309. Best Time to Buy and Sell Stock with Cooldown

Dynamic Programming Medium <u>Array</u>

Problem Description

task is to calculate the maximum profit that can be achieved through making as many buy-and-sell transactions as you wish, with a specific restriction: after selling a stock, you must wait one day before buying again. To be more explicit, you can buy one share and then sell it. However, after you sell, you need to skip a day before you can initiate

In this challenge, you are given an array prices where prices[i] represents the price of a given stock on the i-th day. Your

another purchase. This is called a cooldown period. Additionally, it is important to note that you cannot hold more than one share at a time; before you can buy another share, you must have already sold the previous one. Your goal is to find out the best strategy to maximize your profit under these terms. Intuition

The intuition behind the solution involves dynamic programming. Since we cannot engage in a new transaction on the day

decisions while considering the cooldown. We can maintain two states for each day that represent the decisions and conditions: 1. fo: The maximum profit we can have if we rest (do not buy or sell) on the current day or if we sell on the current day.

following a sale, we can make a decision for each day - to buy, sell, or rest (do nothing). The key is to track the profits of these

means subtracting the current price x from the profit made before cooldown indicated by f).

For day 0, our states would be initialized as for = 0 (we start with no stock and no cooldown) and f1 = -prices[0] (since buying the stock would cost us the price of the stock).

We need a third temporary variable f to store the previous value of fo (from the day before), which will be used to update f1

2. f1: The maximum profit we can have if we buy on the current day.

since we cannot buy the stock on the same day we sell. The <u>dynamic programming</u> transitions can be described as follows:

• fo will be the maximum of itself (choose to rest, and profit doesn't change) or f1 + x (choosing to sell the stock bought at a cost of f1, which means adding the current price \times). • f1 will be the maximum of itself (choosing to do nothing with the previously bought stock) or f - x (buying a new stock after cooldown, which

In each iteration, we go through the prices starting with the second day, because the first day's decisions are predetermined as

- for a not fixed and fixed for a second for the form of last action being a rest or a sale.
- The implementation of this solution successfully combines the conditions of the problem into a dynamic iterative process that captures the essence of buy-and-sell strategies while complying with the cooldown constraint.

The solution is implemented using a <u>dynamic programming</u> approach, which utilizes iteration and optimal substructure properties common in such problems.

o f1 is set to -prices[0], accounting for the cost of buying stock on the first day.

Solution Approach

 A temporary variable f will hold the previous value of fo during iteration, serving as a memory for the state of profits two days ago. Iteration: We begin iterating through the prices array from the second day onward (since the first day's decisions have already been set up). During each iteration, we perform two key updates to represent possible actions and their outcomes:

o fo gets updated to max(fo, f1 + x), where x is the current day's stock price. This represents the maximum profit between not

∘ f1 gets updated to max(f1, f - x), which represents the maximum profit between keeping the stock obtained before (f1) or buying

Initialization: Before the loop begins, we prepare our initial state variables:

f0 is initialized to 0, representing the profit with no stock on hand at the start.

today (which requires the previous day's profit f minus the current price x; note this action is only permitted after a cooldown). Data Structures: We use three integer variables fo, f1, and f, which change with each iteration to keep track of our

selling/buying anything on the current day (f0) and selling the stock bought at f1 price (therefore, f1 + x).

decision impacts. No additional data structures are used, making the space complexity linear.

obtainable with the last action being rest or sale, aligned with the problem requirements.

f1) are considered, with transitions based on conditions specified by the problem statement. The decision on each day depends on the state of the previous days (either one or two days back). Algorithm Completion: The loop continues to make decisions and update states based on the stock prices for each day. The

Pattern: The pattern applied here is reminiscent of state machine logic used in dynamic programming. Multiple states (for and

algorithm completes after the last day in the input array. The return value is for because it represents the maximum profit

The algorithm's overall complexity is O(n), as it involves a single pass through the prices array, and the operations within each iteration are constant time. This approach effectively balances the problem constraints and the need to make optimized decisions at each step, leading to an efficient and intuitive solution.

Let's consider a small example using the solution approach outlined above. Suppose we have the following stock prices over a

series of days: prices = [1, 2, 3, 0, 2]. Using the provided algorithm and dynamic programming approach, we'll calculate the

Day 0: Before the loop starts, let's initialize our variables. f0 is set to 0, because we haven't made any transaction yet. ∘ f1 is set to -prices[0] which is -1, because we bought one share of stock at the price of 1.

• Update f1 as max(f1, f - prices[1]) = max(-1, 0 - 2) = -1.

Day 1:

Example Walkthrough

maximum profit possible.

 At the end of Day 1, f0 is 1 and f1 is −1. Day 2: Update f to the previous value of f0 which is 1. • Calculate f0 as max(f0, f1 + prices[2]) = max(1, -1 + 3) = 2.

After iterating through all the days, our algorithm concludes and the maximum profit that we can yield from the given prices is

stored in f0, which is 4. This means the best strategy would have resulted in a total profit of 4, considering all the buy-and-sell

Day 3:

At the end of Day 2, f0 is 2 and f1 is −1.

At the end of Day 3, f0 is 2 and f1 is 2.

Solution Implementation

Initialize variables:

from typing import List

class Solution:

Example usage:

Java

C++

public:

};

TypeScript

class Solution {

sol = Solution()

class Solution {

print(profit) # Output: 5

Update f to the previous value of f0 which is 0.

• Calculate f0 as max(f0, f1 + prices[1]) = max(0, -1 + 2) = 1.

• Update f1 as max(f1, f - prices[2]) = max(-1, 1 - 3) = -1.

- Update f to the previous value of f0 which is 2. • Calculate f0 as max(f0, f1 + prices[3]) = max(2, -1 + 0) = 2.
- **Day 4**: Update f to the previous value of f0 which is 2.

• Calculate f0 as max(f0, f1 + prices[4]) = max(2, 2 + 2) = 4.

• Update f1 as max(f1, f - prices[3]) = max(-1, 2 - 0) = 2.

At the end of Day 4, f0 is 4 and f1 is 2.

• Update f1 as max(f1, f - prices[4]) = max(2, 2 - 2) = 2.

Python

freeze profit (f) - profit of the day before cooldown

freeze_profit, sell_profit, hold_profit = 0, 0, -prices[0]

Iterate through the stock prices, starting from the second day

freeze profit remains as the sell profit from the previous day

sell profit (f0) - profit after selling the stock

freeze profit, sell_profit, hold_profit = (

// Initialize the placeholders for the maximum profits

for (int i = 1; i < prices.length; <math>i++) {

Update profits for the current day

def maxProfit(self, prices: List[int]) -> int:

for current price in prices[1:]:

transactions and the compulsory cooldown period after each sale.

sell profit, max(sell profit, hold profit + current price), max(hold_profit, freeze_profit - current_price) # The maximum profit will be after all trades are done, which means no stock is being held, hence sell_profit return sell_profit

sell profit is the maximum of either keeping the previous sell profit or selling stock today (hold profit + current pri

hold profit is the max of either keeping the stock bought previously or buying new stock after cooldown (freeze_profit

// f0 represents the max profit till previous day with no stock in hand

int currentNoStock = Math.max(previousNoStock, previousWithStock + prices[i]); // Either keep no stock or sell the stock

previousWithStock = Math.max(previousWithStock, previousNoStock - prices[i]); // Either keep the stock we have or buy ne

hold profit (f1) - profit after buying the stock or holding onto the stock bought previously

tempPreviousNoStock = previousNoStock; // Temporarily store the previous no stock state

// hence we return noStockProfit which represents the max profit with no stock in hand

// prevProfitWithStock represents the profit having stock at the previous day's end.

// Update the profit status by choosing the best profit strategy for the day:

prevNoStockProfit = Math.max(prevNoStockProfit, profitWithStock + price);

// Keep no stock or sell the stock (prevProfitWithStock + price),

profitWithStock = Math.max(profitWithStock, noStockProfit - price);

Iterate through the stock prices, starting from the second day

// Keep the stock or buy new stock (prevNoStockProfit - price).

// Return the maximum profit status when ending with no stock.

previousNoStock = currentNoStock; // Update the previous no stock state with the current state

return previousNoStock; // At the end, the profit with no stock in hand will be the maximum profit

int previousNoStock = 0; int previousWithStock = -prices[0]; // f1 represents the max profit till previous day with stock in hand int tempPreviousNoStock; // Used to store previous no stock state temporarily // Parser through the price list starting from day 1 as we have initial state for day 0 already considered

return noStockProfit;

let noStockProfit: number = 0;

return prevNoStockProfit;

from typing import List

let prevNoStockProfit: number = 0;

function maxProfit(prices: number[]): number {

let profitWithStock: number = -prices[0];

noStockProfit = tempNoStockProfit;

def maxProfit(self, prices: List[int]) -> int:

for current price in prices[1:]:

of stock prices, where prices is a list of stock prices.

for (const price of prices.slice(1)) {

// Initialize the first day's profit status variables;

// fl represents the profit having stock at day's end,

// Iterate over the prices from the second day onward.

const tempNoStockProfit = prevNoStockProfit;

// f0 represents the profit having no stock at day's end,

profit = sol.maxProfit([7,1,5,3,6,4])

public int maxProfit(int[] prices) {

```
int maxProfit(vector<int>& prices) {
    // Initialize the profit states:
    // f0: the max profit we can have at this state if we don't hold a stock
    // fl: the max profit we can have at this state if we hold one stock
    // holdProfit: stores the previous f0 state to calculate the new f1 state
    int noStockProfit = 0;
    int holdProfit = 0;
    int oneStockProfit = -prices[0]; // Assume we bought the first stock
    // Loop through the list of prices starting from the second price
    for (int i = 1; i < prices.size(); ++i) {</pre>
       // Calculate the max profit if we don't hold a stock today
       // either we did not hold a stock yesterday (noStockProfit) or
        // we sold the stock we were holding (oneStockProfit + prices[i]).
        int newNoStockProfit = max(noStockProfit, oneStockProfit + prices[i]);
       // Calculate the max profit if we hold one stock today
       // either we were already holding a stock (oneStockProfit) or
        // we buy a new stock today (holdProfit - prices[i]).
        oneStockProfit = max(oneStockProfit, holdProfit - prices[i]);
        // Update holdProfit to the previous noStockProfit at the end of the day
        holdProfit = noStockProfit;
        // Update noStockProfit to the new calculated noStockProfit
        noStockProfit = newNoStockProfit;
   // Since we want to maximize profit, we should not hold any stock at the end
```

```
# Initialize variables:
# freeze profit (f) - profit of the day before cooldown
# sell profit (f0) - profit after selling the stock
# hold profit (f1) - profit after buying the stock or holding onto the stock bought previously
freeze_profit, sell_profit, hold_profit = 0, 0, -prices[0]
```

class Solution:

```
# Update profits for the current day
           # freeze profit remains as the sell profit from the previous day
            # sell profit is the maximum of either keeping the previous sell profit or selling stock today (hold profit + current pri
           # hold profit is the max of either keeping the stock bought previously or buying new stock after cooldown (freeze profit
            freeze profit. sell_profit, hold_profit = (
               sell profit,
               max(sell profit, hold profit + current price),
               max(hold profit, freeze profit - current price)
       # The maximum profit will be after all trades are done, which means no stock is being held, hence sell_profit
       return sell_profit
# Example usage:
 sol = Solution()
# profit = sol.maxProfit([7,1,5,3,6,4])
# print(profit) # Output: 5
Time and Space Complexity
```

The time complexity of the code is driven by a single loop that iterates through the list of stock prices once (excluding the first price which is used for initial setup). Inside the loop, the code performs a fixed number of comparisons and arithmetic operations

for each price. Because these operations occur within the loop and their number does not depend on the size of prices, they

The given Python code defines a method maxProfit designed to find the maximum profit that can be achieved from a sequence

constitute constant-time operations. Therefore, the time complexity is determined solely by the number of iterations, which is n - 1, where n is the length of prices.

Time Complexity

Considering big O notation, which focuses on the upper limit of performance as the input size grows, the time complexity of this algorithm is O(n). **Space Complexity**

The space complexity is assessed based on the additional memory used by the algorithm as a function of the input size. In this

code, only a fixed number of variables f, f0, and f1 are used, regardless of the size of the input list prices. There is no use of any additional data structures that would grow with the input size. Thus, the space complexity of the algorithm is 0(1), denoting constant space usage.