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

You are given an array prices where prices[i] is the price of a given stock on the ith day.

Find the maximum profit you can achieve. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times) with the following restrictions:

After you sell your stock, you cannot buy stock on the next day (i.e., cooldown one day).

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

**Example 1:**

Input: prices = [1,2,3,0,2]

Output: 3

Explanation: transactions = [buy, sell, cooldown, buy, sell]

**Example 2:**

Input: prices = [1]

Output: 0

**Constraints:**

1 <= prices.length <= 5000

0 <= prices[i] <= 1000

**Attempt 1: 2023-11-03**

**Solution 1: Native DFS (10 min, TLE 208/210)**

class Solution {

public int maxProfit(int[] prices) {

return helper(prices, 0, 1);

}

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

if(index >= prices.length) {

return 0;

}

int profit = 0;

if(buy == 1) {

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

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

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

} else {

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

// After you sell your stock, you cannot buy stock on the next day (i.e., cooldown one day).

int sell\_it = helper(prices, index + 2, 1) + prices[index];

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) {

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

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

}

private int helper(int[] prices, int index, int buy, 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, memo);

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

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

} else {

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

// After you sell your stock, you cannot buy stock on the next day (i.e., cooldown one day).

int sell\_it = helper(prices, index + 2, 1, memo) + prices[index];

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 n = prices.length;

// Taken size of n + 2, because in case of getting

// sold at last day(index) gets cooldown to last + 2

int[][] dp = new int[n + 2][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 + 2][1] + prices[i]);

}

dp[i][buy] = profit;

}

}

return dp[0][1];

}

}

Time Complexity:O(2\*N)

Space Complexity:O(2\*N)

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

class Solution {

public int maxProfit(int[] prices) {

int n = prices.length;

// A bit different than normal space optimization,

// since we have dp[i], dp[i + 1] and dp[i + 2],

// which requires one more previous status storage,

// so besides dp, we will create dpPrev1 and dpPrev2

int[] dp = new int[2];

int[] dpPrev1 = new int[2];

int[] dpPrev2 = 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(dpPrev1[1], dpPrev1[0] - prices[i]);

} else {

profit = Math.max(dpPrev1[0], dpPrev2[1] + prices[i]);

}

dp[buy] = profit;

}

dpPrev2 = dpPrev1.clone();

dpPrev1 = dp.clone();

}

return dp[1];

}

}

Time Complexity:O(2\*3)

Space Complexity:O(2\*3)

**Solution 5: State Machine (60 min)**

**Style 1: Define s0, s1, s2 status as array**

class Solution {

public int maxProfit(int[] prices) {

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];

// s2 means status after sell, have to rest

int[] s2 = new int[n];

// s0 initial status as 0, no buy

s0[0] = 0;

// s1 initial status as after buy

s1[0] = -prices[0];

// s2 initial status as minimum, since s1 not trigger sell yet

// and default as minimum to make sure less than s1 initial

// status as -prices[0]

s2[0] = Integer.MIN\_VALUE;

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

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

// and s2 old status

s0[i] = Math.max(s0[i - 1], s2[i - 1]);

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

// and s0 old status but buy a stock

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

// s2 current status depends on max between s2 old status

// and s1 old status but sell a stock

s2[i] = Math.max(s2[i - 1], s1[i - 1] + prices[i]);

}

// The final max profit must between s0 and s2 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(at s2)

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

// the choice

return Math.max(s0[n - 1], s2[n - 1]);

}

}

Time Complexity:O(N)

Space Complexity:O(N)

**Style 2: Define s0, s1, s2 status as variable**

class Solution {

public int maxProfit(int[] prices) {

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];

// s2 means status after sell, have to rest, s2 initial status as minimum since s1 not trigger sell yet and default as minimum to make sure less than s1 initial status as -prices[0]

//int[] s2 = new int[n];

int s2 = Integer.MIN\_VALUE;

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

// Note: Look at the graph, the critical difference between

// s0 /s1 and s2 is s0 / s1 old status as 'Rest' is self-cycle

// which means s0 / s1 hold on themselves, s0 can be no buy,

// s1 can be no sell, not involve other status, but s2's old

// status is different than current status, its not self-cycle,

// as s1 + sell stock = s2 new status, its value changed hence

// we have to reserve its old value as last\_s2 in for loop

// s2 current status depends on max between s2 old status

// and s1 old status but sell a stock

//s2[i] = Math.max(s2[i - 1], s1[i - 1] + prices[i]);

int last\_s2 = s2;

s2 = s1 + prices[i];

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

// and s0 old status but buy a stock

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

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

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

// and s2 old status

//s0[i] = Math.max(s0[i - 1], s2[i - 1]);

s0 = Math.max(s0, last\_s2);

}

// The final max profit must between s0 and s2 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(at s2)

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

// the choice

//return Math.max(s0[n - 1], s2[n - 1]);

return Math.max(s0, s2);

}

}

Time Complexity:O(N)

Space Complexity:O(1)

**Refer to**

<https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-cooldown/solutions/3115114/tc-o-n-sc-const-o-6-recursive-memoization-tabulation-space-optimization/>

# **Intuition**

# **Complexity**

Time complexity: O(n) for Space Optimization Method

Space complexity: Constant O(6) for Space Optimization Method

# **Code**

\*\*-> RECURSION METHOD\*\*

class Solution {

public:

int f(int ind,int buy,vector<int>&price)

{

if(ind>=price.size()) return 0;

if(buy==1)

{

return max(-price[ind]+f(ind+1,0,price),0+f(ind+1,1,price));

}

return max(price[ind]+f(ind+2,1,price),0+f(ind+1,0,price));

}

int maxProfit(vector<int>& prices) {

return f(0,1,prices);

}

};

//----------------------------------------------------------------------------------------

\*\*-> MEMOIZATION METHOD\*\*

class Solution {

public:

int f(int ind,int buy,vector<int>&price,vector<vector<int>>&dp)

{

if(ind>=price.size()) return 0;

if(dp[ind][buy]!=-1) return dp[ind][buy];

if(buy==1)

{

return dp[ind][buy]=max(-price[ind]+f(ind+1,0,price,dp),0+f(ind+1,1,price,dp));

}

return dp[ind][buy]=max(price[ind]+f(ind+2,1,price,dp),0+f(ind+1,0,price,dp));

}

int maxProfit(vector<int>& prices) {

int n=prices.size();

vector<vector<int>>dp(n,vector<int>(2,-1));

return f(0,1,prices,dp);

}

};

//------------------------------------------------------------------------------------------

\*\*-> TABULATION METHOD\*\*

class Solution {

public:

int maxProfit(vector<int>& prices) {

int n=prices.size();

vector<vector<int>>dp(n+2,vector<int>(2,0));

for(int ind=n-1;ind>=0;ind--)

{

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

{

if(buy==1)

{

dp[ind][buy]=max(-prices[ind]+dp[ind+1][0],0+dp[ind+1][1]);

}

else

dp[ind][buy]=max(prices[ind]+dp[ind+2][1],0+dp[ind+1][0]);

}

}

return dp[0][1];

}

};

//-----------------------------------------------------------------------------------------------

\*\*-> MORE OPTIMIZED TABULATION METHOD\*\*

class Solution {

public:

int maxProfit(vector<int>& prices) {

int n=prices.size();

vector<vector<int>>dp(n+2,vector<int>(2,0));

for(int ind=n-1;ind>=0;ind--)

{

dp[ind][1]=max(-prices[ind]+dp[ind+1][0],0+dp[ind+1][1]);

dp[ind][0]=max(prices[ind]+dp[ind+2][1],0+dp[ind+1][0]);

}

return dp[0][1];

}

};

//------------------------------------------------------------------------------------------------

\*\*-> SPACE OPTIMIZATION USING 3 VECTORS\*\*

class Solution {

public:

int maxProfit(vector<int>& prices) {

int n=prices.size();

vector<int>front2(2,0);

vector<int>front1(2,0);

vector<int>curr(2,0);

for(int ind=n-1;ind>=0;ind--)

{

curr[1]=max(-prices[ind]+front1[0],0+front1[1]);

curr[0]=max(prices[ind]+front2[1],0+front1[0]);

front2=front1;

front1=curr;

}

return curr[1];

}

};

**Refer to**

<https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-cooldown/solutions/1570270/simple-bottom-up-dp-building-up-from-buy-sell-cooldown-states/>

int maxProfit(vector<int>& prices) {

int n=prices.size();

if(n<=1)

return 0;

// States -> 0-> unbought, 1-> bought

// and, 0 -> n+2 are the days.

vector<vector<int>> dp(2,vector<int>(n+2,0));

// taken size of n+2, because in case of getting

// sold at last index gets cooldown to last+2.

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

// for each day from last to front, (going bottom-up)

for(int i=0;i<2;i++){

// the first state of unbought then state of bought at

// ith index.

if(i==0){

dp[i][j]=max(dp[i][j+1],dp[i+1][j+1]-prices[j]);

// if state of unbought, 2 possible cases->

// (a) To skip this, hence call upon

// dp[i][j+1] , state unbought only,

// just get the dp answer of next day.

// (b) To include this, hence call upon

// dp[i+1][j+1] , as state is changed to 1

// (bought) now , and as bought it means

// what we deduct

// that amount from our profit, just like when

// u first invest in something.

}else{

dp[i][j]=max(dp[i][j+1],dp[i-1][j+2]+prices[j]); // plus 2 because of cooldown.

// if state of bought, 2 possible cases->

// (a) To skip this, hence call upon dp[i][j+1] ,

// state bought only, just get the dp answer of

// next day.

// (b) To include this, hence call upon

// dp[i+1][j+2] , as state is changed to 0

// (unbought) now , and

// as sold it means what we include this ith

// amount in our profit, just like when we get

// the returns

// after we sold to someone our investment.

}

}

}

return dp[0][0];

// finally return our answer which would be stored

// in our dp[0][0] , (first day and state unbought)

}

**State Machine Thinking**

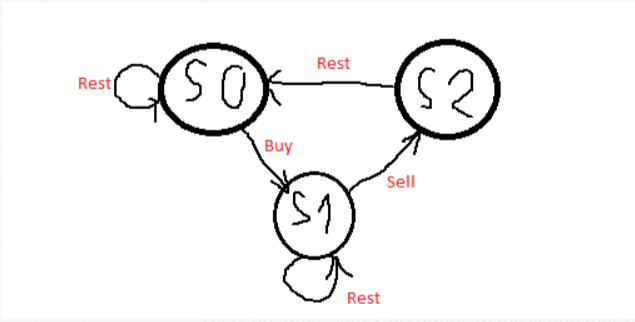
**Refer to**

<https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-cooldown/solutions/75928/share-my-dp-solution-by-state-machine-thinking/>

I just come across this problem, and it's very frustating since I'm bad at DP.

So I just draw all the actions that can be done.

Here is the drawing (Feel like an elementary ...)



There are three states, according to the action that you can take.

s0 means we can buy

s1 means we can sell

s2 means we take a rest, since we have to take a rest before we buy again, s1 can not go to s0 directly and have to go to s2 first.

Hence, from there, you can now the profit at a state at time i as:

s0[i] = max(s0[i - 1], s2[i - 1]); // Stay at s0, or rest from s2

s1[i] = max(s1[i - 1], s0[i - 1] - prices[i]); // Stay at s1, or buy from s0

s2[i] = s1[i - 1] + prices[i]; // Only one way from s1

Then, you just find the maximum of s0[n] and s2[n], since they will be the maximum profit we need (No one can buy stock and left with more profit that sell right :) )

Define base case:

s0[0] = 0; // At the start, you don't have any stock if you just rest

s1[0] = -prices[0]; // After buy, you should have -prices[0] profit. Be positive!

s2[0] = INT\_MIN; // Lower base case

Here is the code😄

class Solution {

public:

int maxProfit(vector<int>& prices){

if (prices.size() <= 1) return 0;

vector<int> s0(prices.size(), 0);

vector<int> s1(prices.size(), 0);

vector<int> s2(prices.size(), 0);

s1[0] = -prices[0];

s0[0] = 0;

s2[0] = INT\_MIN;

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

s0[i] = max(s0[i - 1], s2[i - 1]);

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

s2[i] = s1[i - 1] + prices[i];

}

return max(s0[prices.size() - 1], s2[prices.size() - 1]);

}

};

This is one of the best explanations for this kind of problems! As the state transitions only involve limited states and steps, we should be able to improve the space complexity to O(1):

int maxProfit(vector<int>& prices) {

int sold = 0, hold = INT\_MIN, rest = 0;

for (int i=0; i<prices.size(); ++i)

{

int prvSold = sold;

sold = hold + prices[i];

hold = max(hold, rest-prices[i]);

rest = max(rest, prvSold);

}

return max(sold, rest);

}

==============================================

int maxProfit(vector<int>& prices) {

if (prices.size() < 2) return 0;

int s0 = 0, s1 = -prices[0], s2 = 0;

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

int last\_s2 = s2;

s2 = s1 + prices[i];

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

s0 = max(s0, last\_s2);

}

return max(s0, s2);

}