Problem 122: Best Time to Buy and Sell Stock II

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

You are given an integer array

prices

where

prices[i]

is the price of a given stock on the

i

th

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 sell and buy the stock multiple times on the

same day

, ensuring you never hold more than one share of the stock.

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 * 10

4

0 <= prices[i] <= 10

4

Code Snippets

C++:

```
class Solution {
public:
int maxProfit(vector<int>& prices) {

}
};
```

Java:

```
class Solution {
public int maxProfit(int[] prices) {

}
}
```

Python3:

```
class Solution:
def maxProfit(self, prices: List[int]) -> int:
```

Python:

```
class Solution(object):
def maxProfit(self, prices):
"""
:type prices: List[int]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} prices
* @return {number}
*/
var maxProfit = function(prices) {

};
```

TypeScript:

```
function maxProfit(prices: number[]): number {

};
```

C#:

```
public class Solution {
public int MaxProfit(int[] prices) {

}
}
```

C:

```
int maxProfit(int* prices, int pricesSize) {

}
```

Go:

```
func maxProfit(prices []int) int {

}
```

Kotlin:

```
class Solution {
fun maxProfit(prices: IntArray): Int {

}
}
```

Swift:

```
class Solution {
func maxProfit(_ prices: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn max_profit(prices: Vec<i32>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} prices
# @return {Integer}
def max_profit(prices)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $prices
* @return Integer
*/
function maxProfit($prices) {

}
}
```

Dart:

```
class Solution {
int maxProfit(List<int> prices) {

}
}
```

Scala:

```
object Solution {
def maxProfit(prices: Array[Int]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec max_profit(prices :: [integer]) :: integer
def max_profit(prices) do

end
end
```

Erlang:

```
-spec max_profit(Prices :: [integer()]) -> integer().
max_profit(Prices) ->
.
```

Racket:

```
(define/contract (max-profit prices)
(-> (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Best Time to Buy and Sell Stock II
* Difficulty: Medium
* Tags: array, dp, greedy
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
int maxProfit(vector<int>& prices) {

}
};
```

Java Solution:

```
/**
* Problem: Best Time to Buy and Sell Stock II
* Difficulty: Medium
* Tags: array, dp, greedy
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public int maxProfit(int[] prices) {
```

```
}
}
```

Python3 Solution:

```
"""
Problem: Best Time to Buy and Sell Stock II
Difficulty: Medium
Tags: array, dp, greedy

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def maxProfit(self, prices: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def maxProfit(self, prices):
"""
:type prices: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Best Time to Buy and Sell Stock II
* Difficulty: Medium
* Tags: array, dp, greedy
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/
```

```
/**
* @param {number[]} prices
* @return {number}
*/
var maxProfit = function(prices) {

};
```

TypeScript Solution:

```
/**
* Problem: Best Time to Buy and Sell Stock II
* Difficulty: Medium
* Tags: array, dp, greedy
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

function maxProfit(prices: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Best Time to Buy and Sell Stock II
* Difficulty: Medium
* Tags: array, dp, greedy
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public int MaxProfit(int[] prices) {

}
```

```
}
```

C Solution:

```
/*
* Problem: Best Time to Buy and Sell Stock II
* Difficulty: Medium
* Tags: array, dp, greedy
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

int maxProfit(int* prices, int pricesSize) {

}
```

Go Solution:

```
// Problem: Best Time to Buy and Sell Stock II
// Difficulty: Medium
// Tags: array, dp, greedy
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func maxProfit(prices []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun maxProfit(prices: IntArray): Int {

}
}
```

Swift Solution:

```
class Solution {
func maxProfit(_ prices: [Int]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Best Time to Buy and Sell Stock II
// Difficulty: Medium
// Tags: array, dp, greedy
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn max_profit(prices: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} prices
# @return {Integer}
def max_profit(prices)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $prices
* @return Integer
*/
function maxProfit($prices) {

}
}
```

Dart Solution:

```
class Solution {
int maxProfit(List<int> prices) {

}
}
```

Scala Solution:

```
object Solution {
def maxProfit(prices: Array[Int]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec max_profit(prices :: [integer]) :: integer
def max_profit(prices) do

end
end
```

Erlang Solution:

```
-spec max_profit(Prices :: [integer()]) -> integer().
max_profit(Prices) ->
.
```

Racket Solution:

```
(define/contract (max-profit prices)
(-> (listof exact-integer?) exact-integer?)
)
```