Problem 1043: Partition Array for Maximum Sum

Problem Information

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

Problem Description

Given an integer array

arr

, partition the array into (contiguous) subarrays of length

at most

k

. After partitioning, each subarray has their values changed to become the maximum value of that subarray.

Return

the largest sum of the given array after partitioning. Test cases are generated so that the answer fits in a

32-bit

integer.

Example 1:

Input:

arr = [1,15,7,9,2,5,10], k = 3

Output:

84

Explanation:

arr becomes [15,15,15,9,10,10,10]

Example 2:

Input:

arr = [1,4,1,5,7,3,6,1,9,9,3], k = 4

Output:

83

Example 3:

Input:

arr = [1], k = 1

Output:

1

Constraints:

1 <= arr.length <= 500

0 <= arr[i] <= 10

1 <= k <= arr.length

Code Snippets

C++:

```
class Solution {
public:
int maxSumAfterPartitioning(vector<int>& arr, int k) {

}
};
```

Java:

```
class Solution {
public int maxSumAfterPartitioning(int[] arr, int k) {

}
}
```

Python3:

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

Python:

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

JavaScript:

```
/**
* @param {number[]} arr
* @param {number} k
```

```
 * @return {number}
 */
var maxSumAfterPartitioning = function(arr, k) {

};
```

TypeScript:

```
function maxSumAfterPartitioning(arr: number[], k: number): number {

};
```

C#:

```
public class Solution {
public int MaxSumAfterPartitioning(int[] arr, int k) {

}
}
```

C:

```
int maxSumAfterPartitioning(int* arr, int arrSize, int k) {

}
```

Go:

```
func maxSumAfterPartitioning(arr []int, k int) int {

}
```

Kotlin:

```
class Solution {
fun maxSumAfterPartitioning(arr: IntArray, k: Int): Int {

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

```
# @param {Integer[]} arr
# @param {Integer} k
# @return {Integer}
def max_sum_after_partitioning(arr, k)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $arr
* @param Integer $k
* @return Integer
*/
function maxSumAfterPartitioning($arr, $k) {

}
}
```

Dart:

```
class Solution {
int maxSumAfterPartitioning(List<int> arr, int k) {

}
```

```
}
```

Scala:

```
object Solution {
def maxSumAfterPartitioning(arr: Array[Int], k: Int): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec max_sum_after_partitioning(arr :: [integer], k :: integer) :: integer
def max_sum_after_partitioning(arr, k) do

end
end
```

Erlang:

```
-spec max_sum_after_partitioning(Arr :: [integer()], K :: integer()) ->
integer().
max_sum_after_partitioning(Arr, K) ->
.
```

Racket:

```
(define/contract (max-sum-after-partitioning arr k)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Partition Array for Maximum 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:
int maxSumAfterPartitioning(vector<int>& arr, int k) {

}
};
```

Java Solution:

```
/**
* Problem: Partition Array for Maximum 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 int maxSumAfterPartitioning(int[] arr, int k) {

}
}
```

Python3 Solution:

```
"""
Problem: Partition Array for Maximum 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 maxSumAfterPartitioning(self, arr: List[int], k: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Partition Array for Maximum 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[]} arr
* @param {number} k
* @return {number}
*/
var maxSumAfterPartitioning = function(arr, k) {

};
```

TypeScript Solution:

```
/**
* Problem: Partition Array for Maximum 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 maxSumAfterPartitioning(arr: number[], k: number): number {

};
```

C# Solution:

```
/*
* Problem: Partition Array for Maximum 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 int MaxSumAfterPartitioning(int[] arr, int k) {

}
}
```

C Solution:

```
/*
* Problem: Partition Array for Maximum 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
```

```
*/

int maxSumAfterPartitioning(int* arr, int arrSize, int k) {

}
```

Go Solution:

```
// Problem: Partition Array for Maximum 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 maxSumAfterPartitioning(arr []int, k int) int {

}
```

Kotlin Solution:

```
class Solution {
fun maxSumAfterPartitioning(arr: IntArray, k: Int): Int {

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Partition Array for Maximum 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_sum_after_partitioning(arr: Vec<i32>, k: i32) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} arr
# @param {Integer} k
# @return {Integer}
def max_sum_after_partitioning(arr, k)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $arr
* @param Integer $k
* @return Integer
*/
function maxSumAfterPartitioning($arr, $k) {

}
}
```

Dart Solution:

```
class Solution {
int maxSumAfterPartitioning(List<int> arr, int k) {

}
}
```

Scala Solution:

```
object Solution {
def maxSumAfterPartitioning(arr: Array[Int], k: Int): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec max_sum_after_partitioning(arr :: [integer], k :: integer) :: integer
def max_sum_after_partitioning(arr, k) do

end
end
```

Erlang Solution:

```
-spec max_sum_after_partitioning(Arr :: [integer()], K :: integer()) ->
integer().
max_sum_after_partitioning(Arr, K) ->
.
```

Racket Solution:

```
(define/contract (max-sum-after-partitioning arr k)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```