Problem 2460: Apply Operations to an Array

Problem Information

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

Problem Description

You are given a

0-indexed

array

nums

of size

n

consisting of

non-negative

integers.

You need to apply

n - 1

operations to this array where, in the

i

th

operation (

0-indexed

), you will apply the following on the

i

th

element of

nums

:

If

nums[i] == nums[i + 1]

, then multiply

nums[i]

by

2

and set

nums[i + 1]

to

0

. Otherwise, you skip this operation.

After performing

all

the operations,

shift

all the

0

's to the

end

of the array.

For example, the array

[1,0,2,0,0,1]

after shifting all its

0

's to the end, is

[1,2,1,0,0,0]

.

Return

the resulting array

.

Note

that the operations are applied

sequentially

, not all at once.

Example 1:

Input:

nums = [1,2,2,1,1,0]

Output:

[1,4,2,0,0,0]

Explanation:

We do the following operations: - i = 0: nums[0] and nums[1] are not equal, so we skip this operation. - i = 1: nums[1] and nums[2] are equal, we multiply nums[1] by 2 and change nums[2] to 0. The array becomes [1,

4

,

0

,1,1,0]. - i = 2: nums[2] and nums[3] are not equal, so we skip this operation. - i = 3: nums[3] and nums[4] are equal, we multiply nums[3] by 2 and change nums[4] to 0. The array becomes [1,4,0,

2

,

0

,0]. - i = 4: nums[4] and nums[5] are equal, we multiply nums[4] by 2 and change nums[5] to 0. The array becomes [1,4,0,2,

0

,

0

]. After that, we shift the 0's to the end, which gives the array [1,4,2,0,0,0].

Example 2:

Input:

nums = [0,1]

Output:

[1,0]

Explanation:

No operation can be applied, we just shift the 0 to the end.

Constraints:

2 <= nums.length <= 2000

0 <= nums[i] <= 1000

Code Snippets

C++:

```
class Solution {
public:
vector<int> applyOperations(vector<int>& nums) {
```

```
}
};
```

Java:

```
class Solution {
public int[] applyOperations(int[] nums) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function applyOperations(nums: number[]): number[] {

};
```

C#:

```
public class Solution {
public int[] ApplyOperations(int[] nums) {

}
}
```

C:

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

}
```

Go:

```
func applyOperations(nums []int) []int {

}
```

Kotlin:

```
class Solution {
fun applyOperations(nums: IntArray): IntArray {

}
}
```

Swift:

```
class Solution {
func applyOperations(_ nums: [Int]) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn apply_operations(nums: Vec<i32>) -> Vec<i32> {
```

```
}
}
```

Ruby:

```
# @param {Integer[]} nums
# @return {Integer[]}
def apply_operations(nums)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums
* @return Integer[]
*/
function applyOperations($nums) {

}
}
```

Dart:

```
class Solution {
List<int> applyOperations(List<int> nums) {

}
}
```

Scala:

```
object Solution {
def applyOperations(nums: Array[Int]): Array[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec apply_operations(nums :: [integer]) :: [integer]
def apply_operations(nums) do

end
end
```

Erlang:

```
-spec apply_operations(Nums :: [integer()]) -> [integer()].
apply_operations(Nums) ->
.
```

Racket:

```
(define/contract (apply-operations nums)
(-> (listof exact-integer?) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Apply Operations to an Array
* Difficulty: Easy
* Tags: array
*
* 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> applyOperations(vector<int>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Apply Operations to an Array
* Difficulty: Easy
* Tags: array
*
* 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[] applyOperations(int[] nums) {

}
}
```

Python3 Solution:

```
"""
Problem: Apply Operations to an Array
Difficulty: Easy
Tags: array

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Apply Operations to an Array
* Difficulty: Easy
* Tags: array
*
* 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
* @return {number[]}
*/
var applyOperations = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Apply Operations to an Array
* Difficulty: Easy
* Tags: array
*
* 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 applyOperations(nums: number[]): number[] {

};
```

C# Solution:

```
/*
* Problem: Apply Operations to an Array
* Difficulty: Easy
* Tags: array
*
```

```
* 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[] ApplyOperations(int[] nums) {

}
}
```

C Solution:

```
/*
* Problem: Apply Operations to an Array
* Difficulty: Easy
* Tags: array
*
* 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* applyOperations(int* nums, int numsSize, int* returnSize) {

}
```

Go Solution:

```
// Problem: Apply Operations to an Array
// Difficulty: Easy
// Tags: array
//
// 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 applyOperations(nums []int) []int {
```

```
}
```

Kotlin Solution:

```
class Solution {
fun applyOperations(nums: IntArray): IntArray {

}
}
```

Swift Solution:

```
class Solution {
func applyOperations(_ nums: [Int]) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Apply Operations to an Array
// Difficulty: Easy
// Tags: array
//
// 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 apply_operations(nums: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} nums
# @return {Integer[]}
def apply_operations(nums)
```

```
end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $nums
* @return Integer[]
*/
function applyOperations($nums) {

}
}
```

Dart Solution:

```
class Solution {
List<int> applyOperations(List<int> nums) {

}
}
```

Scala Solution:

```
object Solution {
def applyOperations(nums: Array[Int]): Array[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec apply_operations(nums :: [integer]) :: [integer]
def apply_operations(nums) do

end
end
```

Erlang Solution:

```
-spec apply_operations(Nums :: [integer()]) -> [integer()].
apply_operations(Nums) ->
.
```

Racket Solution:

```
(define/contract (apply-operations nums)
(-> (listof exact-integer?) (listof exact-integer?))
)
```