# Problem 3652: Best Time to Buy and Sell Stock using Strategy

## Problem Information

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

## Problem Description

You are given two integer arrays

prices

and

strategy

, where:

prices[i]

is the price of a given stock on the

i

th

day.

strategy[i]

represents a trading action on the

i

th

day, where:

-1

indicates buying one unit of the stock.

0

indicates holding the stock.

1

indicates selling one unit of the stock.

You are also given an

even

integer

k

, and may perform

at most one

modification to

strategy

. A modification consists of:

Selecting exactly

k

consecutive elements in strategy. Set the first k / 2 elements to 0 (hold). Set the last k / 2 elements to 1 (sell). The profit is defined as the sum of

strategy[i] * prices[i]

across all days.

Return the

maximum

possible profit you can achieve.

Note:

There are no constraints on budget or stock ownership, so all buy and sell operations are feasible regardless of past actions.

Example 1:

Input:

prices = [4,2,8], strategy = [-1,0,1], k = 2

Output:

10

Explanation:

Modification

Strategy

Profit Calculation

Profit

Original

[-1, 0, 1]

(-1 × 4) + (0 × 2) + (1 × 8) = -4 + 0 + 8

4

Modify [0, 1]

[0, 1, 1]

(0 × 4) + (1 × 2) + (1 × 8) = 0 + 2 + 8

10

Modify [1, 2]

[-1, 0, 1]

(-1 × 4) + (0 × 2) + (1 × 8) = -4 + 0 + 8

4

Thus, the maximum possible profit is 10, which is achieved by modifying the subarray

[0, 1]

.

Example 2:

Input:

prices = [5,4,3], strategy = [1,1,0], k = 2

Output:

9

Explanation:

Modification

Strategy

Profit Calculation

Profit

Original

[1, 1, 0]

(1 × 5) + (1 × 4) + (0 × 3) = 5 + 4 + 0

9

Modify [0, 1]

[0, 1, 0]

(0 × 5) + (1 × 4) + (0 × 3) = 0 + 4 + 0

4

Modify [1, 2]

[1, 0, 1]

(1 × 5) + (0 × 4) + (1 × 3) = 5 + 0 + 3

8

Thus, the maximum possible profit is 9, which is achieved without any modification.

Constraints:

2 <= prices.length == strategy.length <= 10

5

1 <= prices[i] <= 10

5

-1 <= strategy[i] <= 1

2 <= k <= prices.length

k

is even

## Code Snippets

**C++:**

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

}
};
```

**Java:**

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

}
}
```

**Python3:**

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

**Python:**

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

### JavaScript:

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

};
```

### TypeScript:

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

};
```

### C#:

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

}
}
```

### C:

```
long long maxProfit(int* prices, int pricesSize, int* strategy, int
strategySize, int k) {

}
```

### Go:

```
func maxProfit(prices []int, strategy []int, k int) int64 {

}
```

### Kotlin:

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

}
}
```

### Swift:

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

}
}
```

### Rust:

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

}
}
```

### Ruby:

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

end
```

### PHP:

```
class Solution {

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

}
}
```

### Dart:

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

}
}
```

### Scala:

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

}
}
```

### Elixir:

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

end
end
```

### Erlang:

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

### Racket:

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

## Solutions

### C++ Solution:

```
/*
* Problem: Best Time to Buy and Sell Stock using Strategy
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

}
};
```

### Java Solution:

```
/**
* Problem: Best Time to Buy and Sell Stock using Strategy
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
```

```
public long maxProfit(int[] prices, int[] strategy, int k) {

}
}
```

## Python3 Solution:

```
"""
Problem: Best Time to Buy and Sell Stock using Strategy
Difficulty: Medium
Tags: array

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

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

## Python Solution:

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

## JavaScript Solution:

```
/**
* Problem: Best Time to Buy and Sell Stock using Strategy
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

};
```

### TypeScript Solution:

```
/**
* Problem: Best Time to Buy and Sell Stock using Strategy
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

};
```

### C# Solution:

```
/*
* Problem: Best Time to Buy and Sell Stock using Strategy
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/
```

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

}
}
```

### C Solution:

```
/*
* Problem: Best Time to Buy and Sell Stock using Strategy
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

long long maxProfit(int* prices, int pricesSize, int* strategy, int
strategySize, int k) {

}
```

### Go Solution:

```
// Problem: Best Time to Buy and Sell Stock using Strategy
// Difficulty: Medium
// Tags: array
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func maxProfit(prices []int, strategy []int, k int) int64 {

}
```

### Kotlin Solution:

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

}
}
```

### Swift Solution:

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

}
}
```

### Rust Solution:

```
// Problem: Best Time to Buy and Sell Stock using Strategy
// Difficulty: Medium
// Tags: array
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn max_profit(prices: Vec<i32>, strategy: Vec<i32>, k: i32) -> i64 {

}
}
```

### Ruby Solution:

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

end
```

### PHP Solution:

```
class Solution {

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

}
}
```

### Dart Solution:

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

}
}
```

### Scala Solution:

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

}
}
```

### Elixir Solution:

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

end
end
```

### Erlang Solution:

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

### Racket Solution:

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