Problem 605: Can Place Flowers

Problem Information

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

Problem Description

You have a long flowerbed in which some of the plots are planted, and some are not. However, flowers cannot be planted in

adjacent

plots.

Given an integer array

flowerbed

containing

0

's and

1

's, where

0

means empty and

1

means not empty, and an integer

n

, return

true

if

n

new flowers can be planted in the

flowerbed

without violating the no-adjacent-flowers rule and

false

otherwise

.

Example 1:

Input:

flowerbed = [1,0,0,0,1], n = 1

Output:

true

Example 2:

Input:

flowerbed = [1,0,0,0,1], n = 2

Output:

false

Constraints:

1 <= flowerbed.length <= 2 * 10

4

flowerbed[i]

is

0

or

1

.

There are no two adjacent flowers in

flowerbed

.

0 <= n <= flowerbed.length

Code Snippets

C++:

```
class Solution {
public:
```

```
bool canPlaceFlowers(vector<int>& flowerbed, int n) {

}
};
```

Java:

```
class Solution {
public boolean canPlaceFlowers(int[] flowerbed, int n) {

}
}
```

Python3:

```
class Solution:
def canPlaceFlowers(self, flowerbed: List[int], n: int) -> bool:
```

Python:

```
class Solution(object):
def canPlaceFlowers(self, flowerbed, n):
"""
:type flowerbed: List[int]
:type n: int
:rtype: bool
"""
```

JavaScript:

```
/**
* @param {number[]} flowerbed
* @param {number} n
* @return {boolean}
*/
var canPlaceFlowers = function(flowerbed, n) {

};
```

TypeScript:

```
function canPlaceFlowers(flowerbed: number[], n: number): boolean {

};
```

C#:

```
public class Solution {
public bool CanPlaceFlowers(int[] flowerbed, int n) {

}
}
```

C:

```
bool canPlaceFlowers(int* flowerbed, int flowerbedSize, int n) {

}
```

Go:

```
func canPlaceFlowers(flowerbed []int, n int) bool {

}
```

Kotlin:

```
class Solution {
fun canPlaceFlowers(flowerbed: IntArray, n: Int): Boolean {

}
}
```

Swift:

```
class Solution {
func canPlaceFlowers(_ flowerbed: [Int], _ n: Int) -> Bool {

}
}
```

Rust:

```
impl Solution {
pub fn can_place_flowers(flowerbed: Vec<i32>, n: i32) -> bool {

}
}
```

Ruby:

```
# @param {Integer[]} flowerbed
# @param {Integer} n
# @return {Boolean}
def can_place_flowers(flowerbed, n)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $flowerbed
* @param Integer $n
* @return Boolean
*/
function canPlaceFlowers($flowerbed, $n) {

}
}
```

Dart:

```
class Solution {
bool canPlaceFlowers(List<int> flowerbed, int n) {

}
}
```

Scala:

```
object Solution {
def canPlaceFlowers(flowerbed: Array[Int], n: Int): Boolean = {

}
```

```
}
```

Elixir:

```
defmodule Solution do
@spec can_place_flowers(flowerbed :: [integer], n :: integer) :: boolean
def can_place_flowers(flowerbed, n) do

end
end
```

Erlang:

```
-spec can_place_flowers(Flowerbed :: [integer()], N :: integer()) ->
boolean().
can_place_flowers(Flowerbed, N) ->
.
```

Racket:

```
(define/contract (can-place-flowers flowerbed n)
(-> (listof exact-integer?) exact-integer? boolean?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Can Place Flowers
* 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 canPlaceFlowers(vector<int>& flowerbed, int n) {

}
};
```

Java Solution:

```
/**
* Problem: Can Place Flowers
* 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 canPlaceFlowers(int[] flowerbed, int n) {

}
}
```

Python3 Solution:

```
"""
Problem: Can Place Flowers
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 canPlaceFlowers(self, flowerbed: List[int], n: int) -> bool:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def canPlaceFlowers(self, flowerbed, n):
"""
:type flowerbed: List[int]
:type n: int
:rtype: bool
"""
```

JavaScript Solution:

```
/**
* Problem: Can Place Flowers
* 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[]} flowerbed
* @param {number} n
* @return {boolean}
*/
var canPlaceFlowers = function(flowerbed, n) {

};
```

TypeScript Solution:

```
/**
* Problem: Can Place Flowers
* 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 canPlaceFlowers(flowerbed: number[], n: number): boolean {
```

```
};
```

C# Solution:

```
/*
* Problem: Can Place Flowers
* 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 CanPlaceFlowers(int[] flowerbed, int n) {

}
}
```

C Solution:

```
/*
* Problem: Can Place Flowers
* 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 canPlaceFlowers(int* flowerbed, int flowerbedSize, int n) {

}
```

Go Solution:

```
// Problem: Can Place Flowers
// 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 canPlaceFlowers(flowerbed []int, n int) bool {

}
```

Kotlin Solution:

```
class Solution {
fun canPlaceFlowers(flowerbed: IntArray, n: Int): Boolean {

}
}
```

Swift Solution:

```
class Solution {
func canPlaceFlowers(_ flowerbed: [Int], _ n: Int) -> Bool {

}
}
```

Rust Solution:

```
// Problem: Can Place Flowers
// 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_place_flowers(flowerbed: Vec<i32>, n: i32) -> bool {

}
}
```

Ruby Solution:

```
# @param {Integer[]} flowerbed
# @param {Integer} n
# @return {Boolean}
def can_place_flowers(flowerbed, n)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $flowerbed
* @param Integer $n
* @return Boolean
*/
function canPlaceFlowers($flowerbed, $n) {

}
}
```

Dart Solution:

```
class Solution {
bool canPlaceFlowers(List<int> flowerbed, int n) {

}
}
```

Scala Solution:

```
object Solution {
def canPlaceFlowers(flowerbed: Array[Int], n: Int): Boolean = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec can_place_flowers(flowerbed :: [integer], n :: integer) :: boolean
def can_place_flowers(flowerbed, n) do

end
end
```

Erlang Solution:

```
-spec can_place_flowers(Flowerbed :: [integer()], N :: integer()) ->
boolean().
can_place_flowers(Flowerbed, N) ->
.
```

Racket Solution:

```
(define/contract (can-place-flowers flowerbed n)
(-> (listof exact-integer?) exact-integer? boolean?)
)
```