714. Best Time to Buy and Sell Stock with Transaction Fee

Most consistent ways of dealing with the series of stock problems

Last Edit: October 26, 2018 9:42 PM

fun4LeetCode ★
5819

Note: this is a repost of my original post here with updated solutions for this problem (714. Best Time to Buy and Sell Stock with Transaction Fee). If you are only looking for solutions, you can go directly to each section in part `II -- Applications to specific cases`.

Up to this point, I believe you have finished the following series of stock problems:

1. 121. Best Time to Buy and Sell Stock
2. 122. Best Time to Buy and Sell Stock II
3. 123. Best Time to Buy and Sell Stock III
4. 188. Best Time to Buy and Sell Stock IV
5. 309. Best Time to Buy and Sell Stock with Cooldown
6. 714. Best Time to Buy and Sell Stock with Transaction Fee

For each problem, we've got a couple of excellent posts explaining how to approach it. However, most of the posts failed to identify the connections among these problems and made it hard to develop a consistent way of dealing with this series of problems. Here I will introduce the most generalized solution applicable to all of these problems, and its specialization to each of the six problems above.

`I -- General cases`

The idea begins with the following question: **Given an array representing the price of stocks on each day, what determines the maximum profit we can obtain?**

Most of you can quickly come up with answers like "it depends on which day we are and how many transactions we are allowed to complete". Sure, those are important factors as they manifest themselves in the problem descriptions.

However, there is a hidden factor that is not so obvious but vital in determining the maximum profit, which is elaborated below.

First let's spell out the notations to streamline our analyses. Let `prices` be the stock price array with length `n`, `i` denote the `i-th` day (`i` will go from `0` to `n-1`), `k` denote the maximum number of transactions allowed to complete, `T[i][k]` be the maximum profit that could be gained at the end of the `i-th` day with at most `k` transactions. Apparently we have base cases: `T[-1][k] = T[i][0] = 0`, that is, no stock or no transaction yield no profit (note the first day has `i = 0` so `i = -1` means no stock). Now if we can somehow relate `T[i][k]` to its subproblems like `T[i-1][k], T[i][k-1], T[i-1][k-1], ...`, we will have a working recurrence relation and the problem can be solved recursively. So how do we achieve that?

The most straightforward way would be looking at actions taken on the `i-th` day. How many options do we have? The answer is three: **buy**, **sell**, **rest**. Which one should we take? The answer is: we don't really know, but to find out which one is easy. We can try each option and then choose the one that maximizes our profit, provided there are no other restrictions. However, we do have an extra restriction saying no multiple transactions are allowed at the same time, meaning if we decide to **buy** on the `i-th` day, there should be `0` stock held in our hand before we buy; if we decide to **sell** on the `i-th` day, there should be exactly `1` stock held in our hand before we sell. The number of stocks held in our hand is the hidden factor mentioned above that will affect the action on the `i-th` day and thus affect the maximum profit.

Therefore our definition of `T[i][k]` should really be split into two: `T[i][k][0]` and `T[i][k][1]`, where the **former** denotes the maximum profit at the end of the `i-th` day with at most `k` transactions and with `0` stock in our hand AFTER taking the action, while the **latter** denotes the maximum profit at the end of the `i-th` day with at most `k` transactions and with `1` stock in our hand AFTER taking the action. Now the base cases and the recurrence relations can be written as:

1. Base cases:
   `T[-1][k][0] = 0, T[-1][k][1] = -Infinity`
   `T[i][0][0] = 0, T[i][0][1] = -Infinity`
2. Recurrence relations:
   `T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])`
   `T[i][k][1] = max(T[i-1][k][1], T[i-1][k-1][0] - prices[i])`

For the base cases, `T[-1][k][0] = T[i][0][0] = 0` has the same meaning as before while `T[-1][k][1] = T[i][0][1] = -Infinity` emphasizes the fact that it is impossible for us to have `1` stock in hand if there is no stock available or no transactions are allowed.

For `T[i][k][0]` in the recurrence relations, the actions taken on the `i-th` day can only be **rest** and **sell**, since we have `0` stock in our hand at the end of the day. `T[i-1][k][0]` is the maximum profit if action **rest** is taken, while `T[i-1][k][1] + prices[i]` is the maximum profit if action **sell** is taken. Note that the maximum number of allowable transactions remains the same, due to the fact that a transaction consists of two actions coming as a pair -- **buy** and **sell**. Only action **buy** will change the maximum number of transactions allowed (well, there is actually an alternative interpretation, see my comment below).

For `T[i][k][1]` in the recurrence relations, the actions taken on the `i-th` day can only be **rest** and **buy**, since we have `1` stock in our hand at the end of the day. `T[i-1][k][1]` is the maximum profit if action **rest** is taken, while `T[i-1][k-1][0] - prices[i]` is the maximum profit if action **buy** is taken. Note that the maximum number

of allowable transactions decreases by one, since buying on the `i-th` day will use one transaction, as explained above.

To find the maximum profit at the end of the last day, we can simply loop through the `prices` array and update `T[i][k][0]` and `T[i][k][1]` according to the recurrence relations above. The final answer will be `T[i][k][0]` (we always have larger profit if we end up with `0` stock in hand).

`II -- Applications to specific cases`

The aforementioned six stock problems are classified by the value of `k`, which is the maximum number of allowable transactions (the last two also have additional requirements such as "cooldown" or "transaction fee"). I will apply the general solution to each of them one by one.

Case I: `k = 1`

For this case, we really have two unknown variables on each day: `T[i][1][0]` and `T[i][1][1]`, and the recurrence relations say:

```
T[i][1][0] = max(T[i-1][1][0], T[i-1][1][1] + prices[i])
T[i][1][1] = max(T[i-1][1][1], T[i-1][0][0] - prices[i]) = max(T[i-1][1][1], -prices[i])
```

where we have taken advantage of the base case `T[i][0][0] = 0` for the second equation.

It is straightforward to write the `O(n)` time and `O(n)` space solution, based on the two equations above. However, if you notice that the maximum profits on the `i-th` day actually only depend on those on the `(i-1)-th` day, the space can be cut down to `O(1)`. Here is the space-optimized solution:

```
public int maxProfit(int[] prices) {
    int T_i10 = 0, T_i11 = Integer.MIN_VALUE;

    for (int price : prices) {
        T_i10 = Math.max(T_i10, T_i11 + price);
        T_i11 = Math.max(T_i11, -price);
    }

    return T_i10;
}
```

Now let's try to gain some insight of the solution above. If we examine the part inside the loop more carefully, `T_i11` really just represents the maximum value of the negative of all stock prices up to the `i-th` day, or equivalently the minimum value of all the stock prices. As for `T_i10`, we just need to decide which action yields a higher profit, sell or rest. And if action sell is taken, the price at which we bought the stock is `T_i11`, i.e., the minimum value before the `i-th` day. This is exactly what we would do in reality if we want to gain maximum profit. I should point out that this is not the only way of solving the problem for this case. You may find some other nice solutions here.

Case II: `k = +Infinity`

If `k` is positive infinity, then there isn't really any difference between `k` and `k - 1` (wonder why? see my comment below), which implies `T[i-1][k-1][0] = T[i-1][k][0]` and `T[i-1][k-1][1] = T[i-1][k][1]`. Therefore, we still have two unknown variables on each day: `T[i][k][0]` and `T[i][k][1]` with `k = +Infinity`, and the recurrence relations say:

`T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])`
`T[i][k][1] = max(T[i-1][k][1], T[i-1][k-1][0] - prices[i]) = max(T[i-1][k][1], T[i-1][k][0] - prices[i])`

where we have taken advantage of the fact that `T[i-1][k-1][0] = T[i-1][k][0]` for the second equation. The `O(n)` time and `O(1)` space solution is as follows:

```
public int maxProfit(int[] prices) {
    int T_ik0 = 0, T_ik1 = Integer.MIN_VALUE;

    for (int price : prices) {
        int T_ik0_old = T_ik0;
        T_ik0 = Math.max(T_ik0, T_ik1 + price);
        T_ik1 = Math.max(T_ik1, T_ik0_old - price);
    }

    return T_ik0;
}
```

(**Note**: The caching of the old values of `T_ik0`, that is, the variable `T_ik0_old`, is unnecessary. Special thanks to 0x0101 and elvina for clarifying this.)

This solution suggests a greedy strategy of gaining maximum profit: as long as possible, buy stock at each local minimum and sell at the immediately followed local maximum. This is equivalent to finding increasing subarrays in `prices` (the stock price array), and buying at the beginning price of each subarray while selling at its end price. It's easy to show that this is the same as accumulating profits as long as it is profitable to do so, as demonstrated in this post.

Case III: `k = 2`

Similar to the case where `k = 1`, except now we have four variables instead of two on each day: `T[i][1][0]`, `T[i][1][1]`, `T[i][2][0]`, `T[i][2][1]`, and the recurrence relations are:

`T[i][2][0] = max(T[i-1][2][0], T[i-1][2][1] + prices[i])`
`T[i][2][1] = max(T[i-1][2][1], T[i-1][1][0] - prices[i])`
`T[i][1][0] = max(T[i-1][1][0], T[i-1][1][1] + prices[i])`
`T[i][1][1] = max(T[i-1][1][1], -prices[i])`

where again we have taken advantage of the base case `T[i][0][0] = 0` for the last equation. The `O(n)` time and `O(1)` space solution is as follows:

```
public int maxProfit(int[] prices) {
    int T_i10 = 0, T_i11 = Integer.MIN_VALUE;
    int T_i20 = 0, T_i21 = Integer.MIN_VALUE;

    for (int price : prices) {
```

```
        T_i20 = Math.max(T_i20, T_i21 + price);
        T_i21 = Math.max(T_i21, T_i10 - price);
        T_i10 = Math.max(T_i10, T_i11 + price);
        T_i11 = Math.max(T_i11, -price);
    }

    return T_i20;
}
```

which is essentially the same as the one given here.

Case IV: `k is arbitrary`

This is the most general case so on each day we need to update all the maximum profits with different `k` values corresponding to `0` or `1` stocks in hand at the end of the day. However, there is a minor optimization we can do if `k` exceeds some critical value, beyond which the maximum profit will no long depend on the number of allowable transactions but instead will be bound by the number of available stocks (length of the `prices` array). Let's figure out what this critical value will be.

A profitable transaction takes at least two days (buy at one day and sell at the other, provided the buying price is less than the selling price). If the length of the `prices` array is `n`, the maximum number of profitable transactions is `n/2` (integer division). After that no profitable transaction is possible, which implies the maximum profit will stay the same. Therefore the critical value of `k` is `n/2`. If the given `k` is no less than this value, i.e., `k >= n/2`, we can extend `k` to positive infinity and the problem is equivalent to `Case II`.

The following is the `O(kn)` time and `O(k)` space solution. Without the optimization, the code will be met with TLE for large `k` values.

```
public int maxProfit(int k, int[] prices) {
    if (k >= prices.length >>> 1) {
        int T_ik0 = 0, T_ik1 = Integer.MIN_VALUE;

        for (int price : prices) {
            int T_ik0_old = T_ik0;
            T_ik0 = Math.max(T_ik0, T_ik1 + price);
            T_ik1 = Math.max(T_ik1, T_ik0_old - price);
        }

        return T_ik0;
    }

    int[] T_ik0 = new int[k + 1];
    int[] T_ik1 = new int[k + 1];
    Arrays.fill(T_ik1, Integer.MIN_VALUE);

    for (int price : prices) {
        for (int j = k; j > 0; j--) {
            T_ik0[j] = Math.max(T_ik0[j], T_ik1[j] + price);
            T_ik1[j] = Math.max(T_ik1[j], T_ik0[j - 1] - price);
        }
    }

    return T_ik0[k];
}
```

The solution is similar to the one found in this post. Here I used backward looping for the `T` array to avoid using temporary variables. It turns out that it is possible to do forward looping without temporary variables, too.

Case V: `k = +Infinity but with cooldown`

This case resembles `Case II` very much due to the fact that they have the same `k` value, except now the recurrence relations have to be modified slightly to account for the "**cooldown**" requirement. The original recurrence relations for `Case II` are given by

```
T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])
T[i][k][1] = max(T[i-1][k][1], T[i-1][k][0] - prices[i])
```

But with "cooldown", we cannot buy on the `i-th` day if a stock is sold on the `(i-1)-th` day. Therefore, in the second equation above, instead of `T[i-1][k][0]`, we should actually use `T[i-2][k][0]` if we intend to buy on the `i-th` day. Everything else remains the same and the new recurrence relations are

```
T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])
T[i][k][1] = max(T[i-1][k][1], T[i-2][k][0] - prices[i])
```

And here is the `O(n)` time and `O(1)` space solution:

```
public int maxProfit(int[] prices) {
    int T_ik0_pre = 0, T_ik0 = 0, T_ik1 = Integer.MIN_VALUE;

    for (int price : prices) {
        int T_ik0_old = T_ik0;
        T_ik0 = Math.max(T_ik0, T_ik1 + price);
        T_ik1 = Math.max(T_ik1, T_ik0_pre - price);
        T_ik0_pre = T_ik0_old;
    }

    return T_ik0;
}
```

dietpepsi shared a very nice solution here with thinking process, which turns out to be the same as the one above.

Case VI: `k = +Infinity but with transaction fee`

Again this case resembles `Case II` very much as they have the same `k` value, except now the recurrence relations need to be modified slightly to account for the "**transaction fee**" requirement. The original recurrence relations for `Case II` are given by

```
T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])
T[i][k][1] = max(T[i-1][k][1], T[i-1][k][0] - prices[i])
```

Since now we need to pay some fee (denoted as `fee`) for each transaction made, the profit after buying or selling the stock on the `i-th` day should be subtracted by this amount, therefore the new recurrence relations will be either

`T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])`
`T[i][k][1] = max(T[i-1][k][1], T[i-1][k][0] - prices[i] - fee)`

or

`T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i] - fee)`
`T[i][k][1] = max(T[i-1][k][1], T[i-1][k][0] - prices[i])`

Note we have two options as for when to subtract the `fee`. This is because (as I mentioned above) each transaction is characterized by two actions coming as a pair - - **buy** and **sell**. The fee can be paid either when we buy the stock (corresponds to the first set of equations) or when we sell it (corresponds to the second set of equations). The following are the `O(n)` time and `O(1)` space solutions corresponding to these two options, where for the second solution we need to pay attention to possible overflows.

Solution I -- pay the fee when buying the stock:

```
public int maxProfit(int[] prices, int fee) {
    int T_ik0 = 0, T_ik1 = Integer.MIN_VALUE;

    for (int price : prices) {
        int T_ik0_old = T_ik0;
        T_ik0 = Math.max(T_ik0, T_ik1 + price);
        T_ik1 = Math.max(T_ik1, T_ik0_old - price - fee);
    }

    return T_ik0;
}
```

Solution II -- pay the fee when selling the stock:

```
public int maxProfit(int[] prices, int fee) {
    long T_ik0 = 0, T_ik1 = Integer.MIN_VALUE;

    for (int price : prices) {
        long T_ik0_old = T_ik0;
        T_ik0 = Math.max(T_ik0, T_ik1 + price - fee);
        T_ik1 = Math.max(T_ik1, T_ik0_old - price);
    }

    return (int)T_ik0;
}
```

`III -- Summary`

In summary, the most general case of the stock problem can be characterized by three factors, the ordinal of the day `i`, the maximum number of allowable transactions `k`, and the number of stocks in our hand at the end of the day. I have shown the recurrence relations for the maximum profits and their termination conditions, which leads to the `O(nk)` time and `O(k)` space solution. The results are then applied to each of the six cases, with the last two using slightly modified recurrence relations due to the additional requirements. I should mention that peterleetcode also introduced a nice solution here which generalizes to arbitrary `k` values. If you have a taste, take a look.

Hope this helps and happy coding!

Comments: 52

Sort By

Login to Comment

winterddd ★ 0 October 27, 2018 1:27 PM

sorry，i cant understand why t[i-1][k-1][0]=t[i-1][k][0] , could anyone explain it?

0 Share

SHOW 1 REPLY

marvel01 ★ 1 October 17, 2018 2:26 PM

Excellent!

1 Share

haotianliu1801 ★ 4 October 8, 2018 6:24 PM

Wow so comprehensive!

1 Share

spmonkey ★ 2 September 22, 2018 3:21 PM

if only there was a collection of all your posts

1 Share

harshlal028 ★ 0 September 17, 2018 6:40 PM

Usually people want to jump to implementation before understanding the theory and these problems are a classic case of that. If you don't understand the recurrence relations then its useless to just solve one problem as it cannot be generalized. One should always try to learn from each problem solved to generalize to further problems.
Great job @fun4LeetCode at explaining the recurrences. Kudos!

Read More

0 Share

007623 ★ 26 September 10, 2018 7:35 AM

Great post! I think that if use `k` as `remained available of transactions`, is easier to understand than max transactions. (yes, so in the code, the new k become max_k-k)

0 Share

GupiBagha ★ 13 September 3, 2018 9:04 PM

Bloody brilliant!

0 Share

marswork ★ 94 August 19, 2018 2:20 PM

Masterpiece

1 Share

mh7 ★ 3 August 16, 2018 4:36 AM

Wish I could give +10 votes. Hats off to such a brilliant analysis!

1 Share

Snake_Cool ★ 0 August 9, 2018 5:33 PM

Hi @fun4LeetCode Nice post, thanks a lot! one more thing: Is that possible to draw a graph for this solution, does it has to be 3-d matrix?

0 Share

SHOW 1 REPLY