Previous Next

309. Best Time to Buy and Sell Stock with Cooldown

April 28, 2020 | 5.2K views

Average Rating: 5 (31 votes)

Say you have an array for which the i^{th} element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times) with the following restrictions:

- You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).
- After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)

Example:

```
Input: [1,2,3,0,2]
Output: 3
Explanation: transactions = [buy, sell, cooldown, buy, sell]
```

Solution

Overview

First of all, we would like to mention that this is yet another problem from the series of Best-Time-to-Buy-and-Sell-Stock problems, which we list as follows:

- Best Time to Buy and Sell Stock
- Best Time to Buy and Sell Stock II
- Best Time to Buy and Sell Stock III
- Best Time to Buy and Sell Stock IV

One could try to resolve them one by one, which certainly could help with this problem.

There have been quite some excellent posts in the Discussion forum. We would like to mention that the user fun4LeetCode even developed a mathematical representation that is able to be generalized to each of the problems.

That being said, here we contribute some approaches, which hopefully could provide you different perspectives for the problem.

As one might have seen the hint from the problem description, which says "dynamic programming" (*i.e.* DP), we could tackle this problem mainly with the technique called **dynamic programming**.

Often the case, in order to come up with a dynamic programming solution, it would be beneficial to draw down some mathematical formulas to model the problem.

> As a reminder, the nature of dynamic programming is to break the original problem into several subproblems, and then reuse the results of subproblems for the original problem.

Therefore, due to the nature of DP, the mathematical formulas that we should come up with would almost certainly assume the form of ***recursion***.

Before embarking on the next sections of this article, we kindly ask the audiences to keep an open mind, fasten your seat belts and enjoy the ride with a heavy (yet healthy) dose of mathematical formulas.

Approach 1: Dynamic Programming with State Machine

Intuition

First of all, let us take a different perspective to look at the problem, unlike the other algorithmic problems.

Here, we will treat the problem as a game, and the trader as an agent in the game. The agent can take actions that lead to gain or lose of game points (*i.e.* profits). And the goal of the game for the agent is to gain the maximal points.

In addition, we will introduce a tool called state machine, which is a mathematical model of computation. Later one will see how the state machine coupled with the dynamic programming technique can help us solve the problem easily.

In the following sections, we will first define a ***state machine*** that is used to model the behaviors and states of the game agent.

Then, we will demonstrate how to apply the state machine to solve the problem.

Definition

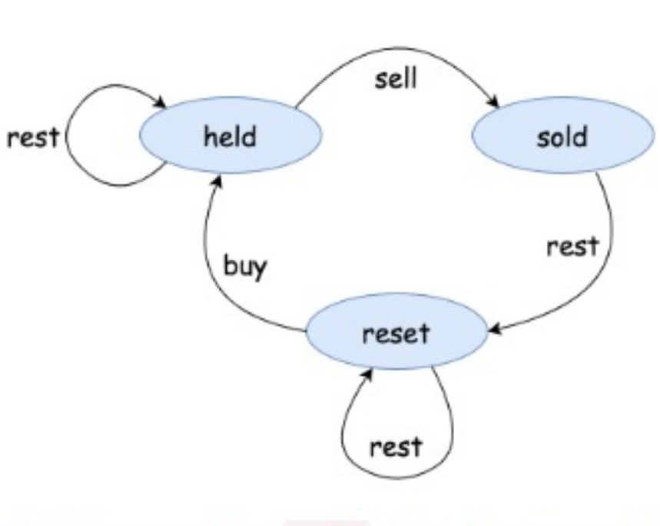
Let us define a ***state machine*** to model our agent. The state machine consists of three states, which we define as follows:

- state `held`: in this state, the agent holds a stock that it bought at some point before.
- state `sold`: in this state, the agent has just sold a stock right before entering this state. And the agent holds no stock at hand.
- state `reset`: first of all, one can consider this state as the starting point, where the agent holds no stock and did not sell a stock before. More importantly, it is also the *transient* state before the `held` and `sold`. Due to the ***cooldown*** rule, after the `sold` state, the agent can not immediately acquire any stock, but is *forced* into the `reset` state. One can consider this state as a "reset" button for the cycles of buy and sell transactions.

At any moment, the agent can only be in ***one*** state. The agent would transition to another state by performing some actions, namely:

- action `sell`: the agent sells a stock at the current moment. After this action, the agent would transition to the `sold` state.
- action `buy`: the agent acquires a stock at the current moment. After this action, the agent would transition to the `held` state.
- action `rest`: this is the action that the agent does no transaction, neither buy or sell. For instance, while holding a stock at the `held` state, the agent might simply do nothing, and at the next moment the agent would remain in the `held` state.

Now, we can assemble the above states and actions into a **state machine**, which we show in the following graph where each node represents a state, and each edge represents a transition between two states. On top of each edge, we indicate the action that triggers the transition.

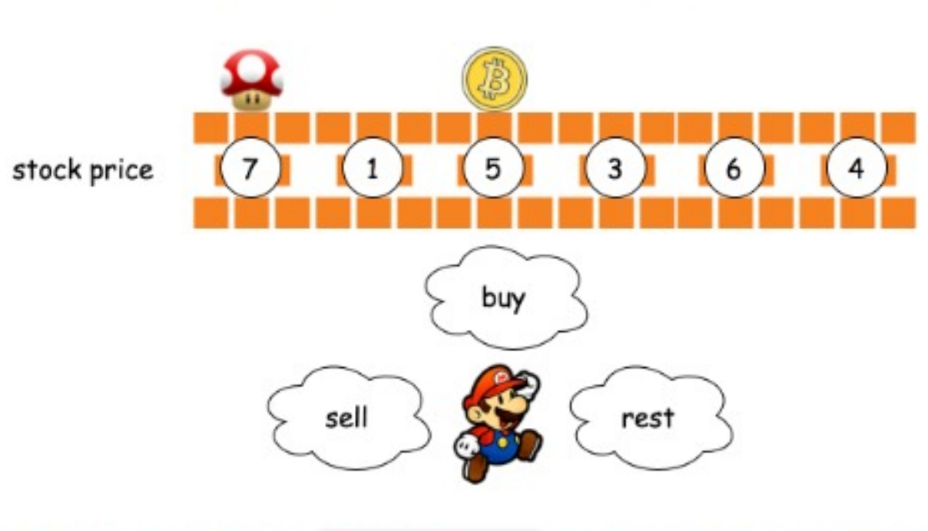


Notice that, in all states except the `sold` state, by doing nothing, we would remain in the same state, which is why there is a self-looped transition on these states.

Deduction

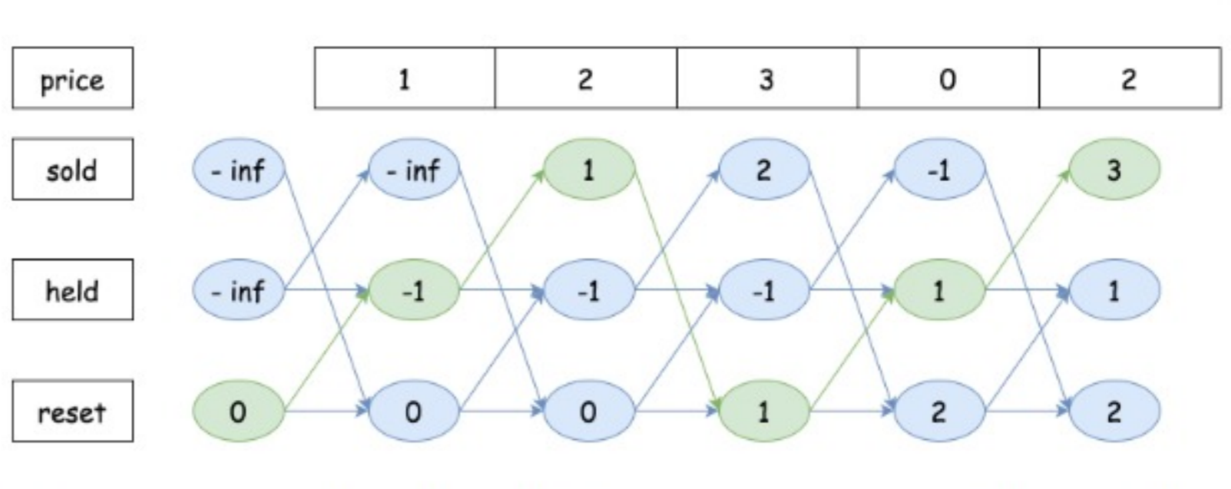
Now, one might wonder how exactly the state machine that we defined can help to solve the problem.

As we mentioned before, we model the problem as a ***game***, and the trader as an ***agent*** in the game. And this is where our state machine comes into the picture. The behaviors and the states of the game agent can be modeled by our state machine.



Given a list stock prices (*i.e.* `price[0...n]`), our agent would walk through each price point one by one. At each point, the agent would be in one of three states (*i.e.* `held`, `sold` and `reset`) that we defined before. And at each point, the agent would take one of the three actions (*i.e.* `buy`, `sell` and `rest`), which then would lead to the next state at the next price point.

> Now if we chain up each state at each price point, it would form a ***graph*** where each ***path*** that starts from the initial price point and ends at the last price point represents a combination of transactions that the agent could perform through out the game.



The above graph shows all possible paths that our game agent agent walks through the list, which corresponds to all possible combinations of transactions that the trader can perform with the given price sequence.

> In order to solve the problem, the goal is to find such a path in the above graph that maximizes the profits.

In each node of graph, we also indicate the maximal profits that the agent has gained so far in each state of each step. And we highlight the path that generates the maximal profits. Don't worry about them for the moment. We will explain in detail how to calculate in the next section.

Algorithm

In order to implement the above state machine, we could define three arrays (*i.e.* `held[i]`, `sold[i]` and `reset[i]`) which correspond to the three states that we defined before.

> Each element in each array represents the maximal profits that we could gain at the specific price point `i` with the specific state. For instance, the element `sold[2]` represents the maximal profits we gain if we sell the stock at the price point `price[2]`.

According to the state machine we defined before, we can then deduce the formulas to calculate the values for the state arrays, as follows:

$$\begin{aligned} \text{sold}[i] &= \text{hold}[i-1] + \text{price}[i] \\ \text{held}[i] &= \max\left(\text{held}[i-1], \quad \text{reset}[i-1] - \text{price}[i]\right) \\ \text{reset}[i] &= \max\left(\text{reset}[i-1], \quad \text{sold}[i-1]\right) \end{aligned}$$

Here is how we interpret each formulas:

- $\text{sold}[i]$: the previous state of `sold` can only be `held`. Therefore, the maximal profits of this state is the maximal profits of the previous state plus the revenue by selling the stock at the current price.
- $\text{held}[i]$: the previous state of `held` could also be `held`, *i.e.* one does no transaction. Or its previous state could be `reset`, from which state, one can acquire a stock at the current price point.
- $\text{reset}[i]$: the previous state of `reset` could either be `reset` or `sold`. Both transitions do not involve any transaction with the stock.

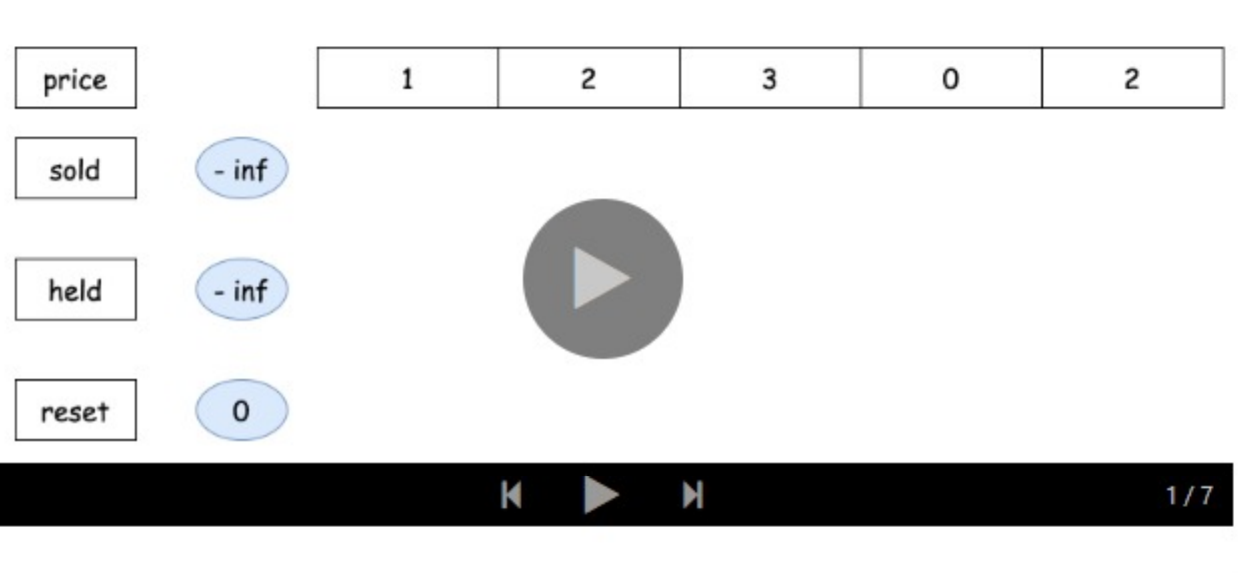
> Finally, the maximal profits that we can gain from this game would be $\max\left(\text{sold}[n], \text{reset}[n]\right)$, *i.e.* at the last price point, either we sell the stock or we simply do no transaction, to have the maximal profits. It makes no sense to acquire the stock at the last price point, which only leads to the reduction of profits.

In particular, as a base case, the game should be kicked off from the state `reset`, since initially we don't hold any stock and we don't have any stock to sell neither. Therefore, we assign the initial values of `sold[-1]` and `held[-1]` to be `Integer.MIN_VALUE`, which are intended to *render* the paths that start from these two states impossible.

As one might notice in the above formulas, in order to calculate the value for each array, we reuse the intermediate values, and this is where the paradigm of ***dynamic programming*** comes into play.

More specifically, we only need the intermediate values at exactly one step before the current step. As a result, rather than keeping all the values in the three arrays, we could use a ***sliding window*** of size `1` to calculate the value for $\max\left(\text{sold}[n], \text{reset}[n]\right)$.

In the following animation, we demonstrate the process on how the three arrays are calculated step by step.



> As a ***byproduct*** of this algorithm, not only would we obtain the maximal profits at the end, but also we could recover each action that we should perform along the path, although this is not required by the problem.

In the above graph, by starting from the final state, and walking backward following the path, we could obtain a sequence of actions that leads to the maximal profits at the end, *i.e.* [`buy`, `sell`, `cooldown`, `buy`, `sell`].

Java Python Copy

```python
class Solution(object):
    def maxProfit(self, prices):
        """
        :type prices: List[int]
        :rtype: int
        """
        sold, held, reset = float('-inf'), float('-inf'), 0

        for price in prices:
            # Alternative: the calculation is done in parallel.
            # Therefore no need to keep temporary variables
            #sold, held, reset = held + price, max(held, reset-price), max(reset, sold)

            pre_sold = sold
            sold = held + price
            held = max(held, reset - price)
            reset = max(reset, pre_sold)

        return max(sold, reset)
```

Complexity Analysis

- Time Complexity: $\mathcal{O}(N)$ where N is the length of the input price list.
  - We have one loop over the input list, and the operation within one iteration takes constant time.
- Space Complexity: $\mathcal{O}(1)$, constant memory is used regardless the size of the input.

Approach 2: Yet-Another Dynamic Programming

Intuition

Most of the times, there are more than one approaches to decompose the problem, so that we could apply the technique of dynamic programming.

Here we would like to propose a different perspective on how to model the problem purely with mathematical formulas.

Again, this would be a journey loaded with mathematical notations, which might be complicated, but it showcases how the mathematics could help one with the dynamic *programming* (pun intended).

Definition

For a sequence of prices, denoted as $\text{price}[0, 1, ..., n]$, let us first define our **target** function called $\text{MP}(i)$. The function $\text{MP}(i)$ gives the maximal profits that we can gain for the price *subsequence* starting from the index i, *i.e.* $\text{price}[i, i+1, ..., n]$.

Given the definition of the $\text{MP}(i)$ function, one can see that when $i = 0$ the output of the function, *i.e.* $\text{MP}(0)$, is exactly the result that we need to solve the problem, which is the maximal profits that one can gain for the price subsequence of $\text{price}[0, 1, ..., n]$.

Suppose that we know all the values for $\text{MP}(i)$ onwards until $\text{MP}(n)$, *i.e.* we know the maximal profits that we can gain for any subsequence of $\text{price}[k...n] \quad k \in [i, n]$.

Now, let us add a new price point $\text{price}[i-1]$ into the subsequence $\text{price}[i...n]$, all we need to do is to deduce the value for the **unknown** $\text{MP}(i-1)$.

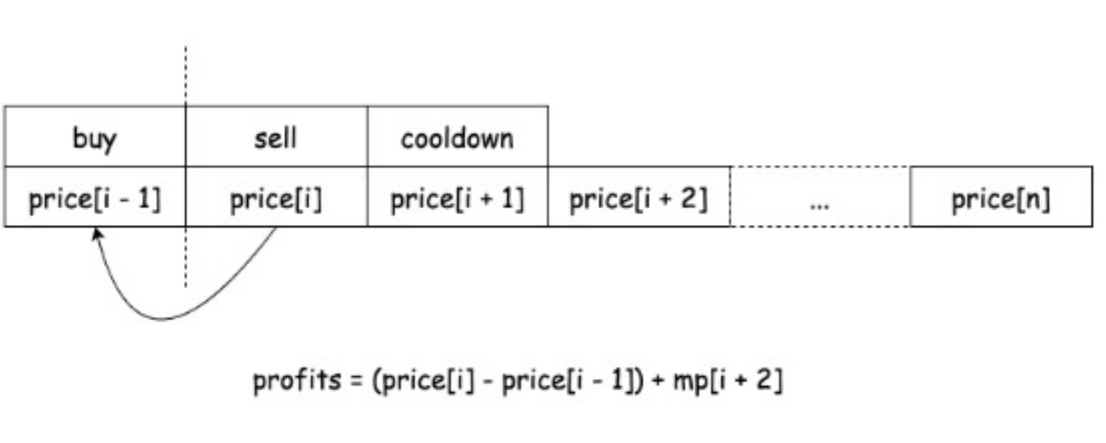
> Up to this point, we have just modeled the problem with our **target** function $\text{MP}(i)$, along with a series of definitions. The problem now is boiled down to deducing the formula for $\text{MP}(i-1)$.

In the following section, we will demonstrate how to deduce the formula for $\text{MP}(i-1)$.

Deduction

With the newly-added price point $\text{price}[i-1]$, we need to consider **all** possible transactions that we can do to the stock at this price point, which can be broken down into two cases:

- Case 1): we buy this stock with $\text{price}[i-1]$ and then sell it at some point in the following price sequence of $\text{price}[i...n]$. Note that, once we sell the stock at a certain point, we need to cool down for a day, then we can reengage with further transactions. Suppose that we sell the stock right after we bought it, at the next price point $\text{price}[i]$, the maximal profits we would gain from this choice would be the profit of this transaction (*i.e.* $\text{price}[i] - \text{price}[i-1]$) **plus** the maximal profits from the rest of the price sequence, as we show in the following:



profits = (price[i] - price[i - 1]) + mp[i + 2]

In addition, we need to **enumerate** all possible points to sell this stock, and take the maximum among them. The maximal profits that we could gain from this case can be represented by the following:

$$C_1 = \max_{\{k \in [i,n]\}}\left(\text{price}[k] - \text{p}[i-1] + \text{MP}(k+2)\right)$$

- Case 2): we simply do nothing with this stock. Then the maximal profits that we can gain from this case would be $\text{MP}(i)$, which are also the maximal profits that we can gain from the rest of the price sequence.

$$C_2 = \text{MP}(i)$$

By combining the above two cases, *i.e.* selecting the max value among them, we can obtain the value for $\text{MP}(i-1)$, as follows:

$$\text{MP}(i-1) = \max(C_1, C_2)$$

$$\text{MP}(i-1) = \max\Big(\max_{\{k \in [i,n]\}}\left(\text{price}[k] - \text{price}[i-1] + \text{MP}(k+2)\right), \quad \text{MP}(i)\Big)$$

By the way, the base case for our recursive function $\text{MP}(i)$ would be $\text{MP}(n)$ which is the maximal profits that we can gain from the sequence with a single price point $\text{price}[n]$. And the best thing we should do with a single price point is to do no transaction, hence we would neither lose money nor gain any profit, *i.e.* $\text{MP}(n) = 0$.

The above formulas do model the problem soundly. In addition, one should be able to translate them directly into code.

Algorithm

With the final formula we derived for our target function $\text{MP}(i)$, we can now go ahead and translate it into any programming language.

- Since the formula deals with subsequences of price that start from the last price point, we then could do an **iteration** over the price list in the reversed order.
- We define an array `MP[i]` to hold the values for our target function $\text{MP}(i)$. We initialize the array with zeros, which correspond to the base case where the minimal profits that we can gain is zero. Note that, here we did a trick to pad the array with two additional elements, which is intended to simplify the branching conditions, as one will see later.
- To calculate the value for each element `MP[i]`, we need to look into two cases as we discussed in the previous section, namely:
  - Case 1). we buy the stock at the price point `price[i]`, then we sell it at a later point. As one might notice, the initial padding on the `MP[i]` array saves us from getting out of boundary in the array.
  - Case 2). we do no transaction with the stock at the price point `price[i]`.
- At the end of each iteration, we then pick the largest value from the above two cases as the final value for `MP[i]`.
- At the end of the loop, the `MP[i]` array will be populated. We then return the value of `MP[0]`, which is the desired solution for the problem.

Java Python Copy

```python
class Solution(object):
    def maxProfit(self, prices):
        """
        :type prices: List[int]
        :rtype: int
        """
        L = len(prices)
        # padding the array with additional zero to simply the logic
        MP = [0] * (L + 2)

        for i in range(L-1, -1, -1):
            C1 = 0
            # Case 1). buy and sell the stock
            for sell in range(i + 1, L):
                profit = (prices[sell] - prices[i]) + MP[sell + 2]
                C1 = max(profit, C1)

            # Case 2). do no transaction with the stock p[i]
            C2 = MP[i + 1]

            # sum up two cases
            MP[i] = max(C1, C2)

        return MP[0]
```

Complexity Analysis

- Time Complexity: $\mathcal{O}(N^2)$ where N is the length of the price list.
  - As one can see, we have nested loops over the price list. The number of iterations in the outer loop is N. The number of iterations in the inner loop varies from 1 to N. Therefore, the total number of iterations that we perform is $\sum_{i=1}^{N} i = \frac{N \cdot (N+1)}{2}$.
  - As a result, the overall time complexity of the algorithm is $\mathcal{O}(N^2)$.
- Space Complexity: $\mathcal{O}(N)$ where N is the length of the price list.
  - We allocated an array to hold all the values for our target function $\text{MP}(i)$.

Rate this article: ★★★★★

Previous Next

Comments: 5 Sort By

Type comment here... (Markdown is supported)

Preview Post

Frankxiao08 ★ 4 May 5, 2020 9:29 AM

The explanation is quite clear! Great Job! At first, I don't want to read it because it's so long, but once I start to read, it takes only about 5 minutes for me finish solution 1. 3 stages is brilliant!

4 Share Reply

Hoffles ★ 4 June 14, 2020 3:28 PM

Excellent state machine explanation. Perhaps it would be helpful if the first qn in the series (Best Time to Buy and Sell Stocks) could have this type of explanation in the solutions. So when people progress through this qn series they can build on their state machine knowledge.

1 Share Reply

rabldong ★ 14 May 6, 2020 11:33 PM

I got pretty confused at first because you are representing actions as state but
the graph made it clear

1 Share Reply

sasawq21 ★ 0 3 days ago

Can someone explain why held[0] and sold[0] are INT_MIN and reset[0] is 0 ?

0 Share Reply

chubing465 ★ 0 July 8, 2020 10:44 PM

`programming (pun intended)`
What's the pun here?

0 Share Reply

SHOW 1 REPLY