach succinctly captures the problem's constraints and optimizes for space while providing an efficient solution to the maximum profit problem with at most `k` stock transactions.

Example Walkthrough

Let's take an example to illustrate the solution approach for the problem statement given. Suppose we have the price array `prices = [3,2,6,5,0,3]` which represents the price of the stock on different days, and we're allowed at most `k = 2` transactions.

Initialization

We start by initializing our DP table `f` with dimensions `[k + 1][2]`, meaning it would have size `[3][2]` for this example (including the `0`th transaction). Initially, our DP table looks like this (note that it has not yet been filled with meaningful values):

```
f = [
  [0, -inf],  // 0 transactions, not holding and holding (impossible)
  [0, -inf],  // 1 transaction, not holding and holding
  [0, -inf],  // 2 transactions, not holding and holding
]
```

Base Case

Before any prices come into play, we cannot have a profit with 0 transactions, so `f[0][0]` is `0`. On the first day, if we buy at price `3`, our base case for holding a stock after 1 transaction is `f[1][1] = -3`.

```
f = [
  [0, -inf],
  [0, -3],
  [0, -inf]
]
```

Process Day by Day

Let's walk through the processing of each day:

Day 1: Price is 2.

- `f[1][0]`: If we are holding, we can sell for a profit of `2 - (-3) = 5`. So max(0, 5) = 5 (we choose to sell).
- `f[1][1]`: If we are not holding, we can buy at `2`. So max(-3, -2) = -2 (we choose to buy at a lower price).

We update our states:

```
f = [
  [0, -inf],
  [5, -2],
  [0, -inf]
]
```

Day 2: Price is 6.

- `f[1][0]`: Max of not selling (`5`) or selling what we have (`6 - (-2) = 8`). We choose 8 (we sell).
- `f[1][1]`: Max of keeping the previous stock (`-2`) or buying new at `6` (but we would need 0 transactions left over, which we do not). We keep `-2`.
- `f[2][0]`: Max of not doing a new transaction (`0`) or selling what we had from the previous transaction (`6 - (-2) = 8`). We choose 8 (if we had sold earlier and bought today, this would be a second transaction).
- `f[2][1]`: Max of buying a new one with the profit from the first transaction (`5 - 6 = -1`) or keeping the old case (`-inf`). We choose -1.

```
f = [
  [0, -inf],
  [8, -2],
  [8, -1]
]
```

(Note: As we do not have transaction 0 holding a stock (`f[0][1]`), it doesn't affect our calculations and stays `-inf`.)

Day 3: Price is 5.

- `f[1][0]`: Max of not selling (`8`) or selling what we have (`5 - (-2) = 7`). We keep 8.
- `f[1][1]`: Max of keeping the previous stock (`-2`) or buying new for less (`-5` but using a previous transaction). We choose `-2`.
- `f[2][0]`: Max of not doing a new transaction (`8`) or selling what we have (`5 - (-1) = 6`). We choose 8.
- `f[2][1]`: Max of keeping the old transaction (`-1`) or buying a new one (`8 - 5 = 3`), but that's a negative profit, so we stay with -1.

```
f = [
  [0, -inf],
  [8, -2],
  [8, -1]
]
```

Day 4: Price is 0.

- `f[1][0]`: Max of not selling (`8`) or selling (`0 - (-2) = 2`), stick with 8.
- `f[1][1]`: Max of keeping the old stock (`-2`) or buying a new one (`8 - 0 = 8`), but that's not valid (as we cannot sell and buy on the same day).
- `f[2][0]`: Max of not doing a transaction (`8`) or selling (`0 - (-1) = 1`), we keep 8.
- `f[2][1]`: Max of keeping the old stock (`-1`) or buying new (`8 - 0 = 8`), we keep -1.

Our updated DP table remains the same as the previous step:

`Day 5` with price 3 is processed similarly, and we keep updating our DP table based on the rules.

After processing all days, the DP table will reflect the maximum profit possible at each day for at most `k` transactions, not holding or holding a stock.

At the end of all the updates, the maximum profit with k transactions and not holding any stock on the last day will be given by `f[k][0]`. For our example with `prices = [3,2,6,5,0,3]` and `k = 2`, the final DP table would represent the maximum profit we can achieve, and `f[2][0]` would give us the answer.

Python Solution

```python
from typing import List

class Solution:
    def maxProfit(self, max_transactions: int, stock_prices: List[int]) -> int:
        # Edge case: if there are no prices or max transactions is zero
        if not stock_prices or max_transactions == 0:
            return 0

        # Initialize a 2D list to hold the state of profits
        # f[transactions][holding]
        # holding = 0 (not holding any stock), 1 (holding stock)
        profits = [[0] * 2 for _ in range(max_transactions + 1)]

        # Initialize the case where we've made one transaction, but are holding a stock,
        # which is the negative value of the first price
        for j in range(1, max_transactions + 1):
            profits[j][1] = -stock_prices[0]

        # Loop through all prices starting from the second day
        for price in stock_prices[1:]:
            # Go through all possible transaction numbers
            for j in range(max_transactions, 0, -1):
                # Update the profit when not holding a stock:
                # max(selling the stock today, not selling and keeping the previous profit)
                profits[j][0] = max(profits[j][1] + price, profits[j][0])

                # Update the profit when holding a stock:
                # max(buying the stock today, not buying and keeping the previous profit)
                profits[j][1] = max(profits[j - 1][0] - price, profits[j][1])

        # Return the maximum profit after the allowable number of transactions
        # without holding any stock
        return profits[max_transactions][0]

```

Java Solution

```java
class Solution {
    public int maxProfit(int k, int[] prices) {
        // The length of the given prices array
        int numberOfDays = prices.length;

        // Initialize an array to store the profit at each transaction stage
        // f[transactionCount][holding] with 0 meaning not holding a stock and 1 meaning holding a stock
        int[][] profits = new int[k + 1][2];

        // For the case where we have not made any transaction, but we are holding a stock,
        // the initial profit should be negative, which is the cost of the stock on the first day
        for (int transactionCount = 1; transactionCount <= k; ++transactionCount) {
            profits[transactionCount][1] = -prices[0];
        }

        // Iterate through each day starting from the second day
        for (int dayIndex = 1; dayIndex < numberOfDays; ++dayIndex) {
            // Iterate for each possible transaction from k down to 1
            for (int transactionCount = k; transactionCount > 0; --transactionCount) {
                // Update profit for the case we sell the stock today:
                // Max of (previous profit for holding a stock and selling it today, previous profit without holding any stock)
                profits[transactionCount][0] = Math.max(profits[transactionCount][1] + prices[dayIndex], profits[transactionCount][0]

                // Update profit for the case we buy the stock today:
                // Max of (profit from the last transaction minus the price of today's stock, previous profit with holding a stock)
                profits[transactionCount][1] = Math.max(profits[transactionCount - 1][0] - prices[dayIndex], profits[transactionCount
            }
        }

        // Return the maximum profit possible without holding a stock after k transactions
        return profits[k][0];
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <cstring>
#include <algorithm>

using namespace std;

class Solution {
public:
    int maxProfit(int numTransactions, vector<int>& stockPrices) {
        int numDays = stockPrices.size();
        if (numDays == 0) return 0; // If there are no stock prices, no profit can be made.

        // Create a 2D dp array with numTransactions + 1 rows and 2 columns.
        // dp[transactions][holding] represents the maximum profit that can be made
        // after 'transactions' transactions and whether we are 'holding' a stock (1) or not (0).
        int dp[numTransactions + 1][2];

        // Initialize the dp array to 0.
        memset(dp, 0, sizeof(dp));

        // For each possible number of transactions, set an initial state indicating we bought
        // one stock at the first day's price, thus a negative profit.
        for (int j = 1; j <= numTransactions; ++j) {
            dp[j][1] = -stockPrices[0];
        }

        // Iterate through all days of stock prices.
        for (int i = 1; i < numDays; ++i) {
            // Iterate through all transactions in reverse order.
            for (int j = numTransactions; j > 0; --j) {
                // Update dp array for the two states:
                // 1. Not holding a stock after selling on the ith day.
                dp[j][0] = max(dp[j][1] + stockPrices[i], dp[j][0]);
                // 2. Holding a stock after buying on the ith day.
                dp[j][1] = max(dp[j - 1][0] - stockPrices[i], dp[j][1]);
            }
        }

        // The final answer is the max profit after 'numTransactions' transactions
        // without holding a stock.
        return dp[numTransactions][0];
    }
};

```

Typescript Solution

```typescript
function maxProfit(maxTransactions: number, stockPrices: number[]): number {
    // Initialize a 2D array to hold the maximum profit up to the i-th transaction and stock holding state
    // f[i][0] represents the maximum profit with at most i transactions and no stock currently held
    // f[i][1] represents the maximum profit with at most i transactions and a stock currently held
    let profit = Array.from({ length: maxTransactions + 1 }, () => Array.from({ length: 2 }, () => 0));

    // Initialize the case where we have performed transactions but holding a stock with -stockPrices[0]
    for (let transaction = 1; transaction <= maxTransactions; ++transaction) {
        profit[transaction][1] = -stockPrices[0];
    }

    // Loop through each price starting from the second day, as we have already used the first price
    for (const currentPrice of stockPrices.slice(1)) {
        // Start from maxTransactions and go down to 1. We skip the 0th transaction count as it means no transactions
        for (let transaction = maxTransactions; transaction > 0; --transaction) {
            // Calculate the Max profit for the state of not holding a stock
            profit[transaction][0] = Math.max(profit[transaction][1] + currentPrice, profit[transaction][0]);

            // Calculate the Max profit for the state of holding a stock
            profit[transaction][1] = Math.max(profit[transaction - 1][0] - currentPrice, profit[transaction][1]);
        }
    }

    // Return the maximum profit after maxTransactions have been completed without holding a stock
    return profit[maxTransactions][0];
}

```

Time and Space Complexity

Time Complexity

The given code has two nested loops:

- The outer loop runs through the price list once, with the exception of the first element, so it runs in `O(N)`, where `N` is the length of the `prices` list.
- The inner loop is running from `k` to `1`, thus it runs `k` times for each iteration of the outer loop.

Therefore, the total time complexity is the product of the two, which is `O(N * k)`.

Space Complexity

Before optimization, the solution uses a 2D list `f` with `(k + 1) * 2` elements, regardless of the size of `prices`. Thus, the space complexity is `O(k)`.

According to the reference answer, by optimizing the first dimension of the space, the space complexity can be further reduced. This implies that instead of storing all `k + 1` states for the 'sell' and 'buy' scenarios, we can only keep track of the current and previous states. If we store just one state and update it as we go, we only need `2 * k` space, one for each 'buy' and 'sell' at every transaction number up to `k`. Hence, after optimization, space complexity remains `O(k)` since it is not dependent on `N` and we are only optimizing in terms of `k`.