Problem 3173: Bitwise OR of Adjacent Elements

Problem Information

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

Problem Description

Given an array

nums

of length

n

, return an array

answer

of length

n - 1

such that

answer[i] = nums[i] | nums[i + 1]

where

|

is the bitwise

OR

operation.

Example 1:

Input:

nums = [1,3,7,15]

Output:

[3,7,15]

Example 2:

Input:

nums = [8,4,2]

Output:

[12,6]

Example 3:

Input:

nums = [5,4,9,11]

Output:

[5,13,11]

Constraints:

2 <= nums.length <= 100

0 <= nums[i] <= 100

Code Snippets

C++:

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

}
};
```

Java:

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

}
}
```

Python3:

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

Python:

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

JavaScript:

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

```
};
```

TypeScript:

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

};
```

C#:

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

}
}
```

C:

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

}
```

Go:

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

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (or-array nums)
(-> (listof exact-integer?) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Bitwise OR of Adjacent Elements
* 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> orArray(vector<int>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Bitwise OR of Adjacent Elements
* 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[] orArray(int[] nums) {

}
}
```

Python3 Solution:

```
"""
Problem: Bitwise OR of Adjacent Elements
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 orArray(self, nums: List[int]) -> List[int]:
# TODO: Implement optimized solution
```

```
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Bitwise OR of Adjacent Elements
* 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 orArray = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Bitwise OR of Adjacent Elements
* 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 orArray(nums: number[]): number[] {

};
```

C# Solution:

```
/*
* Problem: Bitwise OR of Adjacent Elements
* 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[] OrArray(int[] nums) {

}
}
```

C Solution:

```
/*
* Problem: Bitwise OR of Adjacent Elements
* 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* orArray(int* nums, int numsSize, int* returnSize) {
```

```
}
```

Go Solution:

```
// Problem: Bitwise OR of Adjacent Elements
// 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 orArray(nums []int) []int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Bitwise OR of Adjacent Elements
// 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 or_array(nums: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

```
}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (or-array nums)
(-> (listof exact-integer?) (listof exact-integer?))
)
```