Best Time to Buy and Sell Stock

Read | Courses | Practice

Given an array **prices[]** of length **N**, representing the prices of the stocks on different days, the task is to find the maximum profit possible by buying and selling the stocks on different days when at most one transaction is allowed.

Note: *Stock must be bought before being sold.*

Examples:

Input: *prices[] = {7, 1, 5, 3, 6, 4}*
Output: *5*
Explanation:
The lowest price of the stock is on the 2^{nd} day, i.e. price = 1. Starting from the 2nd day, the highest price of the stock is witnessed on the 5^{th} day, i.e. price = 6.
Therefore, maximum possible profit = 6 – 1 = 5.

Input: *prices[] = {7, 6, 4, 3, 1}*
Output: *0*
Explanation: *Since the array is in decreasing order, no possible way exists to solve the problem.*

Best Time to Buy and Sell Stock using Greedy Approach:

*In order to maximize the profit, we have to **minimize** the cost price and **maximize** the selling price. So at every step, we will keep track of the **minimum buy price** of stock encountered so far. If the current price of stock is lower than the previous buy price, then we will update the buy*

price then we can sell at this price to get some profit. After iterating over the entire array, return the maximum profit.

Follow the steps below to implement the above idea:

- Declare a **buy** variable to store the min stock price encountered so far and **max_profit** to store the maximum profit.
- Initialize the **buy** variable to the first element of the **prices array**.
- Iterate over the **prices** array and check if the current price is less than buy price or not.
  - If the current price is **smaller** than buy price, then buy on this **ith** day.
  - If the current price is **greater** than buy price, then make profit from it and maximize the **max_profit.**
- Finally, return the **max_profit**.

Below is the implementation of the above approach:

C++

```
// C++ code for the above approach
#include <iostream>
using namespace std;

int maxProfit(int prices[], int n)
{
    int buy = prices[0], max_profit = 0;
    for (int i = 1; i < n; i++) {

        // Checking for lower buy value
        if (buy > prices[i])
            buy = prices[i];

        // Checking for higher profit
        else if (prices[i] - buy > max_profit)
            max_profit = prices[i] - buy;
    }
    return max_profit;
}

// Driver Code
int main()
```

```
    cout << max_profit << endl;
    return 0;
}
```

Java

```
// Java code for the above approach
class GFG {
    static int maxProfit(int prices[], int n)
    {
        int buy = prices[0], max_profit = 0;
        for (int i = 1; i < n; i++) {

            // Checking for lower buy value
            if (buy > prices[i])
                buy = prices[i];

            // Checking for higher profit
            else if (prices[i] - buy > max_profit)
                max_profit = prices[i] - buy;
        }
        return max_profit;
    }

    // Driver Code
    public static void main(String args[])
    {
        int prices[] = { 7, 1, 5, 6, 4 };
        int n = prices.length;
        int max_profit = maxProfit(prices, n);
        System.out.println(max_profit);
    }
}

// This code is contributed by Lovely Jain
```

Python3

```
# Python program for the above approach:


def maxProfit(prices, n):
    buy = prices[0]
    max_profit = 0
    for i in range(1, n):
```

```
        # Checking for higher profit
        elif (prices[i] - buy > max_profit):
            max_profit = prices[i] - buy
    return max_profit


# Driver code
if __name__ == '__main__':

    prices = [7, 1, 5, 6, 4]
    n = len(prices)
    max_profit = maxProfit(prices, n)
    print(max_profit)
```

C#

```
// C# code for the above approach
using System;
public class GFG {

    static int maxProfit(int[] prices, int n)
    {
        int buy = prices[0], max_profit = 0;
        for (int i = 1; i < n; i++) {

            // Checking for lower buy value
            if (buy > prices[i])
                buy = prices[i];

            // Checking for higher profit
            else if (prices[i] - buy > max_profit)
                max_profit = prices[i] - buy;
        }
        return max_profit;
    }

    static public void Main()
    {

        // Code
        int[] prices = { 7, 1, 5, 6, 4 };
        int n = prices.Length;
        int max_profit = maxProfit(prices, n);
        Console.WriteLine(max_profit);
    }
}

// This code is contributed by lokeshmvs21
```

```
function maxProfit( prices, n)
{
    let buy = prices[0], max_profit = 0;
    for (let i = 1; i < n; i++) {

        // Checking for lower buy value
        if (buy > prices[i])
            buy = prices[i];

        // Checking for higher profit
        else if (prices[i] - buy > max_profit)
            max_profit = prices[i] - buy;
    }
    return max_profit;
}

// Driver Code

    let prices= [ 7, 1, 5, 6, 4 ];
    let n =5;
    let max_profit = maxProfit(prices, n);
    console.log(max_profit);

    // This code is contributed by garg28harsh.
```

Output

```
5
```

Time Complexity: O(N). Where **N** is the size of prices array.

Auxiliary Space: O(1)

Best Time to Buy and Sell Stock using Recursion and

*We can define a recursive function **maxProfit(idx, canSell)** which will return us the maximum profit if the user can buy or sell starting from **idx**.*

- *If idx == N, then return 0 as we have reached the end of the array*
- *If canSell == false, then we can only **buy** at this index. So, we can explore both the choices and return the maximum:*
  - *buy at the current price, so the profit will be: **-prices[idx] + maxProfit(idx+1, true)***
  - *move forward without buying, so the profit will be: **maxProfit(idx+1, false)***
- *If canSell == true, then we can only **sell** at this index. So, we can explore both the choices and return the maximum:*
  - *sell at the current price, so the profit will be: **prices[idx]***
  - *move forward without selling, so the profit will be: **maxProfit(idx+1, true)***

Below is the implementation of the approach:

C++

```
#include <bits/stdc++.h>
using namespace std;

// function to calculate the max profit
int maxProfit(int idx, vector<int>& prices, bool canSell)
{
    // We have reached the end of array
    if (idx == prices.size())
        return 0;
    if (canSell) {
        // We can only sell the stock
        return max(prices[idx],
                   maxProfit(idx + 1, prices, canSell));
    }
    else {
        // We can only buy the stock
        return max( prices[idx]
```

```
}

int main()
{
    vector<int> prices{ 7, 1, 5, 3, 6, 4 };
    cout << maxProfit(0, prices, false) << "\n";
    return 0;
}
```

Java

```
// Java code of the above approach
import java.util.*;

public class GFG {
    // Function to calculate the max profit
    static int maxProfit(int idx, List<Integer> prices,
                         boolean canSell)
    {
        // We have reached the end of the array
        if (idx == prices.size())
            return 0;
        if (canSell) {
            // We can only sell the stock
            return Math.max(
                prices.get(idx),
                maxProfit(idx + 1, prices, canSell));
        }
        else {
            // We can only buy the stock
            return Math.max(
                -prices.get(idx)
                    + maxProfit(idx + 1, prices, true),
                maxProfit(idx + 1, prices, canSell));
        }
    }

    public static void main(String[] args)
    {
        List<Integer> prices = new ArrayList<>(
            Arrays.asList(7, 1, 5, 3, 6, 4));
        System.out.println(maxProfit(0, prices, false));
    }
}


// This code is contributed by Susobhan Akhuli
```

```
# Python code of the above approach

# Function to calculate the max profit
def max_profit(idx, prices, can_sell):
    # We have reached the end of array
    if idx == len(prices):
        return 0
    if can_sell:
        # We can only sell the stock
        return max(prices[idx], max_profit(idx + 1, prices, can_sell))
    else:
        # We can only buy the stock
        return max(-prices[idx] + max_profit(idx + 1, prices, True),
                   max_profit(idx + 1, prices, can_sell))

if __name__ == "__main__":
    prices = [7, 1, 5, 3, 6, 4]
    print(max_profit(0, prices, False))

# This code is contributed by Susobhan Akhuli
```

C#

```
// C# Implementation
using System;
using System.Collections.Generic;

public class Program
{
    public static int MaxProfit(int idx, List<int> prices, bool canSell)
    {
        // We have reached the end of the list
        if (idx == prices.Count)
            return 0;

        if (canSell)
        {
            // We can only sell the stock
            return Math.Max(prices[idx], MaxProfit(idx + 1, prices, canSell));
        }
        else
        {
            // We can only buy the stock
            return Math.Max(-prices[idx] + MaxProfit(idx + 1, prices, true), Ma
        }
    }
```

```
        Console.WriteLine(MaxProfit(0, prices, false));
    }
}

// This code is contributed by Sakshi
```

Javascript

```
// JavaScript program for the above approach

// Function to calculate the max profit
function maxProfit(idx, prices, canSell) {
    // We have reached the end of the array
    if (idx === prices.length) {
        return 0;
    }

    if (canSell) {
        // We can only sell the stock
        return Math.max(prices[idx], maxProfit(idx + 1, prices, canSell));
    } else {
        // We can only buy the stock
        return Math.max(
            -prices[idx] + maxProfit(idx + 1, prices, true),
            maxProfit(idx + 1, prices, canSell)
        );
    }
}

// Driver code
const prices = [7, 1, 5, 3, 6, 4];
console.log(maxProfit(0, prices, false)); // Output the max profit

// This code is contributed by Susobhan Akhuli
```

Output

```
5
```

Time Complexity: $O(2^n)$, where n is the size of input array prices[]
Auxiliary Space: O(N)

Best Time to Buy and Sell Stock using Dynamic Programming:

> *We can optimize the recursive approach by storing the states in a 2D dp array of size NX2. Here, dp[idx][0] will store the answer of maxProfit(idx, false) and dp[idx][1] will store the answer of maxProfit(idx, true).*

Below is the implementation of the approach:

C++

```
// C++ code for the approach

#include <bits/stdc++.h>
using namespace std;

// Function to find the maximum
// profit with atmost 1 transaction
int maxProfit(vector<int>& prices)
{
    int n = prices.size();

    // 2D DP array to store max profit with 0 and 1 stocks
    vector<vector<int> > dp(n, vector<int>(2));

    dp[0][0] = -prices[0];
    dp[0][1] = 0;

    // Loop through prices to calculate max profit at each
    // day
    for (int i = 1; i < n; i++) {
        // choice 1: Buy the stock at i, in which case the
        // profit we get is the maximum profit we could have
        // made till i-1 minus the price at i.
```

```
        // made till i-1 by buying the stock earlier plus
        // the price at i.
        dp[i][1]
            = max(dp[i - 1][1], dp[i - 1][0] + prices[i]);
    }

    // Return the maximum profit calculated from the last
    // day
    return max(dp.back()[0], dp.back()[1]);
}

// Driver's code
int main()
{
    // Given prices
    vector<int> prices = { 7, 1, 5, 3, 6, 4 };

    // Function Call
    int ans = maxProfit(prices);

    // Print answer
    cout << ans << endl;
    return 0;
}
```

Java

```
// Java code for the approach
import java.util.*;

public class GFG {
    public static int maxProfit(List<Integer> prices)
    {
        int n = prices.size();

        // 2D DP array to store max profit with 0 and 1
        // stocks
        int[][] dp = new int[n][2];

        dp[0][0] = -prices.get(0);
        dp[0][1] = 0;

        // Loop through prices to calculate max profit at
        // each day
        for (int i = 1; i < n; i++) {
            // choice 1: Buy the stock at i, in which case
            // the profit we get is the maximum profit we
            // could have made till i-1 minus the price at
```

```
            // choice 2: Sell the stock at i, in which case
            // the profit we get is the maximum profit we
            // could have made till i-1 by buying the stock
            // earlier plus the price at i.
            dp[i][1] = Math.max(
                dp[i - 1][1], dp[i - 1][0] + prices.get(i));
        }

        // Return the maximum profit calculated from the
        // last day
        return Math.max(dp[n - 1][0], dp[n - 1][1]);
    }

    public static void main(String[] args)
    {
        // Given prices
        List<Integer> prices
            = Arrays.asList(7, 1, 5, 3, 6, 4);

        // Function Call
        int ans = maxProfit(prices);

        // Print answer
        System.out.println(ans);
    }
}

// This code is contributed by Susobhan Akhuli
```

Python3

```
# Python code for the approach
def maxProfit(prices):
    n = len(prices)

    # 2D DP array to store max profit with 0 and 1 stocks
    dp = [[0 for _ in range(2)] for _ in range(n)]

    dp[0][0] = -prices[0]
    dp[0][1] = 0

    # Loop through prices to calculate max profit at each day
    for i in range(1, n):
        # choice 1: Buy the stock at i, in which case the
        # profit we get is the maximum profit we could have
        # made till i-1 minus the price at i.
        dp[i][0] = max(dp[i - 1][0], -prices[i])
```

```
        # made till i-1 by buying the stock earlier plus
        # the price at i.
        dp[i][1] = max(dp[i - 1][1], dp[i - 1][0] + prices[i])

    # Return the maximum profit calculated from the last day
    return max(dp[-1][0], dp[-1][1])

# Driver's code
if __name__ == "__main__":
    # Given prices
    prices = [7, 1, 5, 3, 6, 4]

    # Function Call
    ans = maxProfit(prices)

    # Print answer
    print(ans)

# This code is contributed by Susobhan Akhuli
```

C#

```
// C# code for the approach
using System;
using System.Collections.Generic;

public class GFG {
    // Function to find the maximum
    // profit with at most 1 transaction
    public static int maxProfit(List<int> prices)
    {
        int n = prices.Count;

        // 2D DP array to store max profit with 0 and 1
        // stocks
        int[, ] dp = new int[n, 2];

        dp[0, 0] = -prices[0];
        dp[0, 1] = 0;

        // Loop through prices to calculate max profit at
        // each day
        for (int i = 1; i < n; i++) {
            // choice 1: Buy the stock at i, in which case
            // the profit we get is the maximum profit we
            // could have made till i-1 minus the price at
            // i.
            dp[i, 0] = Math.Max(dp[i - 1, 0], -prices[i]);
```

```
            // could have made till i-1 by buying the stock
            // earlier plus the price at i.
            dp[i, 1] = Math.Max(dp[i - 1, 1],
                                dp[i - 1, 0] + prices[i]);
        }

        // Return the maximum profit calculated from the
        // last day
        return Math.Max(dp[n - 1, 0], dp[n - 1, 1]);
    }

    // Driver's code
    public static void Main()
    {
        // Given prices
        List<int> prices
            = new List<int>{ 7, 1, 5, 3, 6, 4 };

        // Function Call
        int ans = maxProfit(prices);

        // Print answer
        Console.WriteLine(ans);
    }
}

// This code is contributed by Susobhan Akhuli
```

Javascript

```
function maxProfit(prices) {
    const n = prices.length;

    // 2D DP array to store max profit with 0 and 1 stocks
    const dp = new Array(n).fill(null).map(() => [0, 0]);

    dp[0][0] = -prices[0];
    dp[0][1] = 0;

    // Loop through prices to calculate max profit at each day
    for (let i = 1; i < n; i++) {
        // choice 1: Buy the stock at i, in which case the profit we get is
        // the maximum profit we could have made till i-1 minus the price at i.
        dp[i][0] = Math.max(dp[i - 1][0], -prices[i]);

        // choice 2: Sell the stock at i, in which case the profit we get is
        // the maximum profit we could have made till i-1 by buying the stock
        // earlier plus the price at i
```

```
    // Return the maximum profit calculated from the last day
    return Math.max(dp[n - 1][0], dp[n - 1][1]);
}

// Given prices
const prices = [7, 1, 5, 3, 6, 4];

// Function Call
const ans = maxProfit(prices);

// Print answer
console.log(ans);
```

Output

```
5
```

Time complexity: O(N), where N is the length of the given array.
Auxiliary Space: O(N)

Recommended Problems

Frequently asked DSA Problems

Solve Problems

Last Updated : 02 Jan, 2024

32

Previous

Maximum number of trailing zeros in the product of the subsets of size k

Next

Find numbers starting from 1 with sum at-most K excluding given numbers

Share your thoughts in the comments

Add Your Comment

Similar Reads

Stock Buy and Sell Problems with Variations

C Program For Stock Buy Sell To Maximize Profit

Stock Buy Sell to Maximize Profit

Python Program For Stock Buy Sell To Maximize Profit

Javascript Program For Stock Buy Sell To Maximize Profit

C++ Program For Stock Buy Sell To Maximize Profit

Java Program For Stock Buy Sell To Maximize Profit

Buy minimum items without change and given coins

Find the minimum and maximum amount to buy all N candies

Time difference between expected time and given time

Learn Algorithms with Javascript | DSA using JavaScript Tutorial

DSA Crash Course | Revision Checklist with Interview Guide

Learn Data Structures and Algorithms | DSA Tutorial

Mathematical and Geometric Algorithms - Data Structure and Algorithm Tutorials

Learn Data Structures with Javascript | DSA using JavaScript Tutorial

shubhagr... Follow

Article Tags : Technical Scripter 2020 , C/C++ Puzzles , DSA , Dynamic Programming , Technical Scripter

Practice Tags : Dynamic Programming

Additional Information

GeeksforGeeks
A-143, 9th Floor, Sovereign Corporate Tower, Sector-136, Noida, Uttar Pradesh - 201305

Git

AWS

Docker

Kubernetes

Azure

GCP

DevOps Roadmap

System Design

What is System Design

Monolithic and Distributed SD

High Level Design or HLD

Low Level Design or LLD

Crack System Design Round

System Design Interview Questions

Grokking Modern System Design

NCERT Solutions

Class 12

Class 11

Class 10

Class 9

Class 8

Complete Study Material

Commerce

Accountancy

Business Studies

Indian Economics

Macroeconomics

Microeconimics

Statistics for Economics

UPSC Study Material

Top DS or Algo for CP

Top 50 Tree

Top 50 Graph

Top 50 Array

Top 50 String

Top 50 DP

Top 15 Websites for CP

JavaScript

TypeScript

ReactJS

NextJS

AngularJS

NodeJS

Express.js

Lodash

Web Browser

School Subjects

Mathematics

Physics

Chemistry

Biology

Social Science

English Grammar

Management & Finance

Management

HR Managament

Income Tax

Finance

Economics

SSC/ BANKING

History Notes

Science and Technology Notes

Economy Notes

Ethics Notes

Previous Year Papers

SBI Clerk Syllabus

IBPS PO Syllabus

IBPS Clerk Syllabus

SSC CGL Practice Papers

Colleges

Indian Colleges Admission & Campus Experiences

Top Engineering Colleges

Top BCA Colleges

Top MBA Colleges

Top Architecture College

Choose College For Graduation

Companies

IT Companies

Software Development Companies

Artificial Intelligence(AI) Companies

CyberSecurity Companies

Service Based Companies

Product Based Companies

PSUs for CS Engineers

Preparation Corner

Company Wise Preparation

Preparation for SDE

Experienced Interviews

Internship Interviews

Competitive Programming

Aptitude Preparation

Puzzles

Exams

JEE Mains

JEE Advanced

GATE CS

NEET

UGC NET

More Tutorials

Software Development

Software Testing

Product Management

SAP

SEO

Linux

Excel

Write & Earn

Write an Article

Improve an Article

Pick Topics to Write

Share your Experiences

Internships