**123. Best Time to Buy and Sell Stock III**

Say you have an array for which the ith 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 (i.e., you must sell the stock before you buy again).

**Example 1:**

**Input:** [3,3,5,0,0,3,1,4]

**Output:** 6

**Explanation:**

Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.

Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = 4-1 = 3.

**Example 2:**

**Input:** [1,2,3,4,5]

**Output:** 4

**Explanation:**

Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.

Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are

engaging multiple transactions at the same time. You must sell before buying again.

**Example 3:**

**Input:** [7,6,4,3,1]

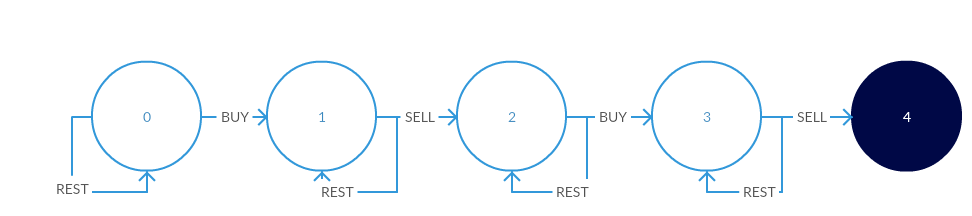
**Output:** 0

**Explanation:** In this case, no transaction is done, i.e. max profit = 0.

**Easy DP solution using state machine, O(n) time complexity, O(1) space complexity**

This approach can be used for all the problems based on stock prices.

The idea is to design a state machine that correctly describes the problem statement.



**Intuition behind the state diagram:**  
We begin at state 0, where we can either rest (i.e. do nothing) or buy stock at a given price.

* If we choose to rest, we remain in state 0
* If we buy, we spend some money (price of the stock on that day) and go to state 1

From state 1, we can once again choose to do nothing, or we can sell our stock.

* If we choose to rest, we remain in state 1
* If we sell, we earn some money (price of the stock on that day) and go to state 2

This completes one transaction for us. Remember, we can only do *at most* 2 transactions.

From state 2, we can choose to do nothing or buy more stock.

* If we choose to rest, we remain in state 2
* If we buy, we go to state 3

From state 3, we can once again choose to do nothing, or we can sell our stock for the last time.

* If we choose to rest, we remain in state 3
* If we sell, we have utilized our allowed transactions and reach the final state 4

**Going from the state diagram to code**

// Assume we are in state S

// If we buy, we are spending money but we can also choose to do nothing

// Doing nothing means going from S->S

// Buying means going from some state X->S, losing some money in the process

S = max(S, X-prices[i])

// Similarly, for selling a stock

S = max(S, X+prices[i])

**Code:**

int maxProfit(vector<int>& prices) {

if(prices.empty()) return 0;

int s1=-prices[0],s2=INT\_MIN,s3=INT\_MIN,s4=INT\_MIN;

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

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

s2 = max(s2, s1+prices[i]);

s3 = max(s3, s2-prices[i]);

s4 = max(s4, s3+prices[i]);

}

return max(0,s4);

}

We can create 4 variables, one for each state excluding the initial state since that's always 0, initializing s1 to -prices[0] and the rest to INT\_MIN since they will get overwritten later.

To reach s1, we either stay in s1 or we buy stock for the first time.  
To reach s2, we either stay in s2 or we sell from s1 and come to s2  
Similarly for s3 and s4.

In the end, we return s4 or more accurately, max(0,s4) since we initialize s4 to INT\_MIN.

This idea works for all problems on stocks, as long as our state diagram is correct, we can code it up like this.

Side Note: Technically, this is a dynammic programming approach and we should actually be doing s2[i] = max(s2[i-1], s1[i-1]+prices[i]) but we can be rest assured that the overwritten value of s1 will always be better than the previous one and hence we do not need temporary variables.