Problem 1013: Partition Array Into Three Parts With Equal Sum

Problem Information

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

Problem Description

Given an array of integers

arr

, return

true

if we can partition the array into three

non-empty

parts with equal sums.

Formally, we can partition the array if we can find indexes

i + 1 < j

with

(arr[0] + arr[1] + ... + arr[i] == arr[i + 1] + arr[i + 2] + ... + arr[j - 1] == arr[j] + arr[j + 1] + ... + arr[arr.length - 1])

Example 1:

Input:

arr = [0,2,1,-6,6,-7,9,1,2,0,1]

Output:

true

Explanation:

0 + 2 + 1 = -6 + 6 - 7 + 9 + 1 = 2 + 0 + 1

Example 2:

Input:

arr = [0,2,1,-6,6,7,9,-1,2,0,1]

Output:

false

Example 3:

Input:

arr = [3,3,6,5,-2,2,5,1,-9,4]

Output:

true

Explanation:

3 + 3 = 6 = 5 - 2 + 2 + 5 + 1 - 9 + 4

Constraints:

3 <= arr.length <= 5 * 10

4

-10

4

<= arr[i] <= 10

4

Code Snippets

C++:

```
class Solution {
public:
bool canThreePartsEqualSum(vector<int>& arr) {

}
};
```

Java:

```
class Solution {
public boolean canThreePartsEqualSum(int[] arr) {

}
}
```

Python3:

```
class Solution:
def canThreePartsEqualSum(self, arr: List[int]) -> bool:
```

Python:

```
class Solution(object):
def canThreePartsEqualSum(self, arr):
"""
:type arr: List[int]
```

```
:rtype: bool
"""
```

JavaScript:

```
/**
* @param {number[]} arr
* @return {boolean}
*/
var canThreePartsEqualSum = function(arr) {

};
```

TypeScript:

```
function canThreePartsEqualSum(arr: number[]): boolean {

};
```

C#:

```
public class Solution {
public bool CanThreePartsEqualSum(int[] arr) {

}
}
```

C:

```
bool canThreePartsEqualSum(int* arr, int arrSize) {

}
```

Go:

```
func canThreePartsEqualSum(arr []int) bool {

}
```

Kotlin:

```
class Solution {
fun canThreePartsEqualSum(arr: IntArray): Boolean {

}
}
```

Swift:

```
class Solution {
func canThreePartsEqualSum(_ arr: [Int]) -> Bool {

}
}
```

Rust:

```
impl Solution {
pub fn can_three_parts_equal_sum(arr: Vec<i32>) -> bool {

}
}
```

Ruby:

```
# @param {Integer[]} arr
# @return {Boolean}
def can_three_parts_equal_sum(arr)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $arr
* @return Boolean
*/
function canThreePartsEqualSum($arr) {

}
}
```

Dart:

```
class Solution {
bool canThreePartsEqualSum(List<int> arr) {

}
}
```

Scala:

```
object Solution {
def canThreePartsEqualSum(arr: Array[Int]): Boolean = {

}
}
```

Elixir:

```
defmodule Solution do
@spec can_three_parts_equal_sum(arr :: [integer]) :: boolean
def can_three_parts_equal_sum(arr) do

end
end
```

Erlang:

```
-spec can_three_parts_equal_sum(Arr :: [integer()]) -> boolean().
can_three_parts_equal_sum(Arr) ->
.
```

Racket:

```
(define/contract (can-three-parts-equal-sum arr)
(-> (listof exact-integer?) boolean?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Partition Array Into Three Parts With Equal Sum
* Difficulty: Easy
* Tags: array, greedy
*
* 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:
bool canThreePartsEqualSum(vector<int>& arr) {

}
};
```

Java Solution:

```
/**
* Problem: Partition Array Into Three Parts With Equal Sum
* Difficulty: Easy
* Tags: array, greedy
*
* 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 boolean canThreePartsEqualSum(int[] arr) {

}
}
```

Python3 Solution:

```
"""
Problem: Partition Array Into Three Parts With Equal Sum
Difficulty: Easy
Tags: array, greedy
```

```
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 canThreePartsEqualSum(self, arr: List[int]) -> bool:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def canThreePartsEqualSum(self, arr):
"""
:type arr: List[int]
:rtype: bool
"""
```

JavaScript Solution:

```
/**
* Problem: Partition Array Into Three Parts With Equal Sum
* Difficulty: Easy
* Tags: array, greedy
*
* 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[]} arr
* @return {boolean}
*/
var canThreePartsEqualSum = function(arr) {

};
```

TypeScript Solution:

```
/**
* Problem: Partition Array Into Three Parts With Equal Sum
* Difficulty: Easy
* Tags: array, greedy
*
* 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 canThreePartsEqualSum(arr: number[]): boolean {

};
```

C# Solution:

```
/*
* Problem: Partition Array Into Three Parts With Equal Sum
* Difficulty: Easy
* Tags: array, greedy
*
* 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 bool CanThreePartsEqualSum(int[] arr) {

}
}
```

C Solution:

```
/*
* Problem: Partition Array Into Three Parts With Equal Sum
* Difficulty: Easy
* Tags: array, greedy
*
* 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
```

```
*/

bool canThreePartsEqualSum(int* arr, int arrSize) {

}
```

Go Solution:

```
// Problem: Partition Array Into Three Parts With Equal Sum
// Difficulty: Easy
// Tags: array, greedy
//
// 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 canThreePartsEqualSum(arr []int) bool {

}
```

Kotlin Solution:

```
class Solution {
fun canThreePartsEqualSum(arr: IntArray): Boolean {

}
}
```

Swift Solution:

```
class Solution {
func canThreePartsEqualSum(_ arr: [Int]) -> Bool {

}
}
```

Rust Solution:

```
// Problem: Partition Array Into Three Parts With Equal Sum
// Difficulty: Easy
// Tags: array, greedy
```

```
//
// 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 can_three_parts_equal_sum(arr: Vec<i32>) -> bool {

}
}
```

Ruby Solution:

```
# @param {Integer[]} arr
# @return {Boolean}
def can_three_parts_equal_sum(arr)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $arr
* @return Boolean
*/
function canThreePartsEqualSum($arr) {

}
}
```

Dart Solution:

```
class Solution {
bool canThreePartsEqualSum(List<int> arr) {

}
}
```

Scala Solution:

```
object Solution {
def canThreePartsEqualSum(arr: Array[Int]): Boolean = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec can_three_parts_equal_sum(arr :: [integer]) :: boolean
def can_three_parts_equal_sum(arr) do

end
end
```

Erlang Solution:

```
-spec can_three_parts_equal_sum(Arr :: [integer()]) -> boolean().
can_three_parts_equal_sum(Arr) ->
.
```

Racket Solution:

```
(define/contract (can-three-parts-equal-sum arr)
(-> (listof exact-integer?) boolean?)
)
```