

Problem 3500: Minimum Cost to Divide Array Into Subarrays

Problem Information

Difficulty: **Hard**

Acceptance Rate: 0.00%

Paid Only: No

Problem Description

You are given two integer arrays,

`nums`

and

`cost`

, of the same size, and an integer

`k`

.

You can divide

`nums`

into

subarrays

. The cost of the

`i`

th

subarray consisting of elements

`nums[l..r]`

is:

$(\text{nums}[0] + \text{nums}[1] + \dots + \text{nums}[r] + k * i) * (\text{cost}[l] + \text{cost}[l + 1] + \dots + \text{cost}[r])$

.

Note

that

i

represents the order of the subarray: 1 for the first subarray, 2 for the second, and so on.

Return the

minimum

total cost possible from any valid division.

Example 1:

Input:

`nums = [3,1,4], cost = [4,6,6], k = 1`

Output:

110

Explanation:

The minimum total cost possible can be achieved by dividing

nums

into subarrays

[3, 1]

and

[4]

.

The cost of the first subarray

[3,1]

is

$$(3 + 1 + 1 * 1) * (4 + 6) = 50$$

.

The cost of the second subarray

[4]

is

$$(3 + 1 + 4 + 1 * 2) * 6 = 60$$

.

Example 2:

Input:

nums = [4,8,5,1,14,2,2,12,1], cost = [7,2,8,4,2,2,1,1,2], k = 7

Output:

985

Explanation:

The minimum total cost possible can be achieved by dividing

nums

into subarrays

[4, 8, 5, 1]

,

[14, 2, 2]

, and

[12, 1]

.

The cost of the first subarray

[4, 8, 5, 1]

is

$$(4 + 8 + 5 + 1 + 7 * 1) * (7 + 2 + 8 + 4) = 525$$

.

The cost of the second subarray

[14, 2, 2]

is

$$(4 + 8 + 5 + 1 + 14 + 2 + 2 + 7 * 2) * (2 + 2 + 1) = 250$$

.

The cost of the third subarray

[12, 1]

is

$$(4 + 8 + 5 + 1 + 14 + 2 + 2 + 12 + 1 + 7 * 3) * (1 + 2) = 210$$

.

Constraints:

$$1 \leq \text{nums.length} \leq 1000$$

$$\text{cost.length} == \text{nums.length}$$

$$1 \leq \text{nums}[i], \text{cost}[i] \leq 1000$$

$$1 \leq k \leq 1000$$

Code Snippets

C++:

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

    }
};
```

Java:

```

class Solution {
public long minimumCost(int[] nums, int[] cost, int k) {

}

}

```

Python3:

```

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

```

Python:

```

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

```

JavaScript:

```

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

};

```

TypeScript:

```

function minimumCost(nums: number[], cost: number[], k: number): number {

};

```

C#:

```

public class Solution {
    public long MinimumCost(int[] nums, int[] cost, int k) {

    }
}

```

C:

```

long long minimumCost(int* nums, int numsSize, int* cost, int costSize, int
k) {

}

```

Go:

```

func minimumCost(nums []int, cost []int, k int) int64 {

}

```

Kotlin:

```

class Solution {
    fun minimumCost(nums: IntArray, cost: IntArray, k: Int): Long {

    }
}

```

Swift:

```

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

    }
}

```

Rust:

```

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

    }
}

```

Ruby:

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

end
```

PHP:

```
class Solution {

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

    }

}
```

Dart:

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

  }
}
```

Scala:

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

  }
}
```

Elixir:

```

defmodule Solution do
  @spec minimum_cost(nums :: [integer], cost :: [integer], k :: integer) ::
    integer
  def minimum_cost(nums, cost, k) do

  end
end

```

Erlang:

```

-spec minimum_cost(Nums :: [integer()], Cost :: [integer()], K :: integer())
-> integer().
minimum_cost(Nums, Cost, K) ->
.

```

Racket:

```

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

```

Solutions

C++ Solution:

```

/*
 * Problem: Minimum Cost to Divide Array Into Subarrays
 * Difficulty: Hard
 * 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 minimumCost(vector<int>& nums, vector<int>& cost, int k) {

    }
}

```

```
};
```

Java Solution:

```
/**
 * Problem: Minimum Cost to Divide Array Into Subarrays
 * Difficulty: Hard
 * 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 minimumCost(int[] nums, int[] cost, int k) {

    }
}
```

Python3 Solution:

```
"""
Problem: Minimum Cost to Divide Array Into Subarrays
Difficulty: Hard
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 minimumCost(self, nums: List[int], cost: List[int], k: int) -> int:
        # TODO: Implement optimized solution
        pass
```

Python Solution:

```
class Solution(object):
    def minimumCost(self, nums, cost, k):
```

```

"""
:type nums: List[int]
:type cost: List[int]
:type k: int
:rtype: int
"""

```

JavaScript Solution:

```

/**
 * Problem: Minimum Cost to Divide Array Into Subarrays
 * Difficulty: Hard
 * 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
 * @param {number[]} cost
 * @param {number} k
 * @return {number}
 */
var minimumCost = function(nums, cost, k) {

};

```

TypeScript Solution:

```

/**
 * Problem: Minimum Cost to Divide Array Into Subarrays
 * Difficulty: Hard
 * 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 minimumCost(nums: number[], cost: number[], k: number): number {

};
```

C# Solution:

```
/*
 * Problem: Minimum Cost to Divide Array Into Subarrays
 * Difficulty: Hard
 * 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 MinimumCost(int[] nums, int[] cost, int k) {

    }
}
```

C Solution:

```
/*
 * Problem: Minimum Cost to Divide Array Into Subarrays
 * Difficulty: Hard
 * 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 minimumCost(int* nums, int numsSize, int* cost, int costSize, int
k) {

}
```

Go Solution:

```
// Problem: Minimum Cost to Divide Array Into Subarrays
// Difficulty: Hard
// 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 minimumCost(nums []int, cost []int, k int) int64 {

}
```

Kotlin Solution:

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

    }
}
```

Swift Solution:

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

    }
}
```

Rust Solution:

```
// Problem: Minimum Cost to Divide Array Into Subarrays
// Difficulty: Hard
// 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 minimum_cost(nums: Vec<i32>, cost: Vec<i32>, k: i32) -> i64 {

    }
}
```

```
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

    }

}
```

Dart Solution:

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

  }
}
```

Scala Solution:

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

  }
}
```

```
}
```

Elixir Solution:

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

  end
end
```

Erlang Solution:

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

Racket Solution:

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