Problem 3670: Maximum Product of Two Integers With No Common Bits

Problem Information

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

Problem Description

You are given an integer array

nums

.

Your task is to find two

distinct

indices

i

and

j

such that the product

nums[i] * nums[j]

is

maximized,

and the binary representations of

nums[i]

and

nums[j]

do not share any common set bits.

Return the

maximum

possible product of such a pair. If no such pair exists, return 0.

Example 1:

Input:

nums = [1,2,3,4,5,6,7]

Output:

12

Explanation:

The best pair is 3 (011) and 4 (100). They share no set bits and

3 * 4 = 12

.

Example 2:

Input:

nums = [5,6,4]

Output:

0

Explanation:

Every pair of numbers has at least one common set bit. Hence, the answer is 0.

Example 3:

Input:

nums = [64,8,32]

Output:

2048

Explanation:

No pair of numbers share a common bit, so the answer is the product of the two maximum elements, 64 and 32 (

64 * 32 = 2048

).

Constraints:

2 <= nums.length <= 10

5

1 <= nums[i] <= 10

6

Code Snippets

C++:

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

}
};
```

Java:

```
class Solution {
public long maxProduct(int[] nums) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

```
public class Solution {
public long MaxProduct(int[] nums) {

}
}
```

C:

```
long long maxProduct(int* nums, int numsSize) {

}
```

Go:

```
func maxProduct(nums []int) int64 {

}
```

Kotlin:

```
class Solution {
fun maxProduct(nums: IntArray): Long {

}
}
```

Swift:

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

}
}
```

Rust:

```
impl Solution {
pub fn max_product(nums: Vec<i32>) -> i64 {

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
}
```

Scala:

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

}
```

```
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

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

Solutions

C++ Solution:

```
/*
* Problem: Maximum Product of Two Integers With No Common Bits
* 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:
long long maxProduct(vector<int>& nums) {
```

```
}
};
```

Java Solution:

```
/**
* Problem: Maximum Product of Two Integers With No Common Bits
* 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 long maxProduct(int[] nums) {

}
}
```

Python3 Solution:

```
"""
Problem: Maximum Product of Two Integers With No Common Bits
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 maxProduct(self, nums: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Maximum Product of Two Integers With No Common Bits
* 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[]} nums
* @return {number}
*/
var maxProduct = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Maximum Product of Two Integers With No Common Bits
* 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 maxProduct(nums: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Maximum Product of Two Integers With No Common Bits
* 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 long MaxProduct(int[] nums) {

}
}
```

C Solution:

```
/*
* Problem: Maximum Product of Two Integers With No Common Bits
* 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
*/

long long maxProduct(int* nums, int numsSize) {

}
```

Go Solution:

```
// Problem: Maximum Product of Two Integers With No Common Bits
// 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 maxProduct(nums []int) int64 {

}
```

Kotlin Solution:

```
class Solution {
fun maxProduct(nums: IntArray): Long {

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Maximum Product of Two Integers With No Common Bits
// 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_product(nums: Vec<i32>) -> i64 {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
```

```
end
```

Erlang Solution:

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

Racket Solution:

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