122. Best Time to Buy and Sell Stock II

Medium Greedy Array Dynamic Programming

Problem Description

This problem presents a scenario where you have an array `prices`, with each element representing the price of a stock on an ith day. The objective is to determine the maximum profit you can achieve from buying and selling these stocks. However, there are some rules you must follow:

1. You can only hold at most one share of the stock at any time.
2. You can buy and sell the stock on the same day.

The goal is to figure out the strategy that allows you to make the maximum profit from these transactions.

Intuition

The intuition behind the solution is grounded in the idea of taking advantage of every profitable opportunity. You scan through the given list of prices and every time you see that the price of the stock on the next day is higher than the current day, you simulate a buy-and-sell transaction to gain profit. This approach is rooted in a greedy algorithm strategy which focuses on making the most beneficial decision at every single step without considering the larger picture.

In mathematical terms, if `prices[i] < prices[i + 1]`, then you would want to buy on day i and sell on day i+1 to gain `prices[i + 1] - prices[i]` profit. If you repeat this process for every increase in the stock price, the sum of all these individual profits will give you the maximum profit that can be achieved.

What is important to note here is that you're not looking to find the highest price to sell and the lowest price to buy in the entire array – that would be a different problem. Instead, you're accumulating profits from every single upswing (every time the stock price increases from one day to the next), thus maximizing your total profit.

The specified algorithm runs with a time complexity of O(n), which means it looks at each element in `prices` only once. The space complexity is O(1) since no additional space is used that grows with the size of the input; the calculation is done using a fixed amount of extra space.

Solution Approach

The solution adopts a simple yet efficient greedy algorithm. Greedy algorithms make the optimal choice at each step, hoping to find the global optimum. In this case, the algorithm buys and sells stock based on the change in price between consecutive days.

No complex data structures are needed; the algorithm employs a straightforward iteration through the `prices` array. The Python function `pairwise` from the `itertools` module is used to create a pair of consecutive elements. This could be replicated manually by iterating through the indices of `prices` and accessing `prices[i]` and `prices[i+1]` for comparison. However, using `pairwise` simplifies the iteration.

The `maxProfit` function iterates over each pair of consecutive prices `(a, b)` obtained from `pairwise(prices)`. It calculates the difference `b - a`, which represents the potential profit one can make if they buy the stock on day `i` (corresponding to price `a`) and sell it on day `i+1` (corresponding to price `b`). If `b` is greater than `a`, there is a positive profit, and that value is summed to the cumulative profit. If there is no profit (i.e., `b` is less than or equal to `a`), then `max(0, b - a)` yields 0, contributing nothing to the profit.

This process loop continues for each pair of consecutive days. All positive profits are accumulated to yield the maximum profit you can achieve by the end of the array.

Using the code given:

```python
def maxProfit(self, prices: List[int]) -> int:
    return sum(max(0, b - a) for a, b in pairwise(prices))
```

The `maxProfit` function is a one-liner that maps the `pairwise` list to the maximum differences and sums them up. It captures the essence of the greedy approach, which is to collect every small profit available to maximize the total return.

In conclusion, this approach relies on the greedy algorithm principle to identify and take advantage fast of profitable price differences, efficiently implemented in Python with minimal complexity. Its time complexity remains linear, `O(n)`, since it goes through the list just once, and it has a constant space complexity, `O(1)`, since no additional space is proportional to the input size is utilized.

Example Walkthrough

Let's say we have an array of prices:

```
prices = [7, 1, 5, 3, 6, 4]
```

The prices correspond to the price of the stock on days 0 through 5. To maximize profit using the aforementioned greedy strategy, you look for every opportunity where the price of the stock goes up from one day to the next.

Walk through the prices array:

- Day 0 to Day 1: The price goes from 7 to 1. No profit is possible because the price drops. Therefore, the profit is `max(0, 1 - 7)` = 0.
- Day 1 to Day 2: The price goes from 1 to 5. The profit is `max(0, 5 - 1)` = 4. You buy at 1 and sell at 5.
- Day 2 to Day 3: The price goes from 5 to 3. No profit is possible because the price drops. The profit is `max(0, 3 - 5)` = 0.
- Day 3 to Day 4: The price goes from 3 to 6. The profit is `max(0, 6 - 3)` = 3. You buy at 3 and sell at 6.
- Day 4 to Day 5: The price goes from 6 to 4. No profit, as the price drops. The profit is `max(0, 4 - 6)` = 0.

Now, you sum up all the profits from the transactions where it was profitable to buy and sell:

Profit = 0 + 4 + 0 + 3 + 0 = 7

The maximum profit that can be achieved with the given prices array is 7.

This is the same as running the provided function `maxProfit` with the input:

```python
def maxProfit(prices: List[int]) -> int:
    return sum(max(0, b - a) for a, b in pairwise(prices))

print(maxProfit([7, 1, 5, 3, 6, 4]))  # Output: 7
```

The function iterates through `pairwise(prices)`, which would yield the pairs (7, 1), (1, 5), (5, 3), (3, 6), and (6, 4), calculates the differences, and sums the positive differences to find the maximum profit, which in this example is 7.

Solution Implementation

Python

```python
from itertools import pairwise  # Importing the pairwise function from itertools module

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        # Initialize total profit to zero
        total_profit = 0

        # Loop through each pair of successive prices using pairwise()
        for buy_price, sell_price in pairwise(prices):
            # Calculate profit for the current pair
            # If sell price is greater than buy price, add the difference to total profit
            # Otherwise, add zero (no loss, no gain)
            profit = max(0, sell_price - buy_price)
            total_profit += profit

        # Return the total calculated profit
        return total_profit
```

Java

```java
class Solution {

    // Method to calculate the maximum profit that can be achieved
    // by buying and selling stocks on different days
    public int maxProfit(int[] prices) {
        int totalProfit = 0; // Initialize total profit to zero

        // Loop through the array of prices
        for (int i = 1; i < prices.length; ++i) {
            // Calculate the profit for the current day by subtracting the previous day's price from the current day's price
            int dailyProfit = Math.max(0, prices[i] - prices[i - 1]);

            // Add the daily profit to the total profit
            // This will accumulate if buying on the i-1 day and selling on the i day is profitable
            totalProfit += dailyProfit;
        }

        // Return the total profit accumulated
        return totalProfit;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // For the max() function

class Solution {
public:
    // Function to calculate the maximum profit from stock prices
    int maxProfit(vector<int>& prices) {
        int totalProfit = 0; // Initialize total profit to 0

        // Iterate through the price vector, starting from the second element
        for (int i = 1; i < prices.size(); ++i) {
            // Calculate the profit by buying at prices[i-1] and selling at prices[i]
            // Add the profit to totalProfit if it is positive
            totalProfit += std::max(0, prices[i] - prices[i - 1]);
        }

        // Return the total accumulated profit
        return totalProfit;
    }
};
```

TypeScript

```typescript
/**
 * Calculates the maximum profit that can be made by buying and selling stocks
 * on different days.
 *
 * @param {number[]} prices - Array of stock prices where the index represents the day.
 * @returns {number} - Maximum profit that can be made.
 */
function maxProfit(prices: number[]): number {
    let totalProfit = 0; // Initialize total profit to zero.

    // Loop through the array of prices
    for (let day = 1; day < prices.length; day++) {
        // Calculate potential profit for the day.
        const dailyProfit = prices[day] - prices[day - 1];

        // If the profit is positive, add it to the total profit.
        totalProfit += Math.max(0, dailyProfit);
    }

    return totalProfit; // Return the total profit.
}
```


```python
from itertools import pairwise  # Importing the pairwise function from itertools module

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        # Initialize total profit to zero
        total_profit = 0

        # Loop through each pair of successive prices using pairwise()
        for buy_price, sell_price in pairwise(prices):
            # Calculate profit for the current pair
            # If sell price is greater than buy price, add the difference to total profit
            # Otherwise, add zero (no loss, no gain)
            profit = max(0, sell_price - buy_price)
            total_profit += profit

        # Return the total calculated profit
        return total_profit
```


Time and Space Complexity

The time complexity of the provided code is `O(n)`, where `n` is the length of the `prices` list. This is because the code iterates through the list once with the help of the `pairwise` function, which generates a tuple for each adjacent pair of elements.

The space complexity is `O(1)` as no additional space proportional to the input size is needed. The `pairwise` function generates one pair at a time which is used in the calculation and then discarded. Therefore, the space used does not grow with the size of the input list.