Problem 2036: Maximum Alternating Subarray Sum

Problem Information

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

Problem Description

A

subarray

of a

0-indexed

integer array is a contiguous

non-empty

sequence of elements within an array.

The

alternating subarray sum

of a subarray that ranges from index

i

to

j

(

inclusive

,

0 <= i <= j < nums.length

) is

nums[i] - nums[i+1] + nums[i+2] - ... +/- nums[j]

.

Given a

0-indexed

integer array

nums

, return

the

maximum alternating subarray sum

of any subarray of

nums

.

Example 1:

Input:

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

Output:

5

Explanation:

The subarray [3,-1,1] has the largest alternating subarray sum. The alternating subarray sum is 3 - (-1) + 1 = 5.

Example 2:

Input:

nums = [2,2,2,2,2]

Output:

2

Explanation:

The subarrays [2], [2,2,2], and [2,2,2,2,2] have the largest alternating subarray sum. The alternating subarray sum of [2] is 2. The alternating subarray sum of [2,2,2] is 2 - 2 + 2 = 2. The alternating subarray sum of [2,2,2,2,2] is 2 - 2 + 2 - 2 + 2 = 2.

Example 3:

Input:

nums = [1]

Output:

1

Explanation:

There is only one non-empty subarray, which is [1]. The alternating subarray sum is 1.

Constraints:

1 <= nums.length <= 10

5

-10

5

<= nums[i] <= 10

5

Code Snippets

C++:

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

}
};
```

Java:

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

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

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

}
}
```

C:

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

}
```

Go:

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

```
}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums
* @return Integer
*/
```

```
function maximumAlternatingSubarraySum($nums) {

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (maximum-alternating-subarray-sum nums)
(-> (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Maximum Alternating Subarray Sum
* 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 maximumAlternatingSubarraySum(vector<int>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Maximum Alternating Subarray Sum
* 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 maximumAlternatingSubarraySum(int[] nums) {

}
}
```

Python3 Solution:

```
"""
Problem: Maximum Alternating Subarray Sum
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 maximumAlternatingSubarraySum(self, nums: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Maximum Alternating Subarray Sum
* 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 maximumAlternatingSubarraySum = function(nums) {
```

```
};
```

TypeScript Solution:

```
/**
* Problem: Maximum Alternating Subarray Sum
* 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 maximumAlternatingSubarraySum(nums: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Maximum Alternating Subarray Sum
* 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 MaximumAlternatingSubarraySum(int[] nums) {

}
}
```

C Solution:

```
/*
* Problem: Maximum Alternating Subarray Sum
* 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 maximumAlternatingSubarraySum(int* nums, int numsSize) {

}
```

Go Solution:

```
// Problem: Maximum Alternating Subarray Sum
// 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 maximumAlternatingSubarraySum(nums []int) int64 {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Maximum Alternating Subarray Sum
// 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_alternating_subarray_sum(nums: Vec<i32>) -> i64 {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

```
}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (maximum-alternating-subarray-sum nums)
(-> (listof exact-integer?) exact-integer?)
)
```