Problem 927: Three Equal Parts

Problem Information

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

Problem Description

You are given an array

arr

which consists of only zeros and ones, divide the array into

three non-empty parts

such that all of these parts represent the same binary value.

If it is possible, return any

[i, j]

with

i + 1 < j

, such that:

arr[0], arr[1], ..., arr[i]

is the first part,

arr[i + 1], arr[i + 2], ..., arr[j - 1]

is the second part, and

arr[j], arr[j + 1], ..., arr[arr.length - 1]

is the third part.

All three parts have equal binary values.

If it is not possible, return

[-1, -1]

.

Note that the entire part is used when considering what binary value it represents. For example,

[1,1,0]

represents

6

in decimal, not

3

. Also, leading zeros

are allowed

, so

[0,1,1]

and

[1,1]

represent the same value.

Example 1:

Input:

arr = [1,0,1,0,1]

Output:

[0,3]

Example 2:

Input:

arr = [1,1,0,1,1]

Output:

[-1,-1]

Example 3:

Input:

arr = [1,1,0,0,1]

Output:

[0,2]

Constraints:

3 <= arr.length <= 3 * 10

4

arr[i]

is

0

or

1

Code Snippets

C++:

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

}
};
```

Java:

```
class Solution {
public int[] threeEqualParts(int[] arr) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function threeEqualParts(arr: number[]): number[] {

};
```

C#:

```
public class Solution {
public int[] ThreeEqualParts(int[] arr) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* threeEqualParts(int* arr, int arrSize, int* returnSize) {

}
```

Go:

```
func threeEqualParts(arr []int) []int {

}
```

Kotlin:

```
class Solution {
fun threeEqualParts(arr: IntArray): IntArray {
```

```
}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

```
class Solution {
List<int> threeEqualParts(List<int> arr) {

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

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

Solutions

C++ Solution:

```
/*
* Problem: Three Equal Parts
```

```
* 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<int> threeEqualParts(vector<int>& arr) {

}
};
```

Java Solution:

```
/**
* Problem: Three Equal Parts
* 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 int[] threeEqualParts(int[] arr) {

}
}
```

Python3 Solution:

```
"""
Problem: Three Equal Parts
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 threeEqualParts(self, arr: List[int]) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Three Equal Parts
* 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[]} arr
* @return {number[]}
*/
var threeEqualParts = function(arr) {

};
```

TypeScript Solution:

```
/**
* Problem: Three Equal Parts
* 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 threeEqualParts(arr: number[]): number[] {

};
```

C# Solution:

```
/*
* Problem: Three Equal Parts
* 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 int[] ThreeEqualParts(int[] arr) {

}
}
```

C Solution:

```
/*
* Problem: Three Equal Parts
* 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().
*/
int* threeEqualParts(int* arr, int arrSize, int* returnSize) {

}
```

Go Solution:

```
// Problem: Three Equal Parts
// 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 threeEqualParts(arr []int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun threeEqualParts(arr: IntArray): IntArray {

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Three Equal Parts
// 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 three_equal_parts(arr: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

```
class Solution {
List<int> threeEqualParts(List<int> arr) {

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

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