Problem 3266: Final Array State After K Multiplication Operations II

Problem Information

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

Problem Description

You are given an integer array

nums

, an integer

k

, and an integer

multiplier

.

You need to perform

k

operations on

nums

. In each operation:

Find the

minimum

value

x

in

nums

. If there are multiple occurrences of the minimum value, select the one that appears

first

.

Replace the selected minimum value

x

with

x * multiplier

.

After the

k

operations, apply

modulo

10

9

+ 7

to every value in

nums

.

Return an integer array denoting the

final state

of

nums

after performing all

k

operations and then applying the modulo.

Example 1:

Input:

nums = [2,1,3,5,6], k = 5, multiplier = 2

Output:

[8,4,6,5,6]

Explanation:

Operation

Result

After operation 1

[2, 2, 3, 5, 6]

After operation 2

[4, 2, 3, 5, 6]

After operation 3

[4, 4, 3, 5, 6]

After operation 4

[4, 4, 6, 5, 6]

After operation 5

[8, 4, 6, 5, 6]

After applying modulo

[8, 4, 6, 5, 6]

Example 2:

Input:

nums = [100000,2000], k = 2, multiplier = 1000000

Output:

[999999307,999999993]

Explanation:

Operation

Result

After operation 1

[100000, 2000000000]

After operation 2

[100000000000, 2000000000]

After applying modulo

[999999307, 999999993]

Constraints:

1 <= nums.length <= 10

4

1 <= nums[i] <= 10

9

1 <= k <= 10

9

1 <= multiplier <= 10

6

Code Snippets

C++:

```
class Solution {
public:
vector<int> getFinalState(vector<int>& nums, int k, int multiplier) {
}
};
```

Java:

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

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

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

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* getFinalState(int* nums, int numsSize, int k, int multiplier, int*
returnSize) {

}
```

Go:

```
func getFinalState(nums []int, k int, multiplier int) []int {

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (get-final-state nums k multiplier)
(-> (listof exact-integer?) exact-integer? exact-integer? (listof
exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Final Array State After K Multiplication Operations II
* Difficulty: Hard
* Tags: array, queue, heap
*
```

```
* 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:
vector<int> getFinalState(vector<int>& nums, int k, int multiplier) {

}
};
```

Java Solution:

```
/**
* Problem: Final Array State After K Multiplication Operations II
* Difficulty: Hard
* Tags: array, queue, heap
*
* 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[] getFinalState(int[] nums, int k, int multiplier) {

}
}
```

Python3 Solution:

```
"""
Problem: Final Array State After K Multiplication Operations II
Difficulty: Hard
Tags: array, queue, heap

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Final Array State After K Multiplication Operations II
* Difficulty: Hard
* Tags: array, queue, heap
*
* 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
* @param {number} multiplier
* @return {number[]}
*/
var getFinalState = function(nums, k, multiplier) {

};
```

TypeScript Solution:

```
/**
* Problem: Final Array State After K Multiplication Operations II
* Difficulty: Hard
* Tags: array, queue, heap
*
* 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 getFinalState(nums: number[], k: number, multiplier: number):
number[] {

};
```

C# Solution:

```
/*
* Problem: Final Array State After K Multiplication Operations II
* Difficulty: Hard
* Tags: array, queue, heap
*
* 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[] GetFinalState(int[] nums, int k, int multiplier) {

}
}
```

C Solution:

```
/*
* Problem: Final Array State After K Multiplication Operations II
* Difficulty: Hard
* Tags: array, queue, heap
*
* 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
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* getFinalState(int* nums, int numsSize, int k, int multiplier, int*
returnSize) {

}
```

Go Solution:

```
// Problem: Final Array State After K Multiplication Operations II
// Difficulty: Hard
// Tags: array, queue, heap
//
// 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 getFinalState(nums []int, k int, multiplier int) []int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Final Array State After K Multiplication Operations II
// Difficulty: Hard
// Tags: array, queue, heap
//
// 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 get_final_state(nums: Vec<i32>, k: i32, multiplier: i32) -> Vec<i32> {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (get-final-state nums k multiplier)
(-> (listof exact-integer?) exact-integer? exact-integer? (listof
exact-integer?))
)
```