Best Time to Buy and Sell Stock III

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 *two* transactions.

Note:
You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Solution 1

The thinking is simple and is inspired by the best solution from Single Number II (I read through the discussion after I use DP). Assume we only have 0 money at first; 4 Variables to maintain some interested 'ceilings' so far: The maximum of if we've just buy 1st stock, if we've just sold 1nd stock, if we've just buy 2nd stock, if we've just sold 2nd stock. Very simple code too and work well. I have to say the logic is simple than those in Single Number II.

```
public class Solution {
    public int maxProfit(int[] prices) {
        int hold1 = Integer.MIN_VALUE, hold2 = Integer.MIN_VALUE;
        int release1 = 0, release2 = 0;
        for(int i:prices){                              // Assume we only have 0 
money at first
            release2 = Math.max(release2, hold2+i);     // The maximum if we've j
ust sold 2nd stock so far.
            hold2    = Math.max(hold2,    release1-i);  // The maximum if we've j
ust buy  2nd stock so far.
            release1 = Math.max(release1, hold1+i);     // The maximum if we've j
ust sold 1nd stock so far.
            hold1    = Math.max(hold1,    -i);          // The maximum if we've j
ust buy  1st stock so far. 
        }
        return release2; ///Since release1 is initiated as 0, so release2 will alw
ays higher than release1.
    }
}
```

written by weijiac original link here

Solution 2

Solution is commented in the code. Time complexity is O(k*n), space complexity can be O(n) because this DP only uses the result from last step. But for cleaness this solution still used O(k*n) space complexity to preserve similarity to the equations in the comments.

```
class Solution {
public:
    int maxProfit(vector<int> &prices) {
        // f[k, ii] represents the max profit up until prices[ii] (Note: NOT endi
ng with prices[ii]) using at most k transactions.
        // f[k, ii] = max(f[k, ii-1], prices[ii] - prices[jj] + f[k-1, jj]) { jj
in range of [0, ii-1] }
        //          = max(f[k, ii-1], prices[ii] + max(f[k-1, jj] - prices[jj]))
        // f[0, ii] = 0; 0 times transation makes 0 profit
        // f[k, 0] = 0; if there is only one price data point you can't make any
money no matter how many times you can trade
        if (prices.size() <= 1) return 0;
        else {
            int K = 2; // number of max transation allowed
            int maxProf = 0;
            vector<vector<int>> f(K+1, vector<int>(prices.size(), 0));
            for (int kk = 1; kk <= K; kk++) {
                int tmpMax = f[kk-1][0] - prices[0];
                for (int ii = 1; ii < prices.size(); ii++) {
                    f[kk][ii] = max(f[kk][ii-1], prices[ii] + tmpMax);
                    tmpMax = max(tmpMax, f[kk-1][ii] - prices[ii]);
                    maxProf = max(f[kk][ii], maxProf);
                }
            }
            return maxProf;
        }
    }
};
```

written by peterleetcode original link here

Solution 3

It is similar to other buy/sell problems. just do DP and define an array of states to track the current maximum profits at different stages. For example, in the below code

- states[][0]: one buy
- states[][1]: one buy, one sell
- states[][2]: two buys, one sell
- states[][3]: two buy, two sells

The states transistions occurs when buy/sell operations are executed. For example, state[][0] can move to state[][1] via one sell operation.

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int states[2][4] = {INT_MIN, 0, INT_MIN, 0}; // 0: 1 buy, 1: one buy/sell
, 2: 2 buys/1 sell, 3, 2 buys/sells
        int len = prices.size(), i, cur = 0, next =1;
        for(i=0; i<len; ++i)
        {
            states[next][0] = max(states[cur][0], -prices[i]);
            states[next][1] = max(states[cur][1], states[cur][0]+prices[i]);
            states[next][2] = max(states[cur][2], states[cur][1]-prices[i]);
            states[next][3] = max(states[cur][3], states[cur][2]+prices[i]);
            swap(next, cur);
        }
        return max(states[cur][1], states[cur][3]);
    }
};
```

written by dong.wang.1694 original link here