121. Best Time to Buy and Sell Stock

Easy Array Dynamic Programming

Problem Description

The problem presents us with an array named `prices` that represents the price of a certain stock on each day. Our objective is to choose a single day for buying one stock and another future day for selling that stock in order to achieve the maximum possible profit. If it's not possible to make any profit, the expected return value is `0`. This problem essentially asks us to find two days, where buying on the first and selling on the second would result in the highest profit margin.

Intuition

The intuition behind the solution revolves around evaluating the difference between buying and selling prices while also making sure that we buy at the lowest price possible before selling. The strategy is to keep track of two key variables as we iterate through the `prices` array: the maximum profit found so far and the lowest price observed. We initialize the maximum profit (`ans`) as `0` and the minimum price (`mi`) as infinity to simulate that we haven't bought any stock yet.

As we iterate over each price in `prices`, we consider it as the potential selling price and calculate the profit we would get if we bought at the lowest price seen until now (`mi`). If this profit is larger than the current maximum profit (`ans`), we update `ans`. Regardless of whether it led to a new maximum profit or not, we then update `mi` if the current price is lower than any we've seen so far.

By the end of the iteration, we will have the highest possible profit that can be made with a single transaction, which is the difference between the lowest purchasing price and the highest subsequent selling price.

Solution Approach

The provided solution uses a simple yet effective approach that relies on a single pass through the `prices` array, a concept that is often referred to as the "One Pass" algorithm. This algorithm falls under the category of greedy algorithms since it makes a locally optimal choice at each step with the hope of finding a global optimum.

Here's a step-by-step walk-through of the implementation:

1. **Initialize Variables:** Two variables are initialized at the start. `ans` is initialized to `0` to represent the maximum profit which starts at no profit, and `mi` is set to `inf`, representing the minimum buying price (set to infinity since we have not encountered any price yet).

2. **Iterate Over `prices`:** We loop through each price `v` in the array `prices`, treating each one as a potential selling price.

3. **Calculate Profit and Update Maximum Profit (`ans`):** For each price `v`, if selling at this price leads to a profit higher than the best profit found so far (`ans`), the maximum profit is updated. This is done by computing the difference `v - mi` and using the `max` function: `ans = max(ans, v - mi)`.

4. **Update Minimum Buying Price (`mi`):** After checking the potential profit at price `v`, update the minimum buying price if `v` is lower than all previously seen prices: `mi = min(mi, v)`.

5. **Return Maximum Profit:** After finishing the traversal of the array `prices`, the variable `ans` holds the maximum profit that can be achieved. This value is returned as the solution to the problem.

The algorithm's efficiency hinges on the fact that the maximum profit can be found by simply keeping track of the lowest price to buy before each potential selling day. It uses constant space (only two variables), and the time complexity is linear, O(n), as it requires only one pass through the list of prices, where `n` is the number of prices in the input array.

This approach comes under dynamic programming as well since it essentially keeps a running track of state (in this case, the minimum element so far and the maximum profit so far), but it is the greedy aspect of choosing the current best action (buy/sell) that defines it more aptly.

Example Walkthrough

Let's assume we have an array of stock prices for 5 consecutive days, represented as follows: `prices = [7, 1, 5, 3, 6, 4]`.

Applying the solution approach step-by-step:

1. **Initialize Variables:** Set `ans = 0` (maximum profit starts at no profit) and `mi = inf` (minimum buying price is initially set to infinity since no prices have been seen yet).

2. **Iterate Over `prices` Day 1 (`prices[0] = 7`):** Start with the first price. Since `mi` is set to infinity, any price we encounter will be lower, so we set `mi = 7`.

3. **Day 2 (`prices[1] = 1`):** Proceed to the next price.
   - Potential profit if we sell today: `1 - mi = 1 - 7 = -6`. Since this is negative, no profit is made. We don't update `ans`.
   - Update `mi` since `1` is less than `7`: `mi = 1`.
4. **Day 3 (`prices[2] = 5`):** Move on to the third price.
   - Potential profit: `5 - mi = 5 - 1 = 4`. This is a profit, and since `ans` is `0`, it becomes the new maximum profit: `ans = 4`.
   - `mi` remains at `1`, as `5` is higher than the current `mi`.
5. **Day 4 (`prices[3] = 3`):** Evaluating the fourth price.
   - Potential profit: `3 - mi = 3 - 1 = 2`. This is less than the current `ans` of `4`, so `ans` remains unchanged.
   - Update `mi` to `3`? No, since `3` is more than the current `mi` of `1`.

Day 4 (`prices[3] = 3`) 6. **Day 5 (`prices[4] = 6`):** Analyzing the fifth price.

- Potential profit: `6 - mi = 6 - 1 = 5`. This exceeds the current `ans` of `4`, so we update `ans = 5`.
- `mi` remains unchanged as `6` is higher than `1`.

7. **Day 6 (`prices[5] = 4`):** Finally, the price on the last day.
   - Potential profit: `4 - mi = 4 - 1 = 3`. This does not beat the current `ans` of `5`, so `ans` stays at `5`.
   - `mi` does not change, as `4` is higher than `1`.
8. **Return Maximum Profit:** Having processed all days, we have found that the maximum profit is `ans = 5`, which is the highest profit achievable (by buying on day 2 at a price of `1` and selling on day 5 at a price of `6`).

The solution approach efficiently found the best day to buy and the best day to sell in a single pass through the `prices` array, confirming the maximum profit of `5` for this example. The linear time complexity (O(n)) and constant space complexity are apparent, as the algorithm only needed to iterate through the array once while using two variables.

Python Solution

```python
from typing import List

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        # Initialize the maximum profit to 0
        max_profit = 0
        # Initialize the minimum price to infinity
        min_price = float('inf')

        # Loop through the prices
        for price in prices:
            # Update the maximum profit if the current price minus the minimum price seen so far is greater
            max_profit = max(max_profit, price - min_price)
            # Update the minimum price seen so far if the current price is lower
            min_price = min(min_price, price)

        # Return the maximum profit achieved
        return max_profit

```

Java Solution

```java
class Solution {
    public int maxProfit(int[] prices) {
        // Initialize 'maxProfit' to 0, which is the minimum profit that can be made.
        int maxProfit = 0;

        // Assume the first price is the minimum buying price.
        int minPrice = prices[0];

        // Loop through all the prices to find the maximum profit.
        for (int price : prices) {
            // Calculate the maximum profit by comparing the current 'maxProfit'
            // with the difference of the current price and the 'minPrice'.
            maxProfit = Math.max(maxProfit, price - minPrice);

            // Update the 'minPrice' if a lower price is found.
            minPrice = Math.min(minPrice, price);
        }

        // Return the maximum profit that can be achieved.
        return maxProfit;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include algorithm header for the min and max functions

class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int max_profit = 0;   // Variable to store the calculated maximum profit
        int min_price = prices[0]; // Initialize the minimum price to the first price

        // Loop through all the prices
        for (int price : prices) {
            // Update max_profit if the difference between current price and min_price is greater than current max_profit
            max_profit = max(max_profit, price - min_price);

            // Update min_price if the current price is less than the current min_price
            min_price = min(min_price, price);
        }

        return max_profit; // Return the calculated maximum profit
    }
};

```

Typescript Solution

```typescript
function maxProfit(prices: number[]): number {
    // Initialize the maximum profit as 0, since the minimum profit we can get is 0
    let maxProfit = 0;
    // Initialize the minimum price to the first price in the array
    let minPrice = prices[0];

    // Loop through each price in the array of prices
    for (const price of prices) {
        // Update maxProfit to the higher value between the existing maxProfit and
        // the profit we get by selling at the current price minus the minPrice
        maxProfit = Math.max(maxProfit, price - minPrice);
        // Update minPrice to be the lower between the current minPrice and the current price
        minPrice = Math.min(minPrice, price);
    }

    // Return the maximum profit that can be achieved
    return maxProfit;
}

```

Time and Space Complexity

The time complexity of the given function is `O(n)`, where `n` is the length of the input list `prices`. This is because the function includes a single loop that iterates through each element in the list exactly once, performing a constant amount of work at each step; thus, the total work done is linear in the size of the input.

The space complexity of the function is `O(1)`, indicating that the amount of additional memory used by the function does not depend on the size of the input. The function only uses a fixed number of extra variables (`ans` and `mi`), which require a constant amount of space regardless of the input size.