Problem 188: Best Time to Buy and Sell Stock IV

Problem Information

Difficulty: Hard
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, and an integer

k

.

Find the maximum profit you can achieve. You may complete at most

k

transactions: i.e. you may buy at most

k

times and sell at most

k

times.

Note:

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

Example 1:

Input:

k = 2, prices = [2,4,1]

Output:

2

Explanation:

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

Example 2:

Input:

k = 2, prices = [3,2,6,5,0,3]

Output:

7

Explanation:

Buy on day 2 (price = 2) and sell on day 3 (price = 6), profit = 6-2 = 4. Then buy on day 5 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.

Constraints:

1 <= k <= 100

1 <= prices.length <= 1000

0 <= prices[i] <= 1000

Code Snippets

C++:

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

}
};
```

Java:

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

}
}
```

Python3:

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

Python:

```
class Solution(object):
def maxProfit(self, k, prices):
"""
```

```
:type k: int
:type prices: List[int]
:rtype: int
"""
```

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

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

}
}
```

C:

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

}
```

Go:

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

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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


}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

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

Solutions

C++ Solution:

```
/*
* Problem: Best Time to Buy and Sell Stock IV
* Difficulty: Hard
* 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:
int maxProfit(int k, vector<int>& prices) {

}
};
```

Java Solution:

```
/**
* Problem: Best Time to Buy and Sell Stock IV
* Difficulty: Hard
* 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 int maxProfit(int k, int[] prices) {

}
}
```

Python3 Solution:

```
"""
Problem: Best Time to Buy and Sell Stock IV
```

```
Difficulty: Hard
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 maxProfit(self, k: int, prices: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Best Time to Buy and Sell Stock IV
* Difficulty: Hard
* 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} k
* @param {number[]} prices
* @return {number}
*/
var maxProfit = function(k, prices) {
};
```

TypeScript Solution:

```
/**
* Problem: Best Time to Buy and Sell Stock IV
* Difficulty: Hard
* 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 maxProfit(k: number, prices: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Best Time to Buy and Sell Stock IV
* Difficulty: Hard
* 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 int MaxProfit(int k, int[] prices) {

}
}
```

C Solution:

```
/*
* Problem: Best Time to Buy and Sell Stock IV
* Difficulty: Hard
* 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
*/

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

}
```

Go Solution:

```
// Problem: Best Time to Buy and Sell Stock IV
// Difficulty: Hard
// 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 maxProfit(k int, prices []int) int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Best Time to Buy and Sell Stock IV
// Difficulty: Hard
// 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 max_profit(k: i32, prices: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

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