Best Time to Buy and Sell Stock IV

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.

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

Credits:
Special thanks to @Freezen for adding this problem and creating all test cases.

Solution 1

The general idea is DP, while I had to add a "quickSolve" function to tackle some corner cases to avoid TLE.

DP: t(i,j) is the max profit for up to i transactions by time j (0<=i<=K, 0<=j<=T).

```
public int maxProfit(int k, int[] prices) {
    int len = prices.length;
    if (k >= len / 2) return quickSolve(prices);

    int[][] t = new int[k + 1][len];
    for (int i = 1; i <= k; i++) {
        int tmpMax =  -prices[0];
        for (int j = 1; j < len; j++) {
            t[i][j] = Math.max(t[i][j - 1], prices[j] + tmpMax);
            tmpMax =  Math.max(tmpMax, t[i - 1][j - 1] - prices[j]);
        }
    }
    return t[k][len - 1];
}


private int quickSolve(int[] prices) {
    int len = prices.length, profit = 0;
    for (int i = 1; i < len; i++)
        // as long as there is a price gap, we gain a profit.
        if (prices[i] > prices[i - 1]) profit += prices[i] - prices[i - 1];
    return profit;
}
```

written by jinrf original link here

Solution 2

We can find all adjacent valley/peak pairs and calculate the profits easily. Instead of accumulating all these profits like Buy&Sell Stock II, we need the highest k ones.

The key point is when there are two v/p pairs (v1, p1) and (v2, p2), satisfying v1 <= v2 and p1 <= p2, we can either make one transaction at [v1, p2], or make two at both [v1, p1] and [v2, p2]. The trick is to treat [v1, p2] as the first transaction, and [v2, p1] as the second. Then we can guarantee the right max profits in both situations, **p2 - v1** for one transaction and **p1 - v1 + p2 - v2** for two.

Finding all v/p pairs and calculating the profits takes O(n) since there are up to n/2 such pairs. And extracting k maximums from the heap consumes another O(klgn).

```
class Solution {
public:
    int maxProfit(int k, vector<int> &prices) {
        int n = (int)prices.size(), ret = 0, v, p = 0;
        priority_queue<int> profits;
        stack<pair<int, int> > vp_pairs;
        while (p < n) {
            // find next valley/peak pair
            for (v = p; v < n - 1 && prices[v] >= prices[v+1]; v++);
            for (p = v + 1; p < n && prices[p] >= prices[p-1]; p++);
            // save profit of 1 transaction at last v/p pair, if current v is lower than last v
            while (!vp_pairs.empty() && prices[v] < prices[vp_pairs.top().first]) {
                profits.push(prices[vp_pairs.top().second-1] - prices[vp_pairs.top().first]);
                vp_pairs.pop();
            }
            // save profit difference between 1 transaction (last v and current p) and 2 transactions (last v/p + current v/p),
            // if current v is higher than last v and current p is higher than last p
            while (!vp_pairs.empty() && prices[p-1] >= prices[vp_pairs.top().second-1]) {
                profits.push(prices[vp_pairs.top().second-1] - prices[v]);
                v = vp_pairs.top().first;
                vp_pairs.pop();
            }
            vp_pairs.push(pair<int, int>(v, p));
        }
        // save profits of the rest v/p pairs
        while (!vp_pairs.empty()) {
            profits.push(prices[vp_pairs.top().second-1] - prices[vp_pairs.top().first]);
            vp_pairs.pop();
        }
        // sum up first k highest profits
        for (int i = 0; i < k && !profits.empty(); i++) {
            ret += profits.top();
            profits.pop();
        }
        return ret;
    }
};
```

written by yishiluo original link here

Solution 3

```
/**
 * dp[i, j] represents the max profit up until prices[j] using at most i transacti
ons.
 * dp[i, j] = max(dp[i, j-1], prices[j] - prices[jj] + dp[i-1, jj]) { jj in range
of [0, j-1] }
 *          = max(dp[i, j-1], prices[j] + max(dp[i-1, jj] - prices[jj]))
 * dp[0, j] = 0; 0 transactions makes 0 profit
 * dp[i, 0] = 0; if there is only one price data point you can't make any transact
ion.
 */

public int maxProfit(int k, int[] prices) {
    int n = prices.length;
    if (n <= 1)
        return 0;

    //if k >= n/2, then you can make maximum number of transactions.
    if (k >=  n/2) {
        int maxPro = 0;
        for (int i = 1; i < n; i++) {
            if (prices[i] > prices[i-1])
                maxPro += prices[i] - prices[i-1];
        }
        return maxPro;
    }

    int[][] dp = new int[k+1][n];
    for (int i = 1; i <= k; i++) {
        int localMax = dp[i-1][0] - prices[0];
        for (int j = 1; j < n; j++) {
            dp[i][j] = Math.max(dp[i][j-1],  prices[j] + localMax);
            localMax = Math.max(localMax, dp[i-1][j] - prices[j]);
        }
    }
    return dp[k][n-1];
}
```

written by jinwu original link here