Problem 1911: Maximum Alternating Subsequence Sum

Problem Information

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

Problem Description

The

alternating sum

of a

0-indexed

array is defined as the

sum

of the elements at

even

indices

minus

the

sum

of the elements at

odd

indices.

For example, the alternating sum of

[4,2,5,3]

is

(4 + 5) - (2 + 3) = 4

.

Given an array

nums

, return

the

maximum alternating sum

of any subsequence of

nums

(after

reindexing

the elements of the subsequence)

.

A

subsequence

of an array is a new array generated from the original array by deleting some elements (possibly none) without changing the remaining elements' relative order. For example,

[2,7,4]

is a subsequence of

[4,

2

,3,

7

,2,1,

4

]

(the underlined elements), while

[2,4,2]

is not.

Example 1:

Input:

nums = [

4

,

2

,

5

,3]

Output:

7

Explanation:

It is optimal to choose the subsequence [4,2,5] with alternating sum (4 + 5) - 2 = 7.

Example 2:

Input:

nums = [5,6,7,

8

]

Output:

8

Explanation:

It is optimal to choose the subsequence [8] with alternating sum 8.

Example 3:

Input:

nums = [

6

,2,

1

,2,4,

5

]

Output:

10

Explanation:

It is optimal to choose the subsequence [6,1,5] with alternating sum (6 + 5) - 1 = 10.

Constraints:

1 <= nums.length <= 10

5

1 <= nums[i] <= 10

5

Code Snippets

C++:

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

```
}
};
```

Java:

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

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

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

}
}
```

C:

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

}
```

Go:

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

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Scala:

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

}
}
```

Racket:

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

)
```

Solutions

C++ Solution:

```
/*
* Problem: Maximum Alternating Subsequence 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 maxAlternatingSum(vector<int>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Maximum Alternating Subsequence 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 maxAlternatingSum(int[] nums) {

}
}
```

Python3 Solution:

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Maximum Alternating Subsequence 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 maxAlternatingSum = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Maximum Alternating Subsequence 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 maxAlternatingSum(nums: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Maximum Alternating Subsequence 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 MaxAlternatingSum(int[] nums) {

}
}
```

C Solution:

```
/*
* Problem: Maximum Alternating Subsequence 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 maxAlternatingSum(int* nums, int numsSize){

}
```

Go Solution:

```
// Problem: Maximum Alternating Subsequence 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 maxAlternatingSum(nums []int) int64 {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Maximum Alternating Subsequence 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 max_alternating_sum(nums: Vec<i32>) -> i64 {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Scala Solution:

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

}
}
```

Racket Solution:

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

)
```