Problem 3430: Maximum and Minimum Sums of at Most Size K Subarrays

Problem Information

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

Problem Description

You are given an integer array

nums

and a

positive

integer

k

. Return the sum of the

maximum

and

minimum

elements of all

subarrays

with

at most

k

elements.

Example 1:

Input:

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

Output:

20

Explanation:

The subarrays of

nums

with at most 2 elements are:

Subarray

Minimum

Maximum

Sum

[1]

1

1

2

[2]

2

2

4

[3]

3

3

6

[1, 2]

1

2

3

[2, 3]

2

3

5

Final Total

20

The output would be 20.

Example 2:

Input:

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

Output:

-6

Explanation:

The subarrays of

nums

with at most 2 elements are:

Subarray

Minimum

Maximum

Sum

[1]

1

1

2

[-3]

-3

-3

-6

[1]

1

1

2

[1, -3]

-3

1

-2

[-3, 1]

-3

1

-2

Final Total

-6

The output would be -6.

Constraints:

1 <= nums.length <= 80000

1 <= k <= nums.length

-10

6

<= nums[i] <= 10

6

Code Snippets

C++:

```
class Solution {
public:
long long minMaxSubarraySum(vector<int>& nums, int k) {

}
};
```

Java:

```
class Solution {
public long minMaxSubarraySum(int[] nums, int k) {

}
}
```

Python3:

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

Python:

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

```
"""
```

JavaScript:

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

};
```

TypeScript:

```
function minMaxSubarraySum(nums: number[], k: number): number {

};
```

C#:

```
public class Solution {
public long MinMaxSubarraySum(int[] nums, int k) {

}
}
```

C:

```
long long minMaxSubarraySum(int* nums, int numsSize, int k) {

}
```

Go:

```
func minMaxSubarraySum(nums []int, k int) int64 {

}
```

Kotlin:

```
class Solution {
fun minMaxSubarraySum(nums: IntArray, k: Int): Long {

}
}
```

Swift:

```
class Solution {
func minMaxSubarraySum(_ nums: [Int], _ k: Int) -> Int {

}
}
```

Rust:

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

}
}
```

Ruby:

```
# @param {Integer[]} nums
# @param {Integer} k
# @return {Integer}
def min_max_subarray_sum(nums, k)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums
* @param Integer $k
* @return Integer
*/
function minMaxSubarraySum($nums, $k) {

}
}
```

Dart:

```
class Solution {
int minMaxSubarraySum(List<int> nums, int k) {

}
}
```

Scala:

```
object Solution {
def minMaxSubarraySum(nums: Array[Int], k: Int): Long = {

}
}
```

Elixir:

```
defmodule Solution do
@spec min_max_subarray_sum(nums :: [integer], k :: integer) :: integer
def min_max_subarray_sum(nums, k) do

end
end
```

Erlang:

```
-spec min_max_subarray_sum(Nums :: [integer()], K :: integer()) -> integer().
min_max_subarray_sum(Nums, K) ->
.
```

Racket:

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

Solutions

C++ Solution:

```
/*
* Problem: Maximum and Minimum Sums of at Most Size K Subarrays
* Difficulty: Hard
* Tags: array, math, 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:
long long minMaxSubarraySum(vector<int>& nums, int k) {

}
};
```

Java Solution:

```
/**
* Problem: Maximum and Minimum Sums of at Most Size K Subarrays
* Difficulty: Hard
* Tags: array, math, 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 long minMaxSubarraySum(int[] nums, int k) {

}
}
```

Python3 Solution:

```
"""
Problem: Maximum and Minimum Sums of at Most Size K Subarrays
Difficulty: Hard
Tags: array, math, 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 minMaxSubarraySum(self, nums: List[int], k: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Maximum and Minimum Sums of at Most Size K Subarrays
* Difficulty: Hard
* Tags: array, math, 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
* @param {number} k
* @return {number}
*/
var minMaxSubarraySum = function(nums, k) {

};
```

TypeScript Solution:

```
/**
* Problem: Maximum and Minimum Sums of at Most Size K Subarrays
* Difficulty: Hard
* Tags: array, math, 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 minMaxSubarraySum(nums: number[], k: number): number {

};
```

C# Solution:

```
/*
* Problem: Maximum and Minimum Sums of at Most Size K Subarrays
* Difficulty: Hard
* Tags: array, math, 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 long MinMaxSubarraySum(int[] nums, int k) {

}
}
```

C Solution:

```
/*
* Problem: Maximum and Minimum Sums of at Most Size K Subarrays
* Difficulty: Hard
* Tags: array, math, 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
*/

long long minMaxSubarraySum(int* nums, int numsSize, int k) {

}
```

Go Solution:

```
// Problem: Maximum and Minimum Sums of at Most Size K Subarrays
// Difficulty: Hard
// Tags: array, math, 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 minMaxSubarraySum(nums []int, k int) int64 {

}
```

Kotlin Solution:

```
class Solution {
fun minMaxSubarraySum(nums: IntArray, k: Int): Long {

}
}
```

Swift Solution:

```
class Solution {
func minMaxSubarraySum(_ nums: [Int], _ k: Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: Maximum and Minimum Sums of at Most Size K Subarrays
// Difficulty: Hard
// Tags: array, math, 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 min_max_subarray_sum(nums: Vec<i32>, k: i32) -> i64 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} nums
# @param {Integer} k
# @return {Integer}
def min_max_subarray_sum(nums, k)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $nums
* @param Integer $k
* @return Integer
*/
function minMaxSubarraySum($nums, $k) {

}
}
```

Dart Solution:

```
class Solution {
int minMaxSubarraySum(List<int> nums, int k) {

}
}
```

Scala Solution:

```
object Solution {
def minMaxSubarraySum(nums: Array[Int], k: Int): Long = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec min_max_subarray_sum(nums :: [integer], k :: integer) :: integer
def min_max_subarray_sum(nums, k) do

end
end
```

Erlang Solution:

```
-spec min_max_subarray_sum(Nums :: [integer()], K :: integer()) -> integer().
min_max_subarray_sum(Nums, K) ->
.
```

Racket Solution:

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