Problem 3549: Multiply Two Polynomials

Problem Information

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

Problem Description

You are given two integer arrays

poly1

and

poly2

, where the element at index

i

in each array represents the coefficient of

x

i

in a polynomial.

Let

A(x)

and

B(x)

be the polynomials represented by

poly1

and

poly2

, respectively.

Return an integer array

result

of length

(poly1.length + poly2.length - 1)

representing the coefficients of the product polynomial

R(x) = A(x) * B(x)

, where

result[i]

denotes the coefficient of

x

i

in

R(x)

.

Example 1:

Input:

poly1 = [3,2,5], poly2 = [1,4]

Output:

[3,14,13,20]

Explanation:

A(x) = 3 + 2x + 5x

2

and

B(x) = 1 + 4x

R(x) = (3 + 2x + 5x

2

) * (1 + 4x)

R(x) = 3 * 1 + (3 * 4 + 2 * 1)x + (2 * 4 + 5 * 1)x

2

+ (5 * 4)x

3

R(x) = 3 + 14x + 13x

2

+ 20x

3

Thus, result =

[3, 14, 13, 20]

.

Example 2:

Input:

poly1 = [1,0,-2], poly2 = [-1]

Output:

[-1,0,2]

Explanation:

A(x) = 1 + 0x - 2x

2

and

B(x) = -1

R(x) = (1 + 0x - 2x

2

) * (-1)

R(x) = -1 + 0x + 2x

2

Thus, result =

[-1, 0, 2]

.

Example 3:

Input:

poly1 = [1,5,-3], poly2 = [-4,2,0]

Output:

[-4,-18,22,-6,0]

Explanation:

A(x) = 1 + 5x - 3x

2

and

B(x) = -4 + 2x + 0x

2

R(x) = (1 + 5x - 3x

2

) * (-4 + 2x + 0x

2

)

R(x) = 1 * -4 + (1 * 2 + 5 * -4)x + (5 * 2 + -3 * -4)x

2

+ (-3 * 2)x

3

+ 0x

4

R(x) = -4 -18x + 22x

2

-6x

3

+ 0x

4

Thus, result =

[-4, -18, 22, -6, 0]

.

Constraints:

1 <= poly1.length, poly2.length <= 5 * 10

4

-10

3

<= poly1[i], poly2[i] <= 10

3

poly1

and

poly2

contain at least one non-zero coefficient.

Code Snippets

C++:

```
class Solution {
public:
vector<long long> multiply(vector<int>& poly1, vector<int>& poly2) {

}
};
```

Java:

```
class Solution {
public long[] multiply(int[] poly1, int[] poly2) {

}
}
```

Python3:

```
class Solution:
def multiply(self, poly1: List[int], poly2: List[int]) -> List[int]:
```

Python:

```
class Solution(object):
def multiply(self, poly1, poly2):
```

```
"""
:type poly1: List[int]
:type poly2: List[int]
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {number[]} poly1
* @param {number[]} poly2
* @return {number[]}
*/
var multiply = function(poly1, poly2) {

};
```

TypeScript:

```
function multiply(poly1: number[], poly2: number[]): number[] {

};
```

C#:

```
public class Solution {
public long[] Multiply(int[] poly1, int[] poly2) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
long long* multiply(int* poly1, int poly1Size, int* poly2, int poly2Size,
int* returnSize) {

}
```

Go:

```
func multiply(poly1 []int, poly2 []int) []int64 {

}
```

Kotlin:

```
class Solution {
fun multiply(poly1: IntArray, poly2: IntArray): LongArray {

}
}
```

Swift:

```
class Solution {
func multiply(_ poly1: [Int], _ poly2: [Int]) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn multiply(poly1: Vec<i32>, poly2: Vec<i32>) -> Vec<i64> {

}
}
```

Ruby:

```
# @param {Integer[]} poly1
# @param {Integer[]} poly2
# @return {Integer[]}
def multiply(poly1, poly2)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $poly1
```

```
* @param Integer[] $poly2
* @return Integer[]
*/
function multiply($poly1, $poly2) {

}
}
```

Dart:

```
class Solution {
List<int> multiply(List<int> poly1, List<int> poly2) {

}
}
```

Scala:

```
object Solution {
def multiply(poly1: Array[Int], poly2: Array[Int]): Array[Long] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec multiply(poly1 :: [integer], poly2 :: [integer]) :: [integer]
def multiply(poly1, poly2) do

end
end
```

Erlang:

```
-spec multiply(Poly1 :: [integer()], Poly2 :: [integer()]) -> [integer()].
multiply(Poly1, Poly2) ->
.
```

Racket:

```
(define/contract (multiply poly1 poly2)
(-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Multiply Two Polynomials
* Difficulty: Hard
* Tags: array, math
*
* 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<long long> multiply(vector<int>& poly1, vector<int>& poly2) {

}
};
```

Java Solution:

```
/**
* Problem: Multiply Two Polynomials
* Difficulty: Hard
* Tags: array, math
*
* 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 long[] multiply(int[] poly1, int[] poly2) {

}
```

```
}
```

Python3 Solution:

```
"""
Problem: Multiply Two Polynomials
Difficulty: Hard
Tags: array, math

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 multiply(self, poly1: List[int], poly2: List[int]) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def multiply(self, poly1, poly2):
"""
:type poly1: List[int]
:type poly2: List[int]
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Multiply Two Polynomials
* Difficulty: Hard
* Tags: array, math
*
* 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[]} poly1
* @param {number[]} poly2
* @return {number[]}
*/
var multiply = function(poly1, poly2) {

};
```

TypeScript Solution:

```
/**
* Problem: Multiply Two Polynomials
* Difficulty: Hard
* Tags: array, math
*
* 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 multiply(poly1: number[], poly2: number[]): number[] {

};
```

C# Solution:

```
/*
* Problem: Multiply Two Polynomials
* Difficulty: Hard
* Tags: array, math
*
* 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 long[] Multiply(int[] poly1, int[] poly2) {

}
```

```
}
```

C Solution:

```
/*
* Problem: Multiply Two Polynomials
* Difficulty: Hard
* Tags: array, math
*
* 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().
*/
long long* multiply(int* poly1, int poly1Size, int* poly2, int poly2Size,
int* returnSize) {

}
```

Go Solution:

```
// Problem: Multiply Two Polynomials
// Difficulty: Hard
// Tags: array, math
//
// 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 multiply(poly1 []int, poly2 []int) []int64 {

}
```

Kotlin Solution:

```
class Solution {
fun multiply(poly1: IntArray, poly2: IntArray): LongArray {
```

```
}
}
```

Swift Solution:

```
class Solution {
func multiply(_ poly1: [Int], _ poly2: [Int]) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Multiply Two Polynomials
// Difficulty: Hard
// Tags: array, math
//
// 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 multiply(poly1: Vec<i32>, poly2: Vec<i32>) -> Vec<i64> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} poly1
# @param {Integer[]} poly2
# @return {Integer[]}
def multiply(poly1, poly2)

end
```

PHP Solution:

```
class Solution {

/**
```

```
* @param Integer[] $poly1
* @param Integer[] $poly2
* @return Integer[]
*/
function multiply($poly1, $poly2) {

}
}
```

Dart Solution:

```
class Solution {
List<int> multiply(List<int> poly1, List<int> poly2) {

}
}
```

Scala Solution:

```
object Solution {
def multiply(poly1: Array[Int], poly2: Array[Int]): Array[Long] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec multiply(poly1 :: [integer], poly2 :: [integer]) :: [integer]
def multiply(poly1, poly2) do

end
end
```

Erlang Solution:

```
-spec multiply(Poly1 :: [integer()], Poly2 :: [integer()]) -> [integer()].
multiply(Poly1, Poly2) ->
.
```

Racket Solution:

```
(define/contract (multiply poly1 poly2)
(-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?))
)
```