

Problem 3573: Best Time to Buy and Sell Stock V

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 stock in dollars on the

i

th

day, and an integer

k

.

You are allowed to make at most

k

transactions, where each transaction can be either of the following:

Normal transaction

: Buy on day

i

, then sell on a later day

j

where

$i < j$

. You profit

$\text{prices}[j] - \text{prices}[i]$

.

Short selling transaction

: Sell on day

i

, then buy back on a later day

j

where

$i < j$

. You profit

$\text{prices}[i] - \text{prices}[j]$

.

Note

that you must complete each transaction before starting another. Additionally, you can't buy or sell on the same day you are selling or buying back as part of a previous transaction.

Return the

maximum

total profit you can earn by making

at most

k

transactions.

Example 1:

Input:

prices = [1,7,9,8,2], $k = 2$

Output:

14

Explanation:

We can make \$14 of profit through 2 transactions:

A normal transaction: buy the stock on day 0 for \$1 then sell it on day 2 for \$9.

A short selling transaction: sell the stock on day 3 for \$8 then buy back on day 4 for \$2.

Example 2:

Input:

prices = [12,16,19,19,8,1,19,13,9], k = 3

Output:

36

Explanation:

We can make \$36 of profit through 3 transactions:

A normal transaction: buy the stock on day 0 for \$12 then sell it on day 2 for \$19.

A short selling transaction: sell the stock on day 3 for \$19 then buy back on day 4 for \$8.

A normal transaction: buy the stock on day 5 for \$1 then sell it on day 6 for \$19.

Constraints:

$2 \leq \text{prices.length} \leq 10$

3

$1 \leq \text{prices}[i] \leq 10$

9

$1 \leq k \leq \text{prices.length} / 2$

Code Snippets

C++:

```
class Solution {
public:
    long long maximumProfit(vector<int>& prices, int k) {
```

```
}  
};
```

Java:

```
class Solution {  
    public long maximumProfit(int[] prices, int k) {  
  
    }  
}
```

Python3:

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

Python:

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

JavaScript:

```
/**  
 * @param {number[]} prices  
 * @param {number} k  
 * @return {number}  
 */  
var maximumProfit = function(prices, k) {  
  
};
```

TypeScript:

```
function maximumProfit(prices: number[], k: number): number {  
  
};
```

C#:

```
public class Solution {  
    public long MaximumProfit(int[] prices, int k) {  
  
    }  
}
```

C:

```
long long maximumProfit(int* prices, int pricesSize, int k) {  
  
}
```

Go:

```
func maximumProfit(prices []int, k int) int64 {  
  
}
```

Kotlin:

```
class Solution {  
    fun maximumProfit(prices: IntArray, k: Int): Long {  
  
    }  
}
```

Swift:

```
class Solution {  
    func maximumProfit(_ prices: [Int], _ k: Int) -> Int {  
  
    }  
}
```

Rust:

```
impl Solution {  
    pub fn maximum_profit(prices: Vec<i32>, k: i32) -> i64 {  
  
    }  
}
```

Ruby:

```
# @param {Integer[]} prices
# @param {Integer} k
# @return {Integer}
def maximum_profit(prices, k)

end
```

PHP:

```
class Solution {

    /**
     * @param Integer[] $prices
     * @param Integer $k
     * @return Integer
     */
    function maximumProfit($prices, $k) {

    }

}
```

Dart:

```
class Solution {
  int maximumProfit(List<int> prices, int k) {

  }
}
```

Scala:

```
object Solution {
  def maximumProfit(prices: Array[Int], k: Int): Long = {

  }
}
```

Elixir:

```
defmodule Solution do
  @spec maximum_profit(prices :: [integer], k :: integer) :: integer
```

```

def maximum_profit(prices, k) do

end

end

```

Erlang:

```

-spec maximum_profit(Prices :: [integer()], K :: integer()) -> integer().
maximum_profit(Prices, K) ->
.

```

Racket:

```

(define/contract (maximum-profit prices k)
  (-> (listof exact-integer?) exact-integer? exact-integer?)
  )

```

Solutions

C++ Solution:

```

/*
 * Problem: Best Time to Buy and Sell Stock V
 * Difficulty: Medium
 * Tags: array, dp
 *
 * 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:
    long long maximumProfit(vector<int>& prices, int k) {

    }

};

```

Java Solution:

```

/**
 * Problem: Best Time to Buy and Sell Stock V
 * Difficulty: Medium
 * Tags: array, dp
 *
 * 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 long maximumProfit(int[] prices, int k) {

}
}

```

Python3 Solution:

```

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

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 maximumProfit(self, prices: List[int], k: int) -> int:
# TODO: Implement optimized solution
pass

```

Python Solution:

```

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

```

JavaScript Solution:

```
/**
 * Problem: Best Time to Buy and Sell Stock V
 * Difficulty: Medium
 * Tags: array, dp
 *
 * 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
 * @param {number} k
 * @return {number}
 */
var maximumProfit = function(prices, k) {

};
```

TypeScript Solution:

```
/**
 * Problem: Best Time to Buy and Sell Stock V
 * Difficulty: Medium
 * Tags: array, dp
 *
 * 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 maximumProfit(prices: number[], k: number): number {

};
```

C# Solution:

```
/*
 * Problem: Best Time to Buy and Sell Stock V
 * Difficulty: Medium
```

```

* Tags: array, dp
*
* 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 long MaximumProfit(int[] prices, int k) {

}
}

```

C Solution:

```

/*
* Problem: Best Time to Buy and Sell Stock V
* Difficulty: Medium
* Tags: array, dp
*
* 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
*/

long long maximumProfit(int* prices, int pricesSize, int k) {

}

```

Go Solution:

```

// Problem: Best Time to Buy and Sell Stock V
// Difficulty: Medium
// Tags: array, dp
//
// 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 maximumProfit(prices []int, k int) int64 {

```

```
}
```

Kotlin Solution:

```
class Solution {  
    fun maximumProfit(prices: IntArray, k: Int): Long {  
  
    }  
}
```

Swift Solution:

```
class Solution {  
    func maximumProfit(_ prices: [Int], _ k: Int) -> Int {  
  
    }  
}
```

Rust Solution:

```
// Problem: Best Time to Buy and Sell Stock V  
// Difficulty: Medium  
// Tags: array, dp  
//  
// 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 maximum_profit(prices: Vec<i32>, k: i32) -> i64 {  
  
    }  
}
```

Ruby Solution:

```
# @param {Integer[]} prices  
# @param {Integer} k  
# @return {Integer}  
def maximum_profit(prices, k)
```

```
end
```

PHP Solution:

```
class Solution {  
  
    /**  
     * @param Integer[] $prices  
     * @param Integer $k  
     * @return Integer  
     */  
    function maximumProfit($prices, $k) {  
  
    }  
}
```

Dart Solution:

```
class Solution {  
    int maximumProfit(List<int> prices, int k) {  
  
    }  
}
```

Scala Solution:

```
object Solution {  
    def maximumProfit(prices: Array[Int], k: Int): Long = {  
  
    }  
}
```

Elixir Solution:

```
defmodule Solution do  
    @spec maximum_profit(prices :: [integer], k :: integer) :: integer  
    def maximum_profit(prices, k) do  
  
    end  
end
```

Erlang Solution:

```
-spec maximum_profit(Prices :: [integer()], K :: integer()) -> integer().  
maximum_profit(Prices, K) ->  
.
```

Racket Solution:

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