Problem 134: Gas Station

Problem Information

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

Problem Description

There are

n

gas stations along a circular route, where the amount of gas at the

i

th

station is

gas[i]

.

You have a car with an unlimited gas tank and it costs

cost[i]

of gas to travel from the

i

th

station to its next

(i + 1)

th

station. You begin the journey with an empty tank at one of the gas stations.

Given two integer arrays

gas

and

cost

, return

the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return

-1

. If there exists a solution, it is

guaranteed

to be

unique

.

Example 1:

Input:

gas = [1,2,3,4,5], cost = [3,4,5,1,2]

Output:

3

Explanation:

Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4 Travel to station 4. Your tank = 4 - 1 + 5 = 8 Travel to station 0. Your tank = 8 - 2 + 1 = 7 Travel to station 1. Your tank = 7 - 3 + 2 = 6 Travel to station 2. Your tank = 6 - 4 + 3 = 5 Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3. Therefore, return 3 as the starting index.

Example 2:

Input:

gas = [2,3,4], cost = [3,4,3]

Output:

-1

Explanation:

You can't start at station 0 or 1, as there is not enough gas to travel to the next station. Let's start at station 2 and fill up with 4 unit of gas. Your tank = 0 + 4 = 4 Travel to station 0. Your tank = 4 - 3 + 2 = 3 Travel to station 1. Your tank = 3 - 3 + 3 = 3 You cannot travel back to station 2, as it requires 4 unit of gas but you only have 3. Therefore, you can't travel around the circuit once no matter where you start.

Constraints:

n == gas.length == cost.length

1 <= n <= 10

5

0 <= gas[i], cost[i] <= 10

4

The input is generated such that the answer is unique.

Code Snippets

C++:

```
class Solution {
public:
int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {

}
};
```

Java:

```
class Solution {
public int canCompleteCircuit(int[] gas, int[] cost) {

}
}
```

Python3:

```
class Solution:
def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:
```

Python:

```
class Solution(object):
def canCompleteCircuit(self, gas, cost):
"""
:type gas: List[int]
:type cost: List[int]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} gas
* @param {number[]} cost
* @return {number}
*/
var canCompleteCircuit = function(gas, cost) {

};
```

TypeScript:

```
function canCompleteCircuit(gas: number[], cost: number[]): number {

};
```

C#:

```
public class Solution {
public int CanCompleteCircuit(int[] gas, int[] cost) {

}
}
```

C:

```
int canCompleteCircuit(int* gas, int gasSize, int* cost, int costSize) {

}
```

Go:

```
func canCompleteCircuit(gas []int, cost []int) int {

}
```

Kotlin:

```
class Solution {
fun canCompleteCircuit(gas: IntArray, cost: IntArray): Int {

}
}
```

Swift:

```
class Solution {
func canCompleteCircuit(_ gas: [Int], _ cost: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn can_complete_circuit(gas: Vec<i32>, cost: Vec<i32>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} gas
# @param {Integer[]} cost
# @return {Integer}
def can_complete_circuit(gas, cost)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $gas
* @param Integer[] $cost
* @return Integer
*/
function canCompleteCircuit($gas, $cost) {

}
}
```

Dart:

```
class Solution {
int canCompleteCircuit(List<int> gas, List<int> cost) {
```

```
}
}
```

Scala:

```
object Solution {
def canCompleteCircuit(gas: Array[Int], cost: Array[Int]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec can_complete_circuit(gas :: [integer], cost :: [integer]) :: integer
def can_complete_circuit(gas, cost) do

end
end
```

Erlang:

```
-spec can_complete_circuit(Gas :: [integer()], Cost :: [integer()]) ->
integer().
can_complete_circuit(Gas, Cost) ->
.
```

Racket:

```
(define/contract (can-complete-circuit gas cost)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Gas Station
```

```
* Difficulty: Medium
* 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:
int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {

}
};
```

Java Solution:

```
/**
* Problem: Gas Station
* Difficulty: Medium
* 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 int canCompleteCircuit(int[] gas, int[] cost) {

}
}
```

Python3 Solution:

```
"""
Problem: Gas Station
Difficulty: Medium
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 canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def canCompleteCircuit(self, gas, cost):
"""
:type gas: List[int]
:type cost: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Gas Station
* Difficulty: Medium
* 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[]} gas
* @param {number[]} cost
* @return {number}
*/
var canCompleteCircuit = function(gas, cost) {

};
```

TypeScript Solution:

```
/**
* Problem: Gas Station
* Difficulty: Medium
* 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 canCompleteCircuit(gas: number[], cost: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Gas Station
* Difficulty: Medium
* 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 int CanCompleteCircuit(int[] gas, int[] cost) {

}
}
```

C Solution:

```
/*
* Problem: Gas Station
* Difficulty: Medium
* 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
```

```
*/

int canCompleteCircuit(int* gas, int gasSize, int* cost, int costSize) {

}
```

Go Solution:

```
// Problem: Gas Station
// Difficulty: Medium
// 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 canCompleteCircuit(gas []int, cost []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun canCompleteCircuit(gas: IntArray, cost: IntArray): Int {

}
}
```

Swift Solution:

```
class Solution {
func canCompleteCircuit(_ gas: [Int], _ cost: [Int]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Gas Station
// Difficulty: Medium
// 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_complete_circuit(gas: Vec<i32>, cost: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} gas
# @param {Integer[]} cost
# @return {Integer}
def can_complete_circuit(gas, cost)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $gas
* @param Integer[] $cost
* @return Integer
*/
function canCompleteCircuit($gas, $cost) {

}
}
```

Dart Solution:

```
class Solution {
int canCompleteCircuit(List<int> gas, List<int> cost) {

}
}
```

Scala Solution:

```
object Solution {
def canCompleteCircuit(gas: Array[Int], cost: Array[Int]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec can_complete_circuit(gas :: [integer], cost :: [integer]) :: integer
def can_complete_circuit(gas, cost) do

end
end
```

Erlang Solution:

```
-spec can_complete_circuit(Gas :: [integer()], Cost :: [integer()]) ->
integer().
can_complete_circuit(Gas, Cost) ->
.
```

Racket Solution:

```
(define/contract (can-complete-circuit gas cost)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?)
)
```