Problem 2547: Minimum Cost to Split an Array

Problem Information

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

Problem Description

You are given an integer array

nums

and an integer

k

.

Split the array into some number of non-empty subarrays. The

cost

of a split is the sum of the

importance value

of each subarray in the split.

Let

trimmed(subarray)

be the version of the subarray where all numbers which appear only once are removed.

For example,

trimmed([3,1,2,4,3,4]) = [3,4,3,4].

The

importance value

of a subarray is

k + trimmed(subarray).length

.

For example, if a subarray is

[1,2,3,3,3,4,4]

, then

trimmed(

[1,2,3,3,3,4,4]) = [3,3,3,4,4].

The importance value of this subarray will be

k + 5

.

Return

the minimum possible cost of a split of

nums

.

A

subarray

is a contiguous

non-empty

sequence of elements within an array.

Example 1:

Input:

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

Output:

8

Explanation:

We split nums to have two subarrays: [1,2], [1,2,1,3,3]. The importance value of [1,2] is 2 + (0) = 2. The importance value of [1,2,1,3,3] is 2 + (2 + 2) = 6. The cost of the split is 2 + 6 = 8. It can be shown that this is the minimum possible cost among all the possible splits.

Example 2:

Input:

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

Output:

6

Explanation:

We split nums to have two subarrays: [1,2], [1,2,1]. The importance value of [1,2] is 2 + (0) = 2. The importance value of [1,2,1] is 2 + (2) = 4. The cost of the split is 2 + 4 = 6. It can be shown that this is the minimum possible cost among all the possible splits.

Example 3:

Input:

nums = [1,2,1,2,1], k = 5

Output:

10

Explanation:

We split nums to have one subarray: [1,2,1,2,1]. The importance value of [1,2,1,2,1] is 5 + (3 + 2) = 10. The cost of the split is 10. It can be shown that this is the minimum possible cost among all the possible splits.

Constraints:

1 <= nums.length <= 1000

0 <= nums[i] < nums.length

1 <= k <= 10

9

Code Snippets

C++:

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

}
};
```

Java:

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

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

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

```
}
}
```

C:

```
int minCost(int* nums, int numsSize, int k) {

}
```

Go:

```
func minCost(nums []int, k int) int {

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

```
end
end
```

Erlang:

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

Racket:

```
(define/contract (min-cost nums k)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Minimum Cost to Split an Array
* Difficulty: Hard
* Tags: array, dp, hash
*
* 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:
int minCost(vector<int>& nums, int k) {

}
};
```

Java Solution:

```
/**
* Problem: Minimum Cost to Split an Array
```

```
* Difficulty: Hard
* Tags: array, dp, hash
*
* 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 int minCost(int[] nums, int k) {

}
}
```

Python3 Solution:

```
"""
Problem: Minimum Cost to Split an Array
Difficulty: Hard
Tags: array, dp, hash

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Minimum Cost to Split an Array
* Difficulty: Hard
* Tags: array, dp, hash
*
* 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
* @param {number} k
* @return {number}
*/
var minCost = function(nums, k) {

};
```

TypeScript Solution:

```
/**
* Problem: Minimum Cost to Split an Array
* Difficulty: Hard
* Tags: array, dp, hash
*
* 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 minCost(nums: number[], k: number): number {

};
```

C# Solution:

```
/*
* Problem: Minimum Cost to Split an Array
* Difficulty: Hard
* Tags: array, dp, hash
```

```
*
* 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 int MinCost(int[] nums, int k) {

}
}
```

C Solution:

```
/*
* Problem: Minimum Cost to Split an Array
* Difficulty: Hard
* Tags: array, dp, hash
*
* 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
*/

int minCost(int* nums, int numsSize, int k) {

}
```

Go Solution:

```
// Problem: Minimum Cost to Split an Array
// Difficulty: Hard
// Tags: array, dp, hash
//
// 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 minCost(nums []int, k int) int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Minimum Cost to Split an Array
// Difficulty: Hard
// Tags: array, dp, hash
//
// 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 min_cost(nums: Vec<i32>, k: i32) -> i32 {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (min-cost nums k)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```