Best Time to Buy and Sell Stock IV

Solution

You are given an integer array `prices` where `prices[i]` is the price of a given stock on the `i^th` day, and an integer `k`.

Find the maximum profit you can achieve. You may complete at most `k` transactions.

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

Example 1:

```
Input: k = 2, prices = [2,4,1]
Output: 2
Explanation: Buy on day 1 (price = 2) and sell on day 2 (price = 4), profit = 4-2 = 2.
```

Example 2:

```
Input: k = 2, prices = [3,2,6,5,0,3]
Output: 7
Explanation: Buy on day 2 (price = 2) and sell on day 3 (price = 6), profit = 6-2 = 4. Then buy on day
```

Constraints:

- `0 <= k <= 100`
- `0 <= prices.length <= 1000`
- `0 <= prices[i] <= 1000`

Solution

Overview

You probably can guess from the problem title, this is the fourth problem in the series of Best Time to Buy and Sell Stock problem. It's strongly recommended that you should finish the previous problems before starting this one. Nevertheless, it's not necessary to finish the previous problems to understand this solution, and you can even use the methods we provide to help you solve the other problems.

Here, two approaches are introduced: *Dynamic Programming* approach, and *Merging* approach. Both are awesome, but the first method is more universal to other problems.

Approach 1: Dynamic Programming

Intuition

Dynamic programming (dp) is a popular method among hard-level problems. Its basic idea is to store the previous result to reduce redundant calculations. However, it is hard for beginners to think of the dp method. Below, a step-by-step tutorial of how to think of dp is introduced. If you are already familiar with dp, you can jump to the algorithm part to check out the actual implementation.

Generally, there are two ways to come up with a dp solution. One way is to start with a brute force approach and reduce unnecessary calculations. Another way is to treat the stored results as "states", and try to jump from the starting state to the ending state.

For beginners, it is recommended to start with the brute force approach. So, how to brute force to solve this problem?

Back to (part of) the question:

> 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 at most k transactions.

Cool, looks like we need to arrange at most k transactions. A natural idea is to iterate all the possible combinations of k transactions, and then find the best combination. As for those with less than k transactions, they are similar and can be considered later. A transaction consists of two parts: buying and selling. Therefore, we need to find 2k points in the stock line, k points for buying, and k points for selling.

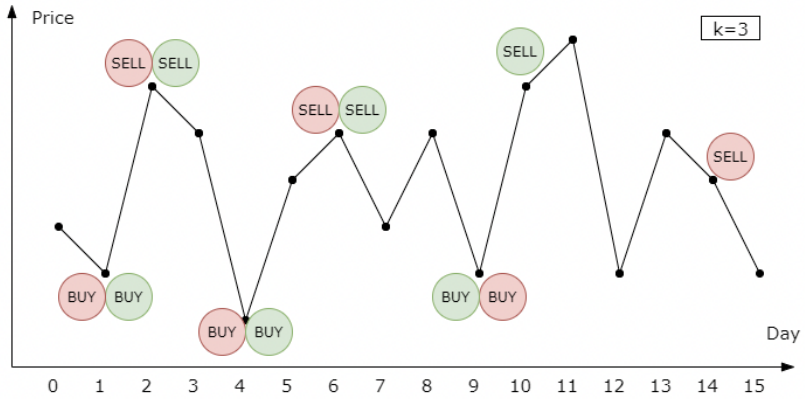
Now, we can roughly estimate the time complexity. Suppose there are n days in total, and we need to pick 2k days. The number of possible situations is about $C_n^{2k} = \frac{n!}{(2k)!(n-2k)!}$. It's not a good result because it involves factorial, which is likely to cause Time Limit Exceeded (TLE). Usually what we need is a polynomial one. However, it includes some invalid situations so the actual number is smaller.

Another problem is that, what if 2k is larger than n? In this case, we are not able to pick 2k points from n points, which means we will not reach the limit no matter how we try. Therefore, all we need to do is to iterate each day, and if the price of day i arise, buy the stock in i-1th day and sell it at ith day.

> 2k > n is a special case and can be addressed easily.

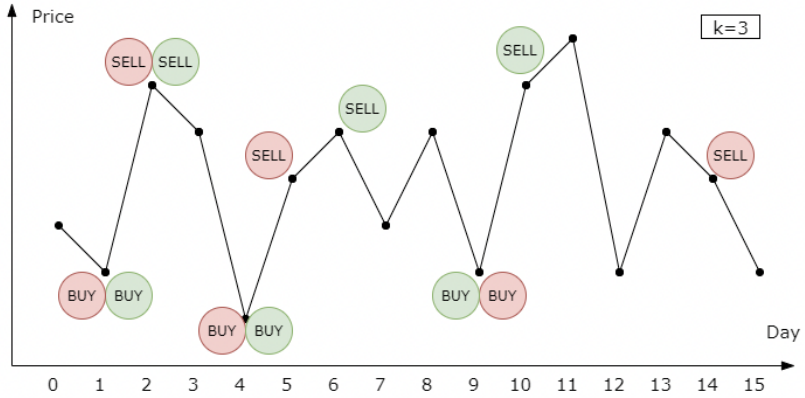
Back to our factorial number. The next step is to review our brute force approach and find out the possible redundant calculations. In our brute force approach, we need to iterate all the possible combinations and calculate the profit of each one to find the best. Can you find out where repeated calculations are?

Consider the following case, where the red color represents a possible combination, and the green represents another one:



The two combinations are the same before day 10. If we calculate the profits separately, we need to calculate the profit before day 10 twice. Here is where dp comes! We can store the current balance on day 9, and reuse it later. Therefore, we can store the result in a dict, where the key is the day number and the transactions we made before, and the value is the balance. Wait a minute, can we do better?

Consider another case:



The only difference is that the red sells stock at a lower price during the second transaction. Therefore, the red has a lower profit on day 10 than the green has. In this case, we need not calculate the rest profit of the red, since it can not beat the green in the future.

Therefore, we can compare those reds, and continue the next day with the one with the highest profit. However, we need to ensure that the best one will not be beaten by the "losers" in the future, so they should have the same "resources" at the time we store and compare the balances.

Hence, we can use three characteristics to store the profit: the day number, the transaction number used, and the stock holding status. You can use other representations of resources, such as using "the day remained" instead of "the day number". Feel free to try. Now, let's go to the algorithm part.

Algorithm

In the previous part, we introduced an intuitive idea from brute force to dp method, and here we need to decide the details of the algorithm.

We can either store the dp results in a dict or an array. Array costs less time for accessing and updating than dict, so we always prefer an array when possible. Because of three needed characteristics (day number, transaction number used, stock holding status), a three-dimensional array is our choice. We can use `dp[day_number][used_transaction_number][stock_holding_status]` to represent our states, where `stock_holding_status` is a 0/1 number representing whether you hold the stock or not.

> The value of `dp[i][j][l]` represents the best profit we can have at the end of the `i`-th day, with `j` remaining transactions to make and `l` stocks.

The next step is finding out the so-called "transition equation", which is a method that tells you how to jump from one state to another.

We start with `dp[0][0][0] = 0` and `dp[0][0][1]=-prices[0]`, and our final aim is max of `dp[n-1][j][0]` from `j=0` to `j=k`. Now, we need to fill out the entire array to find out the result. Assume we have gotten the results before day `i`, and we need to calculate the profit of day `i`. There are only four possible actions we can do on day `i`: 1. keep holding the stock, 2. keep not holding the stock, 3. buy the stock, or 4. sell the stock. The profit is easy to calculate.

1. Keep holding the stock:

$$dp[i][j][1] = dp[i-1][j][1]$$

2. Keep not holding the stock:

$$dp[i][j][0] = dp[i-1][j][0]$$

3. Buying, when j>0:

$$dp[i][j][1] = dp[i-1][j-1][0] - prices[i]$$

4. Selling:

$$dp[i][j][0] = dp[i-1][j][1] + prices[i]$$

We can combine they together to find the maximum profit:

$$dp[i][j][1] = max(dp[i-1][j][1], dp[i-1][j-1][0] - prices[i])$$

$$dp[i][j][0] = max(dp[i-1][j][0], dp[i-1][j][1] + prices[i])$$

Awesome! Now we can use for-loop to calculate the whole dp array and achieve our final result. Remember to solve the special cases when 2k > n.

```
public class Solution {
    public int maxProfit(int k, int[] prices) {
        int n = prices.length;

        // solve special cases
        if (n <= 0 || k <= 0) {
            return 0;
        }

        if (2 * k > n) {
            int res = 0;
            for (int i = 1; i < n; i++) {
                res += Math.max(0, prices[i] - prices[i - 1]);
            }
            return res;
        }

        // dp[i][used_k][ishold] = balance
        // ishold: 0 nothold, 1 hold
```

```
        int[][][] dp = new int[n][k + 1][2];

        // initialize the array with -inf
        // we use -1e9 here to prevent overflow
        for (int i = 0; i < n; i++) {
            for (int j = 0; j <= k; j++) {
                dp[i][j][0] = -1000000000;
                dp[i][j][1] = -1000000000;
            }
        }

        // set starting value
        dp[0][0][0] = 0;
        dp[0][1][1] = -prices[0];

        // fill the array
        for (int i = 1; i < n; i++) {
            for (int j = 0; j <= k; j++) {
                // transition equation
                dp[i][j][0] = Math.max(dp[i - 1][j][0], dp[i - 1][j][1] + prices[i]);
                // you can't hold stock without any transaction
                if (j > 0) {
                    dp[i][j][1] = Math.max(dp[i - 1][j][1], dp[i - 1][j - 1][0] - prices[i]);
                }
            }
        }

        int res = 0;
        for (int j = 0; j <= k; j++) {
            res = Math.max(res, dp[n - 1][j][0]);
        }

        return res;
    }
}
```

There a few points you should notice from the code above:

1. Take care of the initial values in dp array. Generally, it's ok to initialize them to zero. However, in this case, we need to make them -inf to mark impossible situations, such as `dp[0][0][1]`.
2. You can reverse the order of filling the dp array, with some modifications in the transition equation. For example, decreasing `j` instead of increasing it.
3. Some state-compressed method can be applied if you want. For example, we only need `dp[i-1]`, when calculating `dp[i]`, therefore we can delete other useless `dp` to save memory. Just using two arrays to storing `dp[i-1]` and `dp[i]` and refreshing them every iteration will do.
4. The code above is not the fastest because we prioritize the readability. It would be faster if you put the larger dimension in the inner array since it uses CPU cache more efficiently.

Complexity

- Time Complexity: $\mathcal{O}(nk)$ if $2k \leq n$, $\mathcal{O}(n)$ if $2k > n$, where n is the length of the `prices` sequence, since we have two for-loop.
- Space Complexity: $\mathcal{O}(nk)$ without state-compressed, and $\mathcal{O}(k)$ with state-compressed, where n is the length of the `prices` sequence.

Approach 2: Merging

Intuition

This approach starts from a simple situation with k=infinity, and drecrease k one by one.

Consider a weakened problem when k=infinity. Since we already know the prices of tomorrow, our solution is to trade whenever `prices[i-1] < prices[i]`. Below is an example.

We only used 4 transactions! However, what we need to solve is the case with an actual k. Let's decrease k from inf and see what happens. Our solution can handle all the k >=4, since we only used 4 transactions. But what if k=3?

Notice that at day 5, we buy and sell the stock at the same time. We can cancel the redundant transaction without impact the final profit!

When k=2, the best solution is to buy at day 1 and day 9, and to sell on day 6 and day 10. Deleting any transactions cannot reach this solution. However, we can merge the previous two transactions to get to this. A naive approach is iterating all the near transactions and find out the pair with the lowest impact on the revenue. Since we decrease k one by one, reducing one transaction is enough. Ok, let's go to the algorithm part to check the detail.

Algorithm

The general idea is to store all consecutively increasing subsequence as the initial solution. Then delete or merge transactions until the number of transactions less than or equal to k.

```
public class Solution {
    public int maxProfit(int k, int[] prices) {
        int n = prices.length;

        // solve special cases
        if (n <= 0 || k <= 0) {
            return 0;
        }

        // find all consecutively increasing subsequence
        ArrayList<int[]> transactions = new ArrayList<>();
        int start = 0;
        int end = 0;
        for (int i = 1; i < n; i++) {
            if (prices[i] >= prices[i - 1]) {
                end = i;
            } else {
                if (end > start) {
                    int[] t = { start, end };
```

```
            transactions.add(t);
        }
        start = i;
    }
}
if (end > start) {
    int[] t = { start, end };
    transactions.add(t);
}

while (transactions.size() > k) {
    // check delete loss
    int delete_index = 0;
    int min_delete_loss = Integer.MAX_VALUE;
    for (int i = 0; i < transactions.size(); i++) {
        int[] t = transactions.get(i);
        int profit_loss = prices[t[1]] - prices[t[0]];
        if (profit_loss < min_delete_loss) {
            min_delete_loss = profit_loss;
            delete_index = i;
```

```
    }
}

// check merge loss
int merge_index = 0;
int min_merge_loss = Integer.MAX_VALUE;
for (int i = 1; i < transactions.size(); i++) {
    int[] t1 = transactions.get(i - 1);
    int[] t2 = transactions.get(i);
    int profit_loss = prices[t1[1]] - prices[t2[0]];
    if (profit_loss < min_merge_loss) {
        min_merge_loss = profit_loss;
        merge_index = i;
    }
}

// delete or merge
if (min_delete_loss <= min_merge_loss) {
    transactions.remove(delete_index);
} else {
```

```
                int[] t1 = transactions.get(merge_index - 1);
                int[] t2 = transactions.get(merge_index);
                t1[1] = t2[1];
                transactions.remove(merge_index);
            }

        }

        int res = 0;
        for (int[] t : transactions) {
            res += prices[t[1]] - prices[t[0]];
        }

        return res;
    }
}
```

Complexity

- Time Complexity: $\mathcal{O}(n(n-k))$ if $2k \leq n$, $\mathcal{O}(n)$ if $2k > n$, where n is the length of the price sequence. The maximum size of `transactions` is $\mathcal{O}(n)$, and we need $\mathcal{O}(n-k)$ iterations.
- Space Complexity: $\mathcal{O}(n)$, since we need a list to store `transactions` .

Alternatively, one could simply extend Kadane's algo to the case of multiple transactions

```
class Solution:
```

```
    def maxProfit(self, k: int, prices: List[int]) -> int:
        if 2*k >= len(prices):
            return sum(max(0, prices[i]-prices[i-1]) for i in range(1, len(prices)))

        pnl = [0]*len(prices)
        for _ in range(k):
            val = 0
            for i in range(1, len(pnl)):
                val = max(pnl[i], val + prices[i] - prices[i-1])
                pnl[i] = max(pnl[i-1], val)
        return pnl[-1]
```

Extended one pass simulation solution for Best Time to Buy and Sell Stock III for at most `k` transactions.

```
class Solution {
    public int maxProfit(int k, int[] prices) {
        if (k == 0) return 0;

        int[] profit = new int[k+1];
        int[] cost = new int[k+1];

        profit[0] = 0;
```

```
        Arrays.fill(cost, Integer.MAX_VALUE);


        for (int price: prices) {
            for (int i = 0; i < k; i++) {
                cost[i+1] = Math.min(cost[i+1], price - profit[i]);
                profit[i+1] = Math.max(profit[i+1], price - cost[i+1]);
            }
        }
        return profit[k];
    }
}
```

Accepted. Time taken: 6 ms.

Time complexity: O(n*k). Space complexity: O(k).