Problem 2233: Maximum Product After K Increments

Problem Information

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

Problem Description

You are given an array of non-negative integers

nums

and an integer

k

. In one operation, you may choose

any

element from

nums

and

increment

it by

1

.

Return

the

maximum

product

of

nums

after

at most

k

operations.

Since the answer may be very large, return it

modulo

10

9

+ 7

. Note that you should maximize the product before taking the modulo.

Example 1:

Input:

nums = [0,4], k = 5

Output:

20

Explanation:

Increment the first number 5 times. Now nums = [5, 4], with a product of 5 * 4 = 20. It can be shown that 20 is maximum product possible, so we return 20. Note that there may be other ways to increment nums to have the maximum product.

Example 2:

Input:

nums = [6,3,3,2], k = 2

Output:

216

Explanation:

Increment the second number 1 time and increment the fourth number 1 time. Now nums = [6, 4, 3, 3], with a product of 6 * 4 * 3 * 3 = 216. It can be shown that 216 is maximum product possible, so we return 216. Note that there may be other ways to increment nums to have the maximum product.

Constraints:

1 <= nums.length, k <= 10

5

0 <= nums[i] <= 10

6

Code Snippets

C++:

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

}
};
```

Java:

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

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

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

}
}
```

C:

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

}
```

Go:

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

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
}
```

Scala:

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

}
```

```
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (maximum-product nums k)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Maximum Product After K Increments
* Difficulty: Medium
* Tags: array, greedy, 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 maximumProduct(vector<int>& nums, int k) {
```

```
}
};
```

Java Solution:

```
/**
* Problem: Maximum Product After K Increments
* Difficulty: Medium
* Tags: array, greedy, 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 maximumProduct(int[] nums, int k) {

}
}
```

Python3 Solution:

```
"""
Problem: Maximum Product After K Increments
Difficulty: Medium
Tags: array, greedy, 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 maximumProduct(self, nums: List[int], k: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Maximum Product After K Increments
* Difficulty: Medium
* Tags: array, greedy, 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
* @return {number}
*/
var maximumProduct = function(nums, k) {

};
```

TypeScript Solution:

```
/**
* Problem: Maximum Product After K Increments
* Difficulty: Medium
* Tags: array, greedy, 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 maximumProduct(nums: number[], k: number): number {
```

```
};
```

C# Solution:

```
/*
* Problem: Maximum Product After K Increments
* Difficulty: Medium
* Tags: array, greedy, 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 MaximumProduct(int[] nums, int k) {

}
}
```

C Solution:

```
/*
* Problem: Maximum Product After K Increments
* Difficulty: Medium
* Tags: array, greedy, 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
*/

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

}
```

Go Solution:

```
// Problem: Maximum Product After K Increments
// Difficulty: Medium
```

```
// Tags: array, greedy, 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 maximumProduct(nums []int, k int) int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Maximum Product After K Increments
// Difficulty: Medium
// Tags: array, greedy, 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 maximum_product(nums: Vec<i32>, k: i32) -> i32 {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (maximum-product nums k)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```