<https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/description/>

You are given an array prices where prices[i] is the price of a given stock on the ith day, and an integer fee representing a transaction fee.

Find the maximum profit you can achieve. You may complete as many transactions as you like, but you need to pay the transaction fee for each transaction.

**Note:**

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

The transaction fee is only charged once for each stock purchase and sale.

**Example 1:**

Input: prices = [1,3,2,8,4,9], fee = 2

Output: 8

Explanation: The maximum profit can be achieved by:

- Buying at prices[0] = 1

- Selling at prices[3] = 8

- Buying at prices[4] = 4

- Selling at prices[5] = 9

The total profit is ((8 - 1) - 2) + ((9 - 4) - 2) = 8.

**Example 2:**

Input: prices = [1,3,7,5,10,3], fee = 3

Output: 6

**Constraints:**

1 <= prices.length <= 5 \* 104

1 <= prices[i] < 5 \* 104

0 <= fee < 5 \* 104

**Attempt 1: 2023-11-04**

**Solution 1: Native DFS (10 min, TLE 19/44)**

class Solution {

public int maxProfit(int[] prices, int fee) {

return helper(prices, 0, 1, fee);

}

private int helper(int[] prices, int index, int buy, int fee) {

if(index >= prices.length) {

return 0;

}

int profit = 0;

if(buy == 1) {

int not\_buy = helper(prices, index + 1, 1, fee);

int buy\_it = helper(prices, index + 1, 0, fee) - prices[index];

profit = Math.max(not\_buy, buy\_it);

} else {

int not\_sell = helper(prices, index + 1, 0, fee);

int sell\_it = helper(prices, index + 1, 1, fee) + prices[index] - fee;

profit = Math.max(not\_sell, sell\_it);

}

return profit;

}

}

Time Complexity:O(2^n)

Space Complexity:O(n)

**Solution 2: DFS + Memoization (10 min)**

class Solution {

public int maxProfit(int[] prices, int fee) {

Integer[][] memo = new Integer[prices.length + 1][2];

return helper(prices, 0, 1, fee, memo);

}

private int helper(int[] prices, int index, int buy, int fee, Integer[][] memo) {

if(index >= prices.length) {

return 0;

}

if(memo[index][buy] != null) {

return memo[index][buy];

}

int profit = 0;

if(buy == 1) {

int not\_buy = helper(prices, index + 1, 1, fee, memo);

int buy\_it = helper(prices, index + 1, 0, fee, memo) - prices[index];

profit = Math.max(not\_buy, buy\_it);

} else {

int not\_sell = helper(prices, index + 1, 0, fee, memo);

int sell\_it = helper(prices, index + 1, 1, fee, memo) + prices[index] - fee;

profit = Math.max(not\_sell, sell\_it);

}

return memo[index][buy] = profit;

}

}

Time Complexity:O(2\*N)

Space Complexity:O(2\*N) + O(N)

**Solution 3: DP (10 min)**

class Solution {

public int maxProfit(int[] prices, int fee) {

int n = prices.length;

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

dp[n][0] = 0;

for(int i = n - 1; i >= 0; i--) {

for(int buy = 0; buy <= 1; buy++) {

int profit = 0;

if(buy == 1) {

profit = Math.max(dp[i + 1][1], dp[i + 1][0] - prices[i]);

} else {

profit = Math.max(dp[i + 1][0], dp[i + 1][1] + prices[i] - fee);

}

dp[i][buy] = profit;

}

}

return dp[0][1];

}

}

Time Complexity:O(N)

Space Complexity:O(2\*N)

**Solution 4: DP + Space Optimization (10 min)**

class Solution {

public int maxProfit(int[] prices, int fee) {

int n = prices.length;

int[] dpPrev = new int[2];

int[] dp = new int[2];

for(int i = n - 1; i >= 0; i--) {

for(int buy = 0; buy <= 1; buy++) {

int profit = 0;

if(buy == 1) {

profit = Math.max(dpPrev[1], dpPrev[0] - prices[i]);

} else {

profit = Math.max(dpPrev[0], dpPrev[1] + prices[i] - fee);

}

dp[buy] = profit;

}

dpPrev = dp.clone();

}

return dp[1];

}

}

Time Complexity:O(2\*N)

Space Complexity:O(2\*2)

**Solution 5: State Machine (10 min)**

**Style 1: Define s0, s1 status as array**

class Solution {

public int maxProfit(int[] prices, int fee) {

int n = prices.length;

// s0 means initial status, able to buy

int[] s0 = new int[n];

// s1 means status after buy, able to sell

int[] s1 = new int[n];

// s0 initial status as 0, no buy

s0[0] = 0;

// s1 initial status as after buy

s1[0] = -prices[0];

for(int i = 1; i < n; i++) {

// s0 current status depends on max between s0 old status

// and s1 old status but sell stock with fee

s0[i] = Math.max(s0[i - 1], s1[i - 1] + prices[i] - fee);

// s1 current status depends on max between s1 old status

// and s0 old status but buy stock

s1[i] = Math.max(s1[i - 1], s0[i - 1] - prices[i]);

}

// The final max profit must s0 because no one can buy stock and

// left with more profit without sell it, so either no buy no

// sell(at s0), or buy and sell(back from s1 to s0) will create

// the max profit, buy and no sell yet(at s1) not the choice

return s0[n - 1];

}

}

Time Complexity:O(N)

Space Complexity:O(N)

**Style 2: Define s0, s1 status as variable**

class Solution {

public int maxProfit(int[] prices, int fee) {

int n = prices.length;

// s0 means initial status, able to buy, s0 initial status as 0, no buy

//int[] s0 = new int[n];

int s0 = 0;

// s1 means status after buy, able to sell, s1 initial status as after buy

//int[] s1 = new int[n];

int s1 = -prices[0];

for(int i = 1; i < n; i++) {

// s0 current status depends on max between s0 old status

// and s1 old status but sell stock with fee

//s0[i] = Math.max(s0[i - 1], s1[i - 1] + prices[i] - fee);

s0 = Math.max(s0, s1 + prices[i] - fee);

// s1 current status depends on max between s1 old status

// and s0 old status but buy stock

//s1[i] = Math.max(s1[i - 1], s0[i - 1] - prices[i]);

s1 = Math.max(s1, s0 - prices[i]);

}

// The final max profit must s0 because no one can buy stock and

// left with more profit without sell it, so either no buy no

// sell(at s0), or buy and sell(back from s1 to s0) will create

// the max profit, buy and no sell yet(at s1) not the choice

//return s0[n - 1];

return s0;

}

}

Time Complexity:O(N)

Space Complexity:O(1)

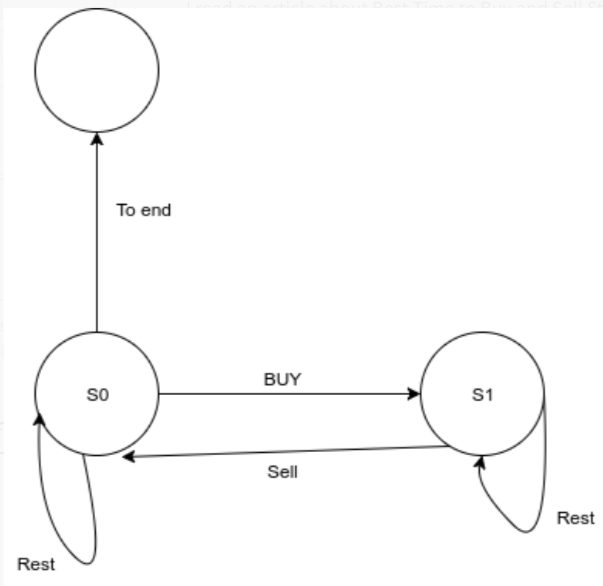
**Refer to**

<https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/solutions/160964/java-using-state-machine-like-stock-iii/>

I read an article about Best Time to Buy and Sell Stock III that used state machine, a very intuitive way.

That inspires me to solve this problem with same idea.

Suppose we have a state machine:



We should maintain each state with maximum profit.

So at S0, we could either do nothing, or we could buy a stock.

At s1, we cloud either do nothing, or we could sell current stock with fee.

To update state:

For s0, the incoming arrows from s0 itself or s1. So s0 = Math.max(s0, s1 + sale\_price - fee)

For s1, the incoming arrows from s0 and s1 itself, So s1 = Math.max(s1, s0 - buying\_prices)

class Solution {

public int maxProfit(int[] prices, int fee) {

if (prices.length == 0) {

return 0;

}

int s0 = 0;

int s1 = -prices[0];

for (int i = 1; i < prices.length; i++) {

s1 = Math.max(s1, s0 - prices[i]);

s0 = Math.max(s0, prices[i] + s1 - fee);

}

return s0;

}

}