<https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iii/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 **at most two transactions**.

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

**Example 1:**

Input: prices = [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: prices = [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: prices = [7,6,4,3,1]

Output: 0

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

**Constraints:**

* 1 <= prices.length <= 105
* 0 <= prices[i] <= 105

**Attempt 1: 2023-11-01**

**Solution 1:  Native DFS (60 min, TLE 201/204)**

**重点是只有在sell并完成的情况下才算完成了一整个transaction，才会导致limit减少1，在只完成buy的情况下并不算完成一整个transaction，不会导致limit减少1**

class Solution {

public int maxProfit(int[] prices) {

return helper(prices, 0, 1, 2);

}

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

if(index >= prices.length || limit <= 0) {

return 0;

}

int profit = 0;

// We need to respect the condition that we can’t buy a stock again,

// that is we need to first sell a stock, and then we can buy that

// again, which convert to the code condition is when buy it, since

// we cannot continue buy, the transaction limit won't decrease 1,

// only when sell it, we finish a full transaction (buy -> then sell),

// the transaction limit will decrease 1

if(buy == 1) {

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

// As we have only bought the stock and not sold it the transaction

// remains incomplete and the ‘cap’ variable value remains unchanged.

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

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

} else {

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

// As we have sold the earlier bought stock, we make one complete

// transaction, therefore now we update the 'limit' variable’s value

// to 'limit - 1'.

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

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

}

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][3];

return helper(prices, 0, 1, 2, memo);

}

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

if(index >= prices.length || limit <= 0) {

return 0;

}

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

return memo[index][buy][limit];

}

int profit = 0;

// We need to respect the condition that we can’t buy a stock again,

// that is we need to first sell a stock, and then we can buy that

// again, which convert to the code condition is when buy it, since

// we cannot continue buy, the transaction limit won't decrease 1,

// only when sell it, we finish a full transaction (buy -> then sell),

// the transaction limit will decrease 1

if(buy == 1) {

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

// As we have only bought the stock and not sold it the transaction

// remains incomplete and the ‘cap’ variable value remains unchanged.

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

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

} else {

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

// As we have sold the earlier bought stock, we make one complete

// transaction, therefore now we update the 'limit' variable’s value

// to 'limit - 1'.

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

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

}

return memo[index][buy][limit] = profit;

}

}

Time Complexity:O(2\*3\*N)

Space Complexity:O(2\*3\*N) + O(N)

**Solution 3: DP (10 min)**

class Solution {

public int maxProfit(int[] prices) {

int n = prices.length;

int[][][] dp = new int[n + 1][2][3];

// Bottom (base condition)

// n -> scan all days,

// 0 -> not able to buy,

// 0 -> no limits left

dp[n][0][0] = 0;

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

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

for(int limit = 1; limit <= 2; limit++) {

int profit = 0;

if(buy == 1) {

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

} else {

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

}

dp[i][buy][limit] = profit;

}

}

}

// Top

// 0 -> not scan any day (not start yet)

// 1 -> able to buy

// 2 -> start with 2 transactions limit

return dp[0][1][2];

}

}

Time Complexity:O(2\*3\*N)

Space Complexity:O(2\*3\*N)

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

class Solution {

public int maxProfit(int[] prices) {

int n = prices.length;

int[][] dpPrev = new int[2][3];

int[][] dp = new int[2][3];

// Bottom (base condition)

// 0 -> not able to buy,

// 0 -> no limits left

dp[0][0] = 0;

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

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

for(int limit = 1; limit <= 2; limit++) {

int profit = 0;

if(buy == 1) {

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

} else {

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

}

dp[buy][limit] = profit;

}

}

dpPrev = dp.clone();

}

// Top

// 1 -> able to buy

// 2 -> start with 2 transactions limit

return dpPrev[1][2];

}

}

Time Complexity:O(2\*3\*N)

Space Complexity:O(1)

**Solution 5:  One Pass (60 min)**

public class Solution {

public int maxProfit(int[] prices) {

int buy1 = Integer.MIN\_VALUE, sell1 = 0, buy2 = Integer.MIN\_VALUE, sell2 = 0;

for (int price : prices) {

buy1 = Math.max(buy1, -price);

sell1 = Math.max(sell1, buy1 + price);

buy2 = Math.max(buy2, sell1 - price);

sell2 = Math.max(sell2, buy2 + price);

}

return sell2;

}

}

Time Complexity:O(N)

Space Complexity:O(1)

**Refer to**

<https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iii/solutions/2391981/c-solution-with-explanation-representation-recursion-dp-space-optimisation/>

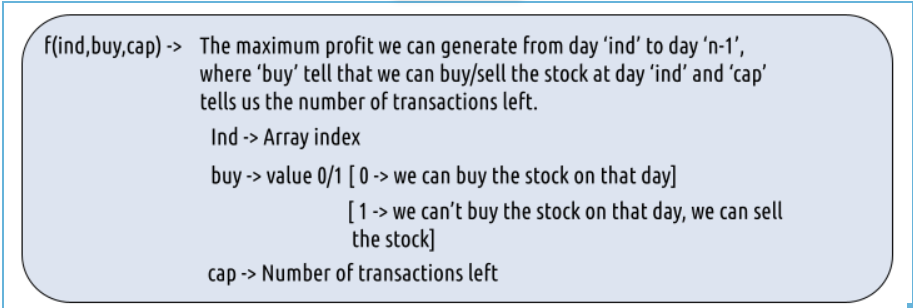
***Solution :***

Every day, we will have two choices, either to do nothing and move to the next day or to buy/sell (based on the last transaction and the number of transactions left) and find out the profit. Therefore we need to generate all the choices in order to compare the profit. As we need to try out all the possible choices, we will use recursion.

***Part 1: Steps to form the recursive solution:***

***Step 1:*** Express the problem in terms of indexes.

We need to think in the terms of the number of days, therefore one variable will be the array index( say ind). Next, we need to respect the condition that we can’t buy a stock again, that is we need to first sell a stock, and then we can buy that again. Therefore we need a second variable ‘buy’ which tells us on a particular day whether we can buy or sell the stock. Next, we have a cap on the number of transactions that we can make. Here the initial cap is 2. We need to always keep in mind this constraint. We can generalize the function as :



***Step 2:*** Try out all possible choices at a given index.

Every day, we have two choices:

* To either buy/sell the stock(based on the buy variable’s value and if ‘cap’ > 0).
* To do nothing and move on to the next day.

***Case 1:*** When buy == 0, we can buy the stock.

If we can buy the stock on a particular day, we have two options:

* Option 1: To do no transaction and move to the next day. In this case, the net profit earned will be 0 from the current transaction, and to calculate the maximum profit starting from the next day, we will recursively call f(ind+1,0, cap). As we have not bought the stock, the ‘buy’ variable value will still remain 0, indicating that we can buy the stock the next day. And the ‘cap’ variable will remain the same as if no transaction took place.
* Option 2: The other option is to buy the stock on the current day. In this case, the net profit earned from the current transaction will be -Arr[i]. As we are buying the stock, we are giving money out of our pocket, therefore the profit we earn is negative. To calculate the maximum profit starting from the next day, we will recursively call f(ind+1,1, cap). As we have bought the stock, the ‘buy’ variable value will change to 1, indicating that we can’t buy and only sell the stock the next day. As we have only bought the stock and not sold it the transaction remains incomplete and the ‘cap’ variable value remains unchanged.

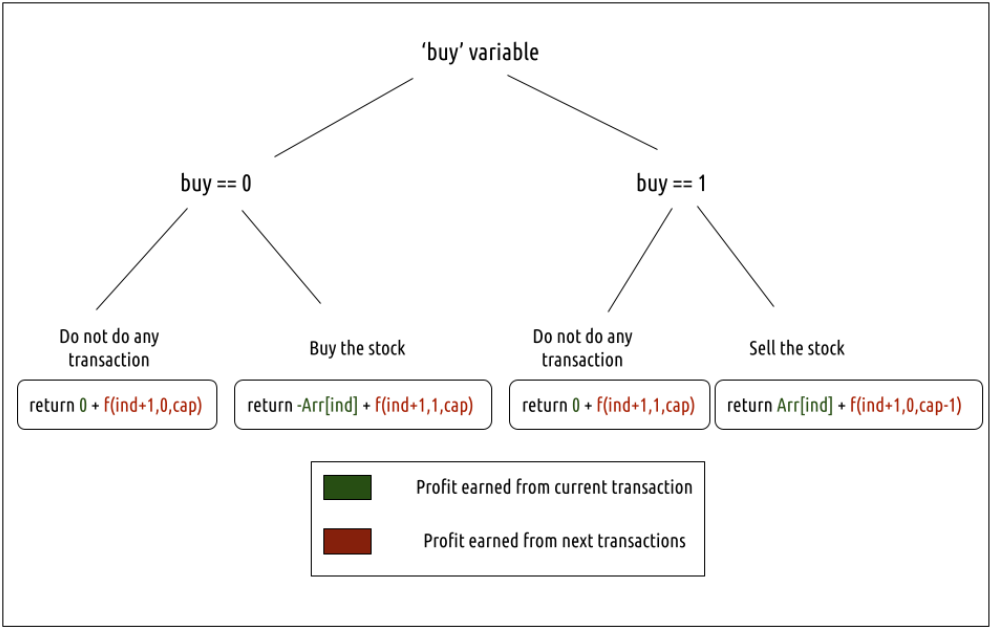
***Case 2:*** When buy == 1, we can sell the stock.

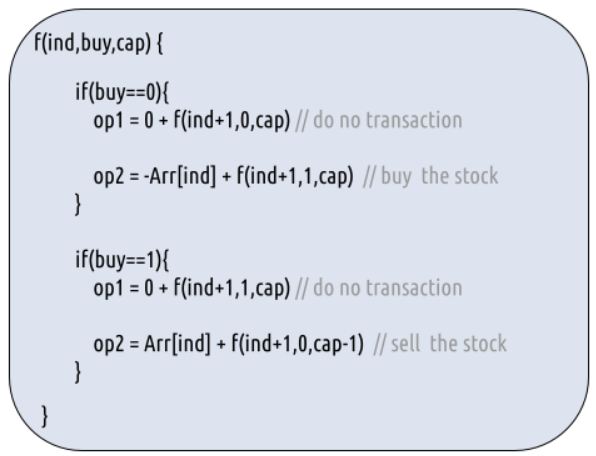
If we can buy the stock on a particular day, we have two options:

* Option 1: To do no transaction and move to the next day. In this case, the net profit earned will be 0 from the current transaction, and to calculate the maximum profit starting from the next day, we will recursively call f(ind+1,1, cap). As we have not bought the stock, the ‘buy’ variable value will still remain at 1, indicating that we can’t buy and only sell the stock the next day. And the ‘cap’ variable will remain the same as if no transaction took place.
* Option 2: The other option is to sell the stock on the current day. In this case, the net profit earned from the current transaction will be +Arr[i]. As we are selling the stock, we are putting the money into our pocket, therefore the profit we earn is positive. To calculate the maximum profit starting from the next day, we will recursively call f(ind+1,0,cap-1). As we have sold the stock, the ‘buy’ variable value will change to 0, indicating that we can buy the stock the next day. As we have sold the earlier bought stock, we make one complete transaction, therefore now we update the ‘cap’ variable’s value to cap-1.

***Note:*** Buying and selling a stock together counts as one complete transaction.

The figure below gives us the summary:





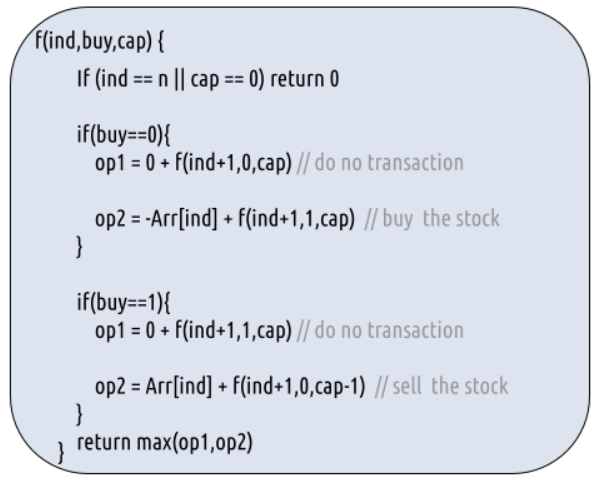
***Step 3:*** Return the maximum

As we are looking to maximize the profit earned, we will return the maximum value in both cases.

***Base Cases:***

* If ind==n, it means we have finished trading on all days, and there is no more money that we can get, therefore we simply return 0.
* If cap==0, it means that we cannot make any more transactions. Therefore we return 0.

The final pseudocode after steps 1, 2, and 3:



***Steps to memoize a recursive solution:***

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size [n][2][3]. The size of the input array is ‘n’, so the index will always lie between ‘0’ and ‘n-1’. The ‘buy’ variable can take only two values: 0 and 1 and the ‘cap’ variable can only take three variables 0, 1, and 2. Therefore we take the dp array as dp[n][2][3].
2. We initialize the dp array to -1.
3. Whenever we want to find the answer of particular parameters (say f(ind,buy,cap)), we first check whether the answer is already calculated using the dp array(i.e dp[ind][buy][cap]!= -1 ). If yes, simply return the value from the dp array.
4. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set dp[ind][buy][cap] to the solution we get.

***C++ Code:***

#include <bits/stdc++.h>

using namespace std;

int getAns(vector<int>& Arr, int n, int ind, int buy, int cap, vector<vector<vector<int>>>& dp ){

if(ind==n || cap==0) return 0; //base case

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

return dp[ind][buy][cap];

int profit;

if(buy==0){// We can buy the stock

profit = max(0+getAns(Arr,n,ind+1,0,cap,dp),

-Arr[ind] + getAns(Arr,n,ind+1,1,cap,dp));

}

if(buy==1){// We can sell the stock

profit = max(0+getAns(Arr,n,ind+1,1,cap,dp),

Arr[ind] + getAns(Arr,n,ind+1,0,cap-1,dp));

}

return dp[ind][buy][cap] = profit;

}

int maxProfit(vector<int>& prices, int n)

{

// Creating a 3d - dp of size [n][2][3]

vector<vector<vector<int>>> dp(n,

vector<vector<int>>

(2,vector<int>(3,-1)));

return getAns(prices,n,0,0,2,dp);

}

int main() {

vector<int> prices = {3,3,5,0,0,3,1,4};

int n = prices.size();

cout<<"The maximum profit that can be generated is "<<maxProfit(prices, n);

}

***Output:***

The maximum profit that can be generated is 6

***Time Complexity: O(Nx2x3)***

Reason: There are N*2*3 states therefore at max ‘N*2*3’ new problems will be solved.

***Space Complexity: O(Nx2x3) + O(N)***

Reason: We are using a recursion stack space(O(N)) and a 3D array ( O(N*2*3)).

***Part 2: Steps to convert Recursive Solution to Tabulation one.***

To convert the memoization approach to a tabulation one, create a dp array with the size [N+1][2][3].

***Handling the base case:***

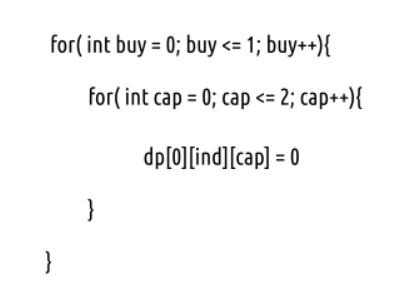
Now, what the base condition in the recursive relation is:

***if( ind == n || cap == 0) return 0***

We handle this in the following way:

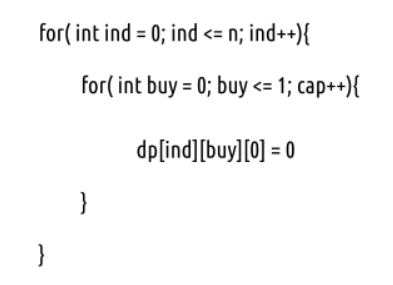
ind == n

When ind == n, the other two variables: cap and buy can take any value, therefore we can set the following two loops and set dp[n][buy][cap] = 0



cap == 0

When cap == 0, the other two variables: ind and cap can take any value, therefore we can set the following two loops and set dp[ind][buy][0] = 0.



Another hack is to initialize the entire 3D DP Array as 0. In this case, we need not worry about explicitly setting the base cases.

* First, we declare the dp array of size [n+1][2][3] as zero.
* As we have initialized the array as 0, we have automatically set the base condition as explained above.
* Now, traverse the array in the opposite direction of that of the memoization technique. We will start from ind = n-1 -> ind =0.
* In every iteration copy the recursive code logic.
* At last dp[0][0][2] ( maximum profit generated on ith day, when we can buy the stock on 0th day and can have a total 2 transactions) gives us the final answer.

***C++ Code:***

#include <bits/stdc++.h>

using namespace std;

int maxProfit(vector<int>& Arr, int n)

{

// Creating a 3d - dp of size [n+1][2][3] initialized to 0

vector<vector<vector<int>>> dp(n+1,

vector<vector<int>>

(2,vector<int>(3,0)));

// As dp array is intialized to 0, we have already covered the base case

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

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

for(int cap=1; cap<=2; cap++){

if(buy==0){// We can buy the stock

dp[ind][buy][cap] = max(0+dp[ind+1][0][cap],

-Arr[ind] + dp[ind+1][1][cap]);

}

if(buy==1){// We can sell the stock

dp[ind][buy][cap] = max(0+dp[ind+1][1][cap],

Arr[ind] + dp[ind+1][0][cap-1]);

}

}

}

}

return dp[0][0][2];

}

int main() {

vector<int> prices = {3,3,5,0,0,3,1,4};

int n = prices.size();

cout<<"The maximum profit that can be generated is "<<maxProfit(prices, n);

}

***Output:***

The maximum profit that can be generated is 6

***Time Complexity: O(Nx2x3)***

Reason: There are three nested loops that account for O(Nx2x3) complexity.

***Space Complexity: O(Nx2x3)***

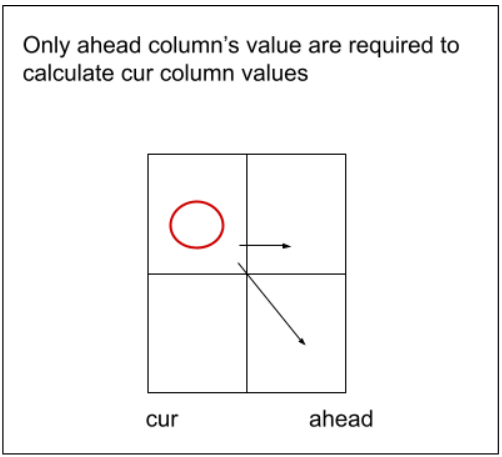
Reason: We are using an external array of size ‘Nx2x3’. Stack Space is eliminated.

***Part 3: Space Optimization***

If we closely look at the relation,

***dp[ind][buy][cap] = max( dp[ind+1][buy][cap] , max( dp[ind+1][!buy][cap])***

We see that to calculate a value of a cell of the dp array, we need only the next row values(say ahead of ind+1). So, we don’t need to store an entire 2-D array. Hence we can space optimize it.



* We set a 2D vector ahead initialized to 0 (base condition) and another 2D
* Then we set three nested loops to calculate the cur array’s values.
* We replace dp[ind] with cur and dp[ind+1] with ahead in our tabulation code.
* After the inner loop execution, we set ahead as cur for the next outer loop iteration.
* At last, we return ahead[0][2] as our answer.

***C++ Code:***

#include <bits/stdc++.h>

using namespace std;

int maxProfit(vector<int>& Arr, int n)

{

vector<vector<int>> ahead(2,vector<int> (3,0));

vector<vector<int>> cur(2,vector<int> (3,0));

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

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

for(int cap=1; cap<=2; cap++){

if(buy==0){// We can buy the stock

cur[buy][cap] = max(0+ahead[0][cap],

-Arr[ind] + ahead[1][cap]);

}

if(buy==1){// We can sell the stock

cur[buy][cap] = max(0+ahead[1][cap],

Arr[ind] + ahead[0][cap-1]);

}

}

}

ahead = cur;

}

return ahead[0][2];

}

int main() {

vector<int> prices = {3,3,5,0,0,3,1,4};

int n = prices.size();

cout<<"The maximum profit that can be generated is "<<maxProfit(prices, n);

}

***Output:***

The maximum profit that can be generated is 6

***Time Complexity: O(Nx2x3)***

Reason: There are three nested loops that account for O(Nx2x3) complexity.

***Space Complexity: O(1)***

Reason: We are using two external arrays of size ‘2x3’.

**Refer to**

<https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iii/solutions/3529289/c-3d-dp-recursion-memoization-tabulation-space-optimization/>

➡️ Recursion

int solve(int index,int buy,int limits,vector<int>&prices){

if(index==prices.size()) return 0;

if(limits==0) return 0;

int profit=0;

if(buy){

int sellkaro=(-prices[index])+solve(index+1,0,limits,prices);

int skipkaro=0+solve(index+1,1,limits,prices);

profit=max(sellkaro,skipkaro);

}else{

int buykaro=(prices[index])+solve(index+1,1,limits-1,prices);

int skipkaro=0+solve(index+1,0,limits,prices);

profit=max(buykaro,skipkaro);

}

return profit;

}

int maxProfit(vector<int>& prices) {

return solve(0,1,2,prices);

}

➡️ Memoization

int solveMem(int index,int buy,int limits,vector<int>&prices,vector<vector<vector<int>>>&dp){

if(index==prices.size()) return 0;

if(limits==0) return 0;

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

int profit=0;

if(buy){

int sellkaro=(-prices[index])+solveMem(index+1,0,limits,prices,dp);

int skipkaro=0+solveMem(index+1,1,limits,prices,dp);

profit=max(sellkaro,skipkaro);

}else{

int buykaro=(prices[index])+solveMem(index+1,1,limits-1,prices,dp);

int skipkaro=0+solveMem(index+1,0,limits,prices,dp);

profit=max(buykaro,skipkaro);

}

return dp[index][buy][limits]=profit;

}

int maxProfit(vector<int>& prices) {

int n=prices.size();

vector<vector<vector<int>>>dp(n,vector<vector<int>>(2,vector<int>(3,-1)));

return solveMem(0,1,2,prices,dp);

}

➡️ Tabulation

int solveTab(vector<int>& prices){

int n=prices.size();

vector<vector<vector<int>>>dp(n+1,vector<vector<int>>(2,vector<int>(3,0)));

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

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

for(int limits=1;limits<=2;limits++){

int profit=0;

if(buy){

int sellkaro=(-prices[index])+dp[index+1][0][limits];

int skipkaro=0+dp[index+1][1][limits];

profit=max(sellkaro,skipkaro);

}else{

int buykaro=(prices[index])+dp[index+1][1][limits-1];

int skipkaro=0+dp[index+1][0][limits];

profit=max(buykaro,skipkaro);

}

dp[index][buy][limits]=profit;

}

}

}

return dp[0][1][2];

}

int maxProfit(vector<int>& prices) {

return solveTab(prices);

}

➡️ Space Optimization

int solveOpt(vector<int>& prices){

int n=prices.size();

//vector<vector<vector<int>>>dp(n+1,vector<vector<int>>(2,vector<int>(3,0)));

vector<vector<int>>curr(2,vector<int>(3,0));

vector<vector<int>>next(2,vector<int>(3,0));

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

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

for(int limits=1;limits<=2;limits++){

int profit=0;

if(buy){

int sellkaro=(-prices[index])+next[0][limits];

int skipkaro=0+next[1][limits];

profit=max(sellkaro,skipkaro);

}else{

int buykaro=(prices[index])+next[1][limits-1];

int skipkaro=0+next[0][limits];

profit=max(buykaro,skipkaro);

}

curr[buy][limits]=profit;

}

}

next=curr;

}

return next[1][2];

}

int maxProfit(vector<int>& prices) {

return solveOpt(prices);

}

**Refer to**

<https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iii/solutions/39611/is-it-best-solution-with-o-n-o-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 just sold 2nd stock so far.

hold2 = Math.max(hold2, release1-i); // The maximum if we've just buy 2nd stock so far.

release1 = Math.max(release1, hold1+i); // The maximum if we've just sold 1nd stock so far.

hold1 = Math.max(hold1, -i); // The maximum if we've just buy 1st stock so far.

}

return release2; ///Since release1 is initiated as 0, so release2 will always higher than release1.

}

}

<https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iii/solutions/39611/is-it-best-solution-with-o-n-o-1/comments/242914>

It really took me hours to completely understand this. The comment really helps me. Now I share my understanding.

var maxProfit = function(prices) {

let oneBuyOneSell = 0;

let twoBuyTwoSell = 0;

let oneBuy = Number.POSITIVE\_INFINITY

let twoBuy = Number.POSITIVE\_INFINITY;

for(let i = 0; i < prices.length; i++) {

const p = prices[i];

oneBuy = Math.min(oneBuy, p);

oneBuyOneSell = Math.max(oneBuyOneSell, p - oneBuy);

twoBuy = Math.min(twoBuy, p - oneBuyOneSell);

twoBuyTwoSell = Math.max(twoBuyTwoSell, p - twoBuy);

}

return twoBuyTwoSell;

};

Similar to Buy and Sell Stock I, where only one transaction allowed, we keep track the lowest price and once the price exceeds the old lowest price, we record the difference.

var maxProfit = function(prices) {

const size = prices.length;

let lowPrice = prices[0];

let profit = 0;

for(let i = 1; i < size; i++) {

if(prices[i] < lowPrice) {

lowPrice = prices[i];

} else {

profit = Math.max(prices[i] - lowPrice, profit);

}

}

return profit;

};

Here, the oneBuykeeps track of the lowest price, and oneBuyOneSellkeeps track of the biggest profit we could get.  
Then the tricky part comes, how to handle the twoBuy? Suppose in real life, you have bought and sold a stock and made 100dollarprofit.Whenyouwanttopurchaseastockwhichcostsyou100 dollar profit. When you want to purchase a stock which costs you100*dollarprofit*.*Whenyouwanttopurchaseastockwhichcostsyou*300 dollars, how would you think this? You must think, um, I have made 100profit,soIthinkthis100 profit, so I think this100*profit*,*soIthinkthis*300 dollar stock is worth $200FOR MEsince I have hold $100 for free.  
There we go, you got the idea how we calculate twoBuy!! We just minimize the cost again!! The twoBuyTwoSellis just making as much profit as possible.