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

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

On each day, you may decide to buy and/or sell the stock. You can only hold **at most one** share of the stock at any time. However, you can buy it then immediately sell it on the **same day**.

Find and return *the* ***maximum*** *profit you can achieve*.

**Example 1:**

Input: prices = [7,1,5,3,6,4]

Output: 7

Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.

Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.

Total profit is 4 + 3 = 7.

**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.

Total profit is 4.

**Example 3:**

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

Output: 0

Explanation: There is no way to make a positive profit, so we never buy the stock to achieve the maximum profit of 0.

**Constraints:**

1 <= prices.length <= 3 \* 104

0 <= prices[i] <= 104

**Attempt 1: 2023-10-30**

**Solution 1: Native DFS (30 min, TLE 198/200)**

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, buy);

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

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

} else {

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

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

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

}

return profit;

}

}

Time Complexity:O(2^n)

Space Complexity:O(n)

**Solution 2: DFS + Memoization (30 min)**

**Style 1: flag 'buy' as 1 & -1**

class Solution {

public int maxProfit(int[] prices) {

Integer[][] memo = new Integer[prices.length + 1][3];

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

}

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

if(index >= prices.length) {

return 0;

}

// 'buy + 1' just to make sure its not negative index

// to match the 2D memo array definition

// actually we can change the way to assign flag 'buy' as

// pure non-negative value as 0 & 1 status, currently

// the -1 & 1 for 'buy' cause the issue and require 'buy + 1'

if(memo[index][buy + 1] != null) {

return memo[index][buy + 1];

}

int profit = 0;

if(buy == 1) {

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

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

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

} else {

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

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

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

}

return memo[index][buy + 1] = profit;

}

}

Time Complexity:O(2\*N)

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

**Style 2: flag 'buy' as 1 & 0**

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);

int sell\_it = helper(prices, index + 1, 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 (30 min)**

class Solution {

public int maxProfit(int[] prices) {

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]);

}

dp[i][buy] = profit;

}

}

return dp[0][1];

}

}

Time Complexity:O(N)

Space Complexity:O(2\*N)

**Solution 4: DP + Space Optimization (30 min)**

class Solution {

public int maxProfit(int[] prices) {

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]);

}

dp[buy] = profit;

}

dpPrev = dp.clone();

}

return dpPrev[1];

}

}

Time Complexity:O(2\*N)

Space Complexity:O(2\*2)

**Solution 5: One Pass + Target at all monotone increasing sequences (30 min)**

**Style 1: Find valleys + peaks**

class Solution {

public int maxProfit(int[] prices) {

// buy: the day when buy stock.

// sell: the day when sell stock.

int profit = 0;

int buy = 0;

int sell = 0;

int n = prices.length;

while(buy < n && sell < n) {

// find the valley of prices

while(buy < n - 1 && prices[buy] > prices[buy + 1]) {

buy++;

}

sell = buy;

// find the peak of prices

while(sell < n - 1 && prices[sell] < prices[sell + 1]) {

sell++;

}

profit += prices[sell] - prices[buy];

buy = sell + 1;

}

return profit;

}

}

Time Complexity:O(N)

Space Complexity:O(1)

**Style 2: Accumulate all sections on all monotone increasing sequences**

class Solution {

public int maxProfit(int[] prices) {

int result = 0;

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

if(prices[i] >= prices[i - 1]) {

result += prices[i] - prices[i - 1];

}

}

return result;

}

}

Time Complexity:O(N)

Space Complexity:O(1)

**Refer to**

<https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii/solutions/2423189/c-intuition-memoization-tabulation-1d-array/>

Intuition - ***When we are at (i) th stock, we have 2 states possible*** -

We have bought a stock before and have not sold it. {We cannot buy a new stock right now, we can only sell or not sell today}. -> **2 options -> [ sell, not sell]**

We have no stock bought right now. {We can buy a new stock right now. we may or may not buy today}. -> **2 options -> [buy, not buy]**

We can keep a boolean variable buy, which will track ***whether we can buy on (i)th day or not.***

When buy = true, we can buy or not buy a new stock.

When buy = false, we can sell or not sell the current stock we have.

We also have an integer variable - idx to ***track which day we're at***. If idx == prices.size(), we will earn 0 profit on that day, as we can neither buy nor sell.

When we ***buy*** a new stock on day idx - our ***net profit decreases by prices[idx]*** , as the **price of the stock on this day will be deducted from the net profit earned.**

Similarly, when we ***sell*** a new stock on day idx - our ***net profit increases by prices[idx]*** , as the **price of the stock on this day will be added to the net profit earned.**

Initially, we are at day 0 and we can buy a stock.

Let's look at the code -

Note -

nb = net profit earned if we don't buy on day idx.

ns = net profit earned if we don't sell on day idx.

b = net profit earned if we buy on day idx.

s = net profit earned if we sell on day idx.

Recursion

class Solution {

public:

int f(vector<int> &p, int idx, bool buy){

int n = p.size();

if(idx == n) return 0; // end of days. Can't buy or sell

int ans = 0;

if(buy){

// either buy or don't buy

int b = -p[idx] + f(p, idx+1, 0);

int nb = 0 + f(p, idx+1, 1);

ans = max(b, nb);

}

else{

// either sell or don't sell

int s = p[idx] + f(p, idx+1, 1);

int ns = 0 + f(p, idx+1, 0);

ans = max(s, ns);

}

return ans;

}

int maxProfit(vector<int>& prices) {

// Initially, we are at day 0 and we can buy a stock.

return f(prices, 0, 1);

}

};

TC -> O(2 ^ N) where N - number of days

SC -> O(N)

We can ***reduce time complexity***, as there are overlapping subproblems that can be ***memoized***.

Memoization -

class Solution {

public:

int f(vector<int> &p, int idx, bool buy, vector<vector<int>> &dp){

int n = p.size();

if(idx == n) return 0;

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

int ans = 0;

if(buy){

int b = -p[idx] + f(p, idx+1, 0, dp);

int nb = 0 + f(p, idx+1, 1, dp);

ans = max(b, nb);

}

else{

int s = p[idx] + f(p, idx+1, 1, dp);

int ns = 0 + f(p, idx+1, 0, dp);

ans = max(s, ns);

}

return dp[idx][buy] = ans;

}

int maxProfit(vector<int>& prices) {

vector<vector<int>> dp(prices.size(), vector<int>(2, -1));

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

}

};

TC -> O(2 \* N)

SC -> O(2 \* N) + O(N)

This ***extra O(N) stack space*** can be avoided by tabulating the above code.

Tabulation( Top - Down) -

class Solution {

public:

int maxProfit(vector<int>& p) {

int n = p.size();

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

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

int b = -p[idx] + dp[idx+1][0];

int nb = 0 + dp[idx+1][1];

int s = p[idx] + dp[idx+1][1];

int ns = 0 + dp[idx+1][0];

dp[idx][0] = max(s, ns);

dp[idx][1] = max(b, nb);

}

return dp[0][1];

}

};

TC -> O( N )

SC -> O(2 \* N)

Now, note that we just need 2 info , to calculate max profit for each day idx -> max profit if we buy/not buy on idx+1 day and max profit if we sell/not sell on idx + 1 day.

So , we just need 2 array -> curr and next. next stores the 2 profits( can buy , can sell) for the idx + 1 day and curr stores the 2 profits for idx day.

1D array Optimized Code -

class Solution {

public:

int maxProfit(vector<int>& p) {

int n = p.size();

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

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

int b = -p[idx] + next[0];

int nb = 0 + next[1];

int s = p[idx] + next[1];

int ns = 0 + next[0];

curr[0] = max(s, ns);

curr[1] = max(b, nb);

next = curr;

}

return curr[1];

}

};

TC -> O( N )

SC -> O(1)

**Refer to**

<https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii/solutions/2490167/c-recursion-memoization-tabulation-spaceoptimization/>

# **Method -1 [Recursion]**

**T->O(2^n) && S->O(n) [Recursion Stack Space]**

class Solution {

public:

int f(int i,int buy,vector<int>& p,int n){

if(i==n) return 0;

if(buy){

int buy=-p[i]+f(i+1,0,p,n);

int notbuy=f(i+1,1,p,n);

return max(buy,notbuy);

}

else{

int sell= p[i]+f(i+1,1,p,n);

int notsell=f(i+1,0,p,n);

return max(sell,notsell);

}

}

int maxProfit(vector<int>& p){

int n=p.size();

return f(0,1,p,n);

}

};

# **Method -2 [Memoization]**

**T->O(2 \* n) && S->O(2 \* n) + O(n) [Recursion Stack Space]**

class Solution {

public:

int f(int i,int buy,vector<int>& p,int n,vector<vector<int>>& dp){

if(i==n) return 0;

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

int pro1=INT\_MIN,pro2=INT\_MIN;

if(buy){

int buy=-p[i]+f(i+1,0,p,n,dp);

int notbuy=f(i+1,1,p,n,dp);

pro1=max(buy,notbuy);

}

else{

int sell= p[i]+f(i+1,1,p,n,dp);

int notsell=f(i+1,0,p,n,dp);

pro2=max(sell,notsell);

}

return dp[i][buy]=max(pro1,pro2);

}

int maxProfit(vector<int>& p){

int n=p.size();

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

return f(0,1,p,n,dp);

}

};

# **Method - 3 [Tabulation]**

**T->O(2 \* n) && S->O(2 \* n)**

class Solution {

public:

int maxProfit(vector<int>& p){

int n=p.size();

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

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

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

int profit;

if(buy){

int buy=-p[i]+dp[i+1][0];

int notbuy=dp[i+1][1];

profit=max(buy,notbuy);

}

else{

int sell= p[i]+dp[i+1][1];

int notsell=dp[i+1][0];

profit=max(sell,notsell);

}

dp[i][buy]=profit;

}

}

return dp[0][1];

}

};

# **Method - 4 [SpaceOptimization]**

**T->O(2 X n) && S->O(2 X 2)**

class Solution {

public:

int maxProfit(vector<int>& p){

int n=p.size();

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

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

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

int profit;

if(buy){

int buy=-p[i]+prev[0];

int notbuy=prev[1];

profit=max(buy,notbuy);

}

else{

int sell= p[i]+prev[1];

int notsell=prev[0];

profit=max(sell,notsell);

}

curr[buy]=profit;

}

prev=curr;

}

return prev[1];

}

};

**Refer to**

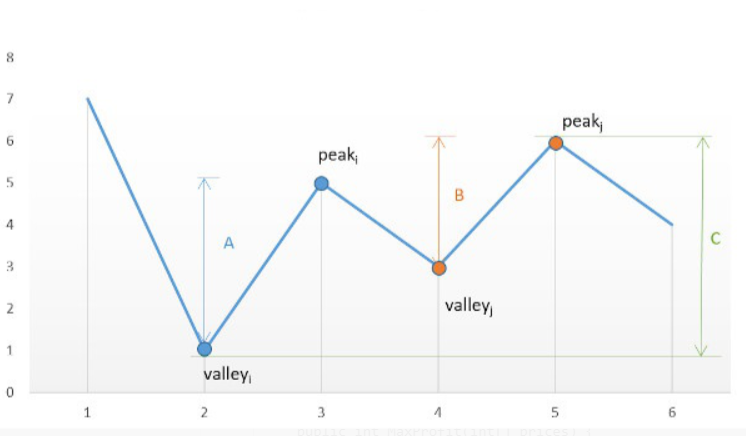
<https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii/solutions/208241/explanation-for-the-dummy-like-me/comments/477661>

The general idea is to buy stock at valleys (lowest price) and then sell stock at its next peak (highest price). In the below graph, there are two valleys and two peaks. Since (1) we have to buy stock before selling it and (2) we can not have back-to-back buy or sell trades, thus there are two options of trading stocks.

Option 1: Buy stock at valley[i] then sell at peak[i] makes profit **A** (peak[i] - valley[i]). Then buy stock at valley [j] and sell at peak[j] makes proft **B** (peak[j] - valley[j]). So the total profit of this trade option is **A + B.**

Option 2: Skip the intermediate trades, i.e,, we buy stock at valley[i] then sell at peak[j]. In this case, the total profit will be **C** (peak[j]-valley[i]).

Based on the graph shown below, **A + B > C** (if not, peak[i] and valley[j] won't exist). So in order to maximize the profit, we can **buy stock at valleys and then sell stock at peaks**.



**Implementation**

public class Solution {

public int MaxProfit(int[] prices) {

/\*

profit: maximum total profit.

buy: the day when buy stock.

sell: the day when sell stock.

days: maximum trade days.

\*/

int profit = 0, buy = 0, sell = 0, days = prices.Length;

while(buy < days && sell < days)

{

while(buy < days - 1 && prices[buy + 1] < prices[buy])

buy++; // find the valley of prices

sell = buy;

while(sell < days - 1 && prices[sell + 1] > prices[sell])

sell++; // find the peak of prices

profit += prices[sell] - prices[buy];

buy = sell + 1;

}

return profit;

}

}

**Refer to**

<https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii/solutions/39420/three-lines-in-c-with-explanation/>

First we post the code here.

int maxProfit(vector<int> &prices) {

int ret = 0;

for (size\_t p = 1; p < prices.size(); ++p)

ret += max(prices[p] - prices[p - 1], 0);

return ret;

}

Second, suppose the first sequence is "a <= b <= c <= d", the profit is "d - a = (b - a) + (c - b) + (d - c)" without a doubt. And suppose another one is "a <= b >= b' <= c <= d", the profit is not difficult to be figured out as "(b - a) + (d - b')". So you just target at all monotone increasing sequences.