My Unique Solution to Leetcode #123 With Intuition and Proof

Jacob Goldberg

December 2025

Contents

1 Introduction

While practicing some Leetcode, I came across one of my favorite problems: 123. Best Time to Buy and Sell Stock III. This problem has an elegant DP solution with time complexity $\mathcal{O}(n)$; however, when I originally solved this problem, I came up with a different approach, also with $\mathcal{O}(n)$ time complexity, that I thought I would share in detail here. Before I spoil the fun, I recommend you attempt the problem yourself.

1.1 The Problem

You are given an array `prices` where `prices[i]` is the price of a given stock on the `ith` day.

Find the maximum profit you can achieve. You may complete **at most two transactions**.

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        ...
```

1.1.1 Examples

Example 1:
Input: prices = [3,3,5,0,0,3,1,4]
Output: 6
Explanation: Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3. Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = 4-1 = 3.

Example 2:
Input: prices = [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.
Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are engaging multiple transactions at the same time. You must sell before buying again.

Example 3:
Input: prices = [7,6,4,3,1]
Output: 0
Explanation: In this case, no transaction is done, i.e. max profit = 0.

1.1.2 Constraints

- 1 <= prices.length <= 10^5
- 0 <= prices[i] <= 10^5

2 My Solution

2.1 The Solution

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        best_return, best_buy, best_sell = maxProfitHelper(prices)
        before, after = prices[: best_buy], prices[best_sell + 1:]
        during = prices[best_buy + 1: best_sell]
        during.reverse()

        best_before = maxProfitHelper(before)[0]
        best_after = maxProfitHelper(after)[0]
        best_during = maxProfitHelper(during)[0]

        return max(best_before, best_after, best_during) + best_return


def maxProfitHelper(prices):
    if len(prices) <= 1:
        return (0, 0, 0)
    max_profit = (0, 0, 0)
    buy, max_sell = (prices[0], 0), (prices[0], 0)

    for i in range(1, len(prices)):
        price = prices[i]
        if price < buy[0]:
            profit = (max_sell[0] - buy[0], buy[1], max_sell[1])
            max_profit = max(max_profit, profit)
            buy, max_sell = (price, i), (price, i)
        max_sell = max(max_sell, (price, i))

    profit = (max_sell[0] - buy[0], buy[1], max_sell[1])
    max_profit = max(max_profit, profit)
    return max_profit
```

2.2 The Intuition

To understand this strategy, first ignore the “two transactions” part altogether.

If you were only allowed to make *one* transaction, the goal would simply be to buy at the lowest price seen so far and sell at the highest price afterward. There is a single buy–sell interval that achieves the maximum possible profit, and no other individual transaction can beat it. This transaction becomes the natural backbone of the solution. (This is the same strategy behind the ”easy” Leetcode #121: Best Time to Buy and Sell Stock).

Now suppose we are allowed a second transaction. Once the best single transaction is fixed, there are only three meaningful places where another transaction could live:

- completely before the best transaction,
- completely after the best transaction, or
- entirely inside the interval of the best transaction.

That's it. Any transaction that partially overlaps with the best one is not actually doing anything new—it can always be shifted so that it starts at the same buy time as the optimal transaction without losing profit. In other words, overlap never helps; it just disguises one of the three cases above.

This observation is what drives the algorithm. We first find the single best transaction and record its buy and sell indices. Then we independently compute the best profit achievable in each of the three regions: before, after, and inside that interval. When we search inside the interval we reverse the array because we are first selling from the first transaction then buying for the second, so we want to find the biggest dip (which is the biggest gain of the reversed array). This gain found in the reversed array is the extra profit squeezed out, by selling right before this dip and buying the dip within the best transaction interval, on top of the profit from the best transaction. The best valid two-transaction solution must consist of the optimal transaction profit plus the profit from the best option among those three.

Each step is done in linear time. Finding the best single transaction is a single pass through the array. Each of the three follow-up computations is also a single pass over a disjoint subarray. As a result, we loop over the entire prices array twice and the overall time complexity remains $\mathcal{O}(n)$, with constant extra space.

The proof below formalizes this intuition and shows that no valid solution can fall outside these cases.

2.3 The Proof

2.3.1 Definitions

Definition 1. A *transaction pair* is an ordered pair $T = (b, s)$ with $0 \leq b \leq s < n$, yielding profit

$$|T| = \text{prices}[s] - \text{prices}[b].$$

Definition 2. Let A be the set of all possible transaction pairs in prices. We define the *optimal transaction pair*, T^*, such that $\forall T \in A$: $|T^*| \geq |T|$.

Definition 3. A transaction pair X *overlaps* transaction pair Y if $b_X < b_Y \leq s_X \leq s_Y$ or $b_Y \leq b_X \leq s_Y < s_X$.

Definition 4. Define the following quantities:

$$\begin{aligned} P_{\text{before}} &= \max_{0\leq b\leq s<b^*} \big(\text{prices}[s] - \text{prices}[b]\big), \\ P_{\text{after}} &= \max_{s^*<b\leq s<n} \big(\text{prices}[s] - \text{prices}[b]\big), \\ P_{\text{during}} &= \max_{b^*\leq b\leq s\leq s^*} \big(\text{prices}[b] - \text{prices}[s]\big). \end{aligned}$$

2.3.2 Key Lemmas

Lemma 1. *Let $T^* = (b^*, s^*)$ be the optimal transaction. Then*

$$\mathit{prices}[b^*] = \min_{0\leq i\leq s^*} \mathit{prices}[i].$$

Proof. Assume for contradiction that there exists an index $i \leq s^*$ such that

$$\text{prices}[i] < \text{prices}[b^*].$$

Then the transaction (i, s^*) yields profit

$$\text{prices}[s^*] - \text{prices}[i] > \text{prices}[s^*] - \text{prices}[b^*] = |T^*|,$$

contradicting the optimality of T^*. Hence, no such index i exists, and b^* attains the minimum price on $[0, s^*]$. □

2.3.3 Main Theorem

Theorem 1. *The algorithm returns the maximum achievable profit, $P_{algorithm} = |T^*| + max(P_{before}, P_{after}, P_{during})$, using at most two transactions.*

Proof. Assume for contradiction that there exists a feasible solution, $\mathcal{S}$, using at most two transactions whose total profit, $P_{\mathcal{S}}$, is strictly greater than the profit returned by the algorithm.

Case 1: $\mathcal{S}$ consists of a single transaction, T_1.

We know that:

$$\begin{aligned} P_{\mathcal{S}} &= |T_1| \\ &\leq |T^*| \\ &\leq |T^*| + max(P_{\text{before}}, P_{\text{after}}, P_{\text{during}}) \\ &= P_{\text{algorithm}} \end{aligned}$$

meaning $P_{\mathcal{S}} \leq P_{\text{algorithm}}$. We have reached a contradiction since $P_{\mathcal{S}} \not> P_{\text{algorithm}}$.

Case 2: $\mathcal{S}$ consists of two non-overlapping transactions, $\{T_1, T_2\}$, where $0 \leq b_1 \leq s_1 \leq b_2 \leq s_2 < n$.

Here we have two sub-cases:

Case 2.a: Neither transaction in $\mathcal{S}$ overlaps T^*.

Here we have two sub-cases:

Case 2.a.i: $b^* \le b_1 \le s_2 \le s^*$, meaning both transactions happen within the optimal transaction range.

We see that:

$$\text{prices}[s_1] - \text{prices}[b_2] \le P_{\text{during}}, \text{prices}[s_2] - \text{prices}[b_1] \le |T^*|.$$

We then see that:

$$\begin{aligned} P_{\mathcal{S}} &= |T_1| + |T_2| \\ &= \text{prices}[s_1] - \text{prices}[b_1] + \text{prices}[s_2] - \text{prices}[b_2] \\ &\le |T^*| + P_{\text{during}} \\ &\le |T^*| + max(P_{\text{before}}, P_{\text{after}}, P_{\text{during}}) \\ &= P_{\text{algorithm}} \end{aligned}$$

meaning $P_{\mathcal{S}} \le P_{\text{algorithm}}$. We have reached a contradiction since $P_{\mathcal{S}} \not> P_{\text{algorithm}}$.

Case 2.a.ii: There exists some $i \in \{1, 2\}$ such that $(s_i < b^*) \vee (b_i > s^*)$, meaning at least one transaction happens outside the optimal transaction range.

Without loss of generality, T_1 happens outside the optimal transaction range. We see that:

$$|T_1| \le max(P_{\text{before}}, P_{\text{after}}), |T_2| \le |T^*|$$

We then see that:

$$\begin{aligned} P_{\mathcal{S}} &= |T_1| + |T_2| \\ &\le max(P_{\text{before}}, P_{\text{after}}) + |T^*| \\ &\le |T^*| + max(P_{\text{before}}, P_{\text{after}}, P_{\text{during}}) \\ &= P_{\text{algorithm}} \end{aligned}$$

meaning $P_{\mathcal{S}} \le P_{\text{algorithm}}$. We have reached a contradiction since $P_{\mathcal{S}} \not> P_{\text{algorithm}}$.

Thus, Case 2.a cannot occur.

Case 2.b: At least one transaction in $\mathcal{S}$ overlaps T^*.

Without loss of generality, say T_1 overlaps T^* and $b_1 < b^* \le s_1 \le s^*$. Define a new transaction $T_1' = (b^*, s_1)$, and let $\mathcal{S}' = \{T_1', T_2\}$. We claim that $\mathcal{S}'$ is a feasible solution whose total profit is at least that of $\mathcal{S}$.

By Lemma 1, $\text{prices}[b^*] \le \text{prices}[b_1]$ since $b_1 < b^* \le s_1 \le s^*$. Therefore,

$$\begin{aligned} |T_1'| &= \text{prices}[s_1] - \text{prices}[b^*] \\ &\ge \text{prices}[s_1] - \text{prices}[b_1] \\ &= |T_1| \end{aligned}$$

and so replacing T_1 with $T_1' = (b^*, s_1)$ does not decrease profit.

Second, T_1' lies entirely within $[b^*, s^*]$, and therefore does not overlap T^*. Moreover, since $\mathcal{S}$ was a feasible and $b_1 < b^* \leq s_1$, shifting the buy time of T_1 forward preserves non-overlap with T_2 and preserves feasibility. Hence $\mathcal{S}'$ consists of two non-overlapping transactions.

Therefore, $\mathcal{S}'$ satisfies the conditions of Case 2.a, meaning

$$P_{\mathcal{S}} \leq P_{\mathcal{S}'} \leq P_{\text{algorithm}},$$

yielding a contradiction since $P_{\mathcal{S}} \not> P_{\text{algorithm}}$.

Thus, Case 2.b cannot occur.

Therefore, since Case 1 and Case 2 cannot occur, no such solution $\mathcal{S}$ exists, and the algorithm is correct. $\square$

3 The Standard DP Solution

3.1 The Solution

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        first_profit, second_profit = 0, 0
        first_buy, second_buy = float("inf"), float("-inf")

        for price in prices:
            first_buy = min(price, first_buy)
            first_profit = max(first_profit, price - first_buy)

            second_buy = max(second_buy, first_profit - price)
            second_profit = max(second_profit, second_buy + price)
        return second_profit
```

3.2 The Intuition

The intuition is very similar to the one transaction version; however, instead of just keeping track of the maximal profit from one transaction we also keep track of the most profit we could get from a second buy. The best potential gain of using your second buy at each index in the array is the maximal profit we have seen so far, calculated in the same way by buying at the lowest price seen so far and selling at the highest price thereafter, minus the price at the current index. We maximize this potential gain from a second buy as we iterate through prices. The maximal profit from at most two transactions is simply the potential gain from a second buy plus the highest price at which we can sell it thereafter.

This solution is very elegant; we loop over the entire prices array just once, resulting in an overall time complexity of $\mathcal{O}(n)$ with constant extra space.

3.3 The Proof

3.3.1 Definitions

Definition 5. For each index $i \in \{0, \dots, n-1\}$, define the following optimal values:

$$P_1(i) = \max_{0 \le b \le s \le i} \big(\text{prices}[s] - \text{prices}[b]\big),$$
$$P_2(i) = \max_{0 \le b_1 \le s_1 < b_2 \le s_2 \le i} \Big((\text{prices}[s_1] - \text{prices}[b_1]) + (\text{prices}[s_2] - \text{prices}[b_2])\Big),$$

with the convention that the maximum over an empty set is 0.

Definition 6. Define auxiliary quantities

$$B_1(i) = \min_{0 \le j \le i} \text{prices}[j],$$
$$B_2(i) = \max_{0 \le j \le i} \big(P_1(j) - \text{prices}[j]\big).$$

3.3.2 Key Lemmas

Lemma 2. *For all i, the value $P_1(i)$ satisfies*

$$P_1(i) = \max\big(P_1(i-1),\ \text{prices}[i] - B_1(i)\big).$$

Proof. Any optimal one-transaction strategy over $[0, i]$ either sells on day i or sells earlier. If it sells earlier, its profit is $P_1(i-1)$. If it sells on day i, optimality requires buying at the minimum price in $[0, i]$, which is $B_1(i)$. Taking the maximum of these two cases yields the result. □

Lemma 3. *For all i, the value $P_2(i)$ satisfies*

$$P_2(i) = \max\big(P_2(i-1),\ B_2(i) + \text{prices}[i]\big).$$

Proof. Any optimal two-transaction strategy over $[0, i]$ either sells its second transaction strictly before i, yielding profit $P_2(i-1)$, or sells its second transaction on day i.

In the latter case, suppose the second transaction buys on day $j < i$ and sells on day i. The total profit is

$$P_1(j) + (\text{prices}[i] - \text{prices}[j]) = (P_1(j) - \text{prices}[j]) + \text{prices}[i].$$

Maximizing over all $j \le i$ yields $B_2(i) + \text{prices}[i]$. Taking the maximum of the two cases gives the result. □

3.3.3 Main Theorem

Theorem 2. *The algorithm returns the maximum achievable profit using at most two non-overlapping transactions.*

Proof. We prove by induction on i that after processing index i, the algorithm maintains

$$\texttt{first\_profit} = P_1(i) \quad \text{and} \quad \texttt{second\_profit} = P_2(i).$$

Base Case ($i = 0$). No transaction is possible using only day 0, so by definition

$$P_1(0) = P_2(0) = 0.$$

The algorithm initializes

$$\texttt{first\_profit} = \texttt{second\_profit} = 0,$$

hence the claim holds.

Inductive Hypothesis. Assume that after processing day $i - 1$,

$$\texttt{first\_profit} = P_1(i-1) \quad \text{and} \quad \texttt{second\_profit} = P_2(i-1).$$

Inductive Step. On day i, the algorithm updates

$$\texttt{first\_buy} \leftarrow \min(\texttt{first\_buy},\ \text{prices}[i]),$$

so $\texttt{first\_buy} = B_1(i)$. It then updates

$$\texttt{first\_profit} \leftarrow \max(\texttt{first\_profit},\ \text{prices}[i] - \texttt{first\_buy}),$$

which equals $P_1(i)$ by Lemma 1.

Next, the algorithm updates

$$\texttt{second\_buy} \leftarrow \max(\texttt{second\_buy},\ \texttt{first\_profit} - \text{prices}[i]),$$

so $\texttt{second\_buy} = B_2(i)$. Finally, it updates

$$\texttt{second\_profit} \leftarrow \max(\texttt{second\_profit},\ \texttt{second\_buy} + \text{prices}[i]),$$

which equals $P_2(i)$ by Lemma 2.

Thus, the inductive claim holds for day i.

By induction, the claim holds for all $i \in \{0, \ldots, n-1\}$. In particular, after the final iteration,

$$\texttt{second\_profit} = P_2(n-1),$$

which is the maximum achievable profit using at most two transactions. □

4 Conclusion

Undoubtedly the standard DP solution is extremely clean: it folds the entire problem into a single pass, and its proof follows almost mechanically from the update rules.

My approach takes a slightly longer route. By first fixing the best single transaction and then reasoning about where a second transaction can live, it makes the structure of the solution explicit. Instead of tracking states, it partitions the problem into a three unavoidable cases.

I hope I was able to share new perspective on this problem!