Problem 3196: Maximize Total Cost of Alternating Subarrays

Problem Information

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

Problem Description

You are given an integer array

nums

with length

n

.

The

cost

of a

subarray

nums[l..r]

, where

0 <= l <= r < n

, is defined as:

cost(l, r) = nums[l] - nums[l + 1] + ... + nums[r] * (–1)

r – l

Your task is to

split

nums

into subarrays such that the

total

cost

of the subarrays is

maximized

, ensuring each element belongs to

exactly one

subarray.

Formally, if

nums

is split into

k

subarrays, where

k > 1

, at indices

i

1

, i

2

, ..., i

k – 1

, where

0 <= i

1

< i

2

< ... < i

k - 1

< n - 1

, then the total cost will be:

cost(0, i

1

) + cost(i

1

+ 1, i

2

) + ... + cost(i

k – 1

+ 1, n – 1)

Return an integer denoting the

maximum total cost

of the subarrays after splitting the array optimally.

Note:

If

nums

is not split into subarrays, i.e.

k = 1

, the total cost is simply

cost(0, n - 1)

.

Example 1:

Input:

nums = [1,-2,3,4]

Output:

10

Explanation:

One way to maximize the total cost is by splitting

[1, -2, 3, 4]

into subarrays

[1, -2, 3]

and

[4]

. The total cost will be

(1 + 2 + 3) + 4 = 10

.

Example 2:

Input:

nums = [1,-1,1,-1]

Output:

4

Explanation:

One way to maximize the total cost is by splitting

[1, -1, 1, -1]

into subarrays

[1, -1]

and

[1, -1]

. The total cost will be

(1 + 1) + (1 + 1) = 4

.

Example 3:

Input:

nums = [0]

Output:

0

Explanation:

We cannot split the array further, so the answer is 0.

Example 4:

Input:

nums = [1,-1]

Output:

2

Explanation:

Selecting the whole array gives a total cost of

1 + 1 = 2

, which is the maximum.

Constraints:

1 <= nums.length <= 10

5

-10

9

<= nums[i] <= 10

9

Code Snippets

C++:

```
class Solution {
public:
long long maximumTotalCost(vector<int>& nums) {

}
};
```

Java:

```
class Solution {
public long maximumTotalCost(int[] nums) {

}
```

```
}
```

Python3:

```
class Solution:
def maximumTotalCost(self, nums: List[int]) -> int:
```

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function maximumTotalCost(nums: number[]): number {

};
```

C#:

```
public class Solution {
public long MaximumTotalCost(int[] nums) {

}
}
```

C:

```
long long maximumTotalCost(int* nums, int numsSize) {

}
```

Go:

```
func maximumTotalCost(nums []int) int64 {

}
```

Kotlin:

```
class Solution {
fun maximumTotalCost(nums: IntArray): Long {

}
}
```

Swift:

```
class Solution {
func maximumTotalCost(_ nums: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn maximum_total_cost(nums: Vec<i32>) -> i64 {

}
}
```

Ruby:

```
# @param {Integer[]} nums
# @return {Integer}
def maximum_total_cost(nums)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums
* @return Integer
*/
function maximumTotalCost($nums) {

}
}
```

Dart:

```
class Solution {
int maximumTotalCost(List<int> nums) {

}
}
```

Scala:

```
object Solution {
def maximumTotalCost(nums: Array[Int]): Long = {

}
}
```

Elixir:

```
defmodule Solution do
@spec maximum_total_cost(nums :: [integer]) :: integer
def maximum_total_cost(nums) do

end
end
```

Erlang:

```
-spec maximum_total_cost(Nums :: [integer()]) -> integer().
maximum_total_cost(Nums) ->
.
```

Racket:

```
(define/contract (maximum-total-cost nums)
(-> (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Maximize Total Cost of Alternating Subarrays
* 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 maximumTotalCost(vector<int>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Maximize Total Cost of Alternating Subarrays
* 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 maximumTotalCost(int[] nums) {

}
```

```
}
```

Python3 Solution:

```
"""
Problem: Maximize Total Cost of Alternating Subarrays
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 maximumTotalCost(self, nums: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Maximize Total Cost of Alternating Subarrays
* 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[]} nums
* @return {number}
*/
var maximumTotalCost = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Maximize Total Cost of Alternating Subarrays
* 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 maximumTotalCost(nums: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Maximize Total Cost of Alternating Subarrays
* 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 MaximumTotalCost(int[] nums) {

}
}
```

C Solution:

```
/*
* Problem: Maximize Total Cost of Alternating Subarrays
* 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 maximumTotalCost(int* nums, int numsSize) {

}
```

Go Solution:

```
// Problem: Maximize Total Cost of Alternating Subarrays
// 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 maximumTotalCost(nums []int) int64 {

}
```

Kotlin Solution:

```
class Solution {
fun maximumTotalCost(nums: IntArray): Long {

}
}
```

Swift Solution:

```
class Solution {
func maximumTotalCost(_ nums: [Int]) -> Int {
```

```
}
}
```

Rust Solution:

```
// Problem: Maximize Total Cost of Alternating Subarrays
// 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_total_cost(nums: Vec<i32>) -> i64 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} nums
# @return {Integer}
def maximum_total_cost(nums)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $nums
* @return Integer
*/
function maximumTotalCost($nums) {

}
}
```

Dart Solution:

```
class Solution {
int maximumTotalCost(List<int> nums) {

}
}
```

Scala Solution:

```
object Solution {
def maximumTotalCost(nums: Array[Int]): Long = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec maximum_total_cost(nums :: [integer]) :: integer
def maximum_total_cost(nums) do

end
end
```

Erlang Solution:

```
-spec maximum_total_cost(Nums :: [integer()]) -> integer().
maximum_total_cost(Nums) ->
.
```

Racket Solution:

```
(define/contract (maximum-total-cost nums)
(-> (listof exact-integer?) exact-integer?)
)
```