Problem 2569: Handling Sum Queries After Update

Problem Information

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

Problem Description

You are given two

0-indexed

arrays

nums1

and

nums2

and a 2D array

queries

of queries. There are three types of queries:

For a query of type 1,

queries[i] = [1, l, r]

. Flip the values from

0

to

1

and from

1

to

0

in

nums1

from index

l

to index

r

. Both

l

and

r

are

0-indexed

.

For a query of type 2,

queries[i] = [2, p, 0]

. For every index

0 <= i < n

, set

nums2[i] = nums2[i] + nums1[i] * p

.

For a query of type 3,

queries[i] = [3, 0, 0]

. Find the sum of the elements in

nums2

.

Return

an array containing all the answers to the third type queries.

Example 1:

Input:

nums1 = [1,0,1], nums2 = [0,0,0], queries = [[1,1,1],[2,1,0],[3,0,0]]

Output:

[3]

Explanation:

After the first query nums1 becomes [1,1,1]. After the second query, nums2 becomes [1,1,1], so the answer to the third query is 3. Thus, [3] is returned.

Example 2:

Input:

nums1 = [1], nums2 = [5], queries = [[2,0,0],[3,0,0]]

Output:

[5]

Explanation:

After the first query, nums2 remains [5], so the answer to the second query is 5. Thus, [5] is returned.

Constraints:

1 <= nums1.length,nums2.length <= 10

5

nums1.length = nums2.length

1 <= queries.length <= 10

5

queries[i].length = 3

0 <= l <= r <= nums1.length - 1

0 <= p <= 10

6

0 <= nums1[i] <= 1

0 <= nums2[i] <= 10

9

Code Snippets

C++:

```
class Solution {
public:
vector<long long> handleQuery(vector<int>& nums1, vector<int>& nums2,
vector<vector<int>>& queries) {

}
};
```

Java:

```
class Solution {
public long[] handleQuery(int[] nums1, int[] nums2, int[][] queries) {

}
}
```

Python3:

```
class Solution:
def handleQuery(self, nums1: List[int], nums2: List[int], queries:
List[List[int]]) -> List[int]:
```

Python:

```
class Solution(object):
def handleQuery(self, nums1, nums2, queries):
"""
:type nums1: List[int]
:type nums2: List[int]
:type queries: List[List[int]]
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {number[]} nums1
* @param {number[]} nums2
* @param {number[][]} queries
* @return {number[]}
*/
var handleQuery = function(nums1, nums2, queries) {

};
```

TypeScript:

```
function handleQuery(nums1: number[], nums2: number[], queries: number[][]):
number[] {

};
```

C#:

```
public class Solution {
public long[] HandleQuery(int[] nums1, int[] nums2, int[][] queries) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
long long* handleQuery(int* nums1, int nums1Size, int* nums2, int nums2Size,
int** queries, int queriesSize, int* queriesColSize, int* returnSize) {

}
```

Go:

```
func handleQuery(nums1 []int, nums2 []int, queries [][]int) []int64 {

}
```

Kotlin:

```
class Solution {
fun handleQuery(nums1: IntArray, nums2: IntArray, queries: Array<IntArray>):
LongArray {

}
}
```

Swift:

```
class Solution {
func handleQuery(_ nums1: [Int], _ nums2: [Int], _ queries: [[Int]]) -> [Int]
{

}
}
```

Rust:

```
impl Solution {
pub fn handle_query(nums1: Vec<i32>, nums2: Vec<i32>, queries: Vec<Vec<i32>>)
-> Vec<i64> {

}
}
```

Ruby:

```
# @param {Integer[]} nums1
# @param {Integer[]} nums2
# @param {Integer[][]} queries
# @return {Integer[]}
def handle_query(nums1, nums2, queries)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums1
```

```
* @param Integer[] $nums2
* @param Integer[][] $queries
* @return Integer[]
*/
function handleQuery($nums1, $nums2, $queries) {

}
}
```

Dart:

```
class Solution {
List<int> handleQuery(List<int> nums1, List<int> nums2, List<List<int>>
queries) {

}
}
```

Scala:

```
object Solution {
def handleQuery(nums1: Array[Int], nums2: Array[Int], queries:
Array[Array[Int]]): Array[Long] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec handle_query(nums1 :: [integer], nums2 :: [integer], queries ::
[[integer]]) :: [integer]
def handle_query(nums1, nums2, queries) do

end
end
```

Erlang:

```
-spec handle_query(Nums1 :: [integer()], Nums2 :: [integer()], Queries ::
[[integer()]]) -> [integer()].
handle_query(Nums1, Nums2, Queries) ->
```

```
.
```

Racket:

```
(define/contract (handle-query nums1 nums2 queries)
(-> (listof exact-integer?) (listof exact-integer?) (listof (listof
exact-integer?)) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Handling Sum Queries After Update
* Difficulty: Hard
* Tags: array, tree
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public:
vector<long long> handleQuery(vector<int>& nums1, vector<int>& nums2,
vector<vector<int>>& queries) {

}
};
```

Java Solution:

```
/**
* Problem: Handling Sum Queries After Update
* Difficulty: Hard
* Tags: array, tree
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
```

```
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public long[] handleQuery(int[] nums1, int[] nums2, int[][] queries) {

}
}
```

Python3 Solution:

```
"""
Problem: Handling Sum Queries After Update
Difficulty: Hard
Tags: array, tree

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(h) for recursion stack where h is height
"""

class Solution:
def handleQuery(self, nums1: List[int], nums2: List[int], queries:
List[List[int]]) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def handleQuery(self, nums1, nums2, queries):
"""
:type nums1: List[int]
:type nums2: List[int]
:type queries: List[List[int]]
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Handling Sum Queries After Update
* Difficulty: Hard
* Tags: array, tree
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* @param {number[]} nums1
* @param {number[]} nums2
* @param {number[][]} queries
* @return {number[]}
*/
var handleQuery = function(nums1, nums2, queries) {

};
```

TypeScript Solution:

```
/**
* Problem: Handling Sum Queries After Update
* Difficulty: Hard
* Tags: array, tree
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

function handleQuery(nums1: number[], nums2: number[], queries: number[][]):
number[] {

};
```

C# Solution:

```
/*
* Problem: Handling Sum Queries After Update
* Difficulty: Hard
```

```
* Tags: array, tree
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

public class Solution {
public long[] HandleQuery(int[] nums1, int[] nums2, int[][] queries) {

}
}
```

C Solution:

```
/*
* Problem: Handling Sum Queries After Update
* Difficulty: Hard
* Tags: array, tree
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
long long* handleQuery(int* nums1, int nums1Size, int* nums2, int nums2Size,
int** queries, int queriesSize, int* queriesColSize, int* returnSize) {

}
```

Go Solution:

```
// Problem: Handling Sum Queries After Update
// Difficulty: Hard
// Tags: array, tree
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
```

```
// Space Complexity: O(h) for recursion stack where h is height

func handleQuery(nums1 []int, nums2 []int, queries [][]int) []int64 {

}
```

Kotlin Solution:

```
class Solution {
fun handleQuery(nums1: IntArray, nums2: IntArray, queries: Array<IntArray>):
LongArray {

}
}
```

Swift Solution:

```
class Solution {
func handleQuery(_ nums1: [Int], _ nums2: [Int], _ queries: [[Int]]) -> [Int]
{

}
}
```

Rust Solution:

```
// Problem: Handling Sum Queries After Update
// Difficulty: Hard
// Tags: array, tree
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

impl Solution {
pub fn handle_query(nums1: Vec<i32>, nums2: Vec<i32>, queries: Vec<Vec<i32>>)
-> Vec<i64> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} nums1
# @param {Integer[]} nums2
# @param {Integer[][]} queries
# @return {Integer[]}
def handle_query(nums1, nums2, queries)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $nums1
* @param Integer[] $nums2
* @param Integer[][] $queries
* @return Integer[]
*/
function handleQuery($nums1, $nums2, $queries) {

}
}
```

Dart Solution:

```
class Solution {
List<int> handleQuery(List<int> nums1, List<int> nums2, List<List<int>>
queries) {

}
}
```

Scala Solution:

```
object Solution {
def handleQuery(nums1: Array[Int], nums2: Array[Int], queries:
Array[Array[Int]]): Array[Long] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec handle_query(nums1 :: [integer], nums2 :: [integer], queries ::
[[integer]]) :: [integer]
def handle_query(nums1, nums2, queries) do

end
end
```

Erlang Solution:

```
-spec handle_query(Nums1 :: [integer()], Nums2 :: [integer()], Queries ::
[[integer()]]) -> [integer()].
handle_query(Nums1, Nums2, Queries) ->
.
```

Racket Solution:

```
(define/contract (handle-query nums1 nums2 queries)
(-> (listof exact-integer?) (listof exact-integer?) (listof (listof
exact-integer?)) (listof exact-integer?))
)
```