Problem 1856: Maximum Subarray Min-Product

Problem Information

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

Problem Description

The

min-product

of an array is equal to the

minimum value

in the array

multiplied by

the array's

sum

.

For example, the array

[3,2,5]

(minimum value is

2

) has a min-product of

2 * (3+2+5) = 2 * 10 = 20

.

Given an array of integers

nums

, return

the

maximum min-product

of any

non-empty subarray

of

nums

. Since the answer may be large, return it

modulo

10

9

+ 7

.

Note that the min-product should be maximized

before

performing the modulo operation. Testcases are generated such that the maximum min-product

without

modulo will fit in a

64-bit signed integer

.

A

subarray

is a

contiguous

part of an array.

Example 1:

Input:

nums = [1,

2,3,2

]

Output:

14

Explanation:

The maximum min-product is achieved with the subarray [2,3,2] (minimum value is 2). 2 * (2+3+2) = 2 * 7 = 14.

Example 2:

Input:

nums = [2,

3,3

,1,2]

Output:

18

Explanation:

The maximum min-product is achieved with the subarray [3,3] (minimum value is 3). 3 * (3+3) = 3 * 6 = 18.

Example 3:

Input:

nums = [3,1,

5,6,4

,2]

Output:

60

Explanation:

The maximum min-product is achieved with the subarray [5,6,4] (minimum value is 4). 4 * (5+6+4) = 4 * 15 = 60.

Constraints:

1 <= nums.length <= 10

5

1 <= nums[i] <= 10

7

Code Snippets

C++:

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

}
};
```

Java:

```
class Solution {
public int maxSumMinProduct(int[] nums) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

```
public class Solution {
public int MaxSumMinProduct(int[] nums) {

}
}
```

C:

```
int maxSumMinProduct(int* nums, int numsSize) {

}
```

Go:

```
func maxSumMinProduct(nums []int) int {

}
```

Kotlin:

```
class Solution {
fun maxSumMinProduct(nums: IntArray): Int {

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
```

```
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

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

Solutions

C++ Solution:

```
/*
* Problem: Maximum Subarray Min-Product
* Difficulty: Medium
* Tags: array, stack
*
```

```
* 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:
int maxSumMinProduct(vector<int>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Maximum Subarray Min-Product
* Difficulty: Medium
* Tags: array, stack
*
* 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 int maxSumMinProduct(int[] nums) {

}
}
```

Python3 Solution:

```
"""
Problem: Maximum Subarray Min-Product
Difficulty: Medium
Tags: array, stack

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 maxSumMinProduct(self, nums: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Maximum Subarray Min-Product
* Difficulty: Medium
* Tags: array, stack
*
* 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[]} nums
* @return {number}
*/
var maxSumMinProduct = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Maximum Subarray Min-Product
* Difficulty: Medium
* Tags: array, stack
```

```
*
* 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 maxSumMinProduct(nums: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Maximum Subarray Min-Product
* Difficulty: Medium
* Tags: array, stack
*
* 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 int MaxSumMinProduct(int[] nums) {

}
}
```

C Solution:

```
/*
* Problem: Maximum Subarray Min-Product
* Difficulty: Medium
* Tags: array, stack
*
* 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
*/

int maxSumMinProduct(int* nums, int numsSize) {
```

```
}
```

Go Solution:

```
// Problem: Maximum Subarray Min-Product
// Difficulty: Medium
// Tags: array, stack
//
// 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 maxSumMinProduct(nums []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun maxSumMinProduct(nums: IntArray): Int {

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Maximum Subarray Min-Product
// Difficulty: Medium
// Tags: array, stack
//
// 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_sum_min_product(nums: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

```
}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

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