Problem 1094: Car Pooling

Problem Information

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

Problem Description

There is a car with

capacity

empty seats. The vehicle only drives east (i.e., it cannot turn around and drive west).

You are given the integer

capacity

and an array

trips

where

trips[i] = [numPassengers

i

, from

i

, to

i

]

indicates that the

i

th

trip has

numPassengers

i

passengers and the locations to pick them up and drop them off are

from

i

and

to

i

respectively. The locations are given as the number of kilometers due east from the car's initial location.

Return

true

if it is possible to pick up and drop off all passengers for all the given trips, or

false

otherwise

.

Example 1:

Input:

trips = [[2,1,5],[3,3,7]], capacity = 4

Output:

false

Example 2:

Input:

trips = [[2,1,5],[3,3,7]], capacity = 5

Output:

true

Constraints:

1 <= trips.length <= 1000

trips[i].length == 3

1 <= numPassengers

i

<= 100

0 <= from

i

< to

i

<= 1000

1 <= capacity <= 10

5

Code Snippets

C++:

```
class Solution {
public:
bool carPooling(vector<vector<int>>& trips, int capacity) {

}
};
```

Java:

```
class Solution {
public boolean carPooling(int[][] trips, int capacity) {

}
}
```

Python3:

```
class Solution:
def carPooling(self, trips: List[List[int]], capacity: int) -> bool:
```

Python:

```
class Solution(object):
def carPooling(self, trips, capacity):
"""
:type trips: List[List[int]]
```

```
:type capacity: int
:rtype: bool
"""
```

JavaScript:

```
/**
* @param {number[][]} trips
* @param {number} capacity
* @return {boolean}
*/
var carPooling = function(trips, capacity) {

};
```

TypeScript:

```
function carPooling(trips: number[][], capacity: number): boolean {

};
```

C#:

```
public class Solution {
public bool CarPooling(int[][] trips, int capacity) {

}
}
```

C:

```
bool carPooling(int** trips, int tripsSize, int* tripsColSize, int capacity)
{

}
```

Go:

```
func carPooling(trips [][]int, capacity int) bool {

}
```

Kotlin:

```
class Solution {
fun carPooling(trips: Array<IntArray>, capacity: Int): Boolean {

}
}
```

Swift:

```
class Solution {
func carPooling(_ trips: [[Int]], _ capacity: Int) -> Bool {

}
}
```

Rust:

```
impl Solution {
pub fn car_pooling(trips: Vec<Vec<i32>>, capacity: i32) -> bool {

}
}
```

Ruby:

```
# @param {Integer[][]} trips
# @param {Integer} capacity
# @return {Boolean}
def car_pooling(trips, capacity)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $trips
* @param Integer $capacity
* @return Boolean
*/
function carPooling($trips, $capacity) {
```

```
}
}
```

Dart:

```
class Solution {
bool carPooling(List<List<int>> trips, int capacity) {

}
}
```

Scala:

```
object Solution {
def carPooling(trips: Array[Array[Int]], capacity: Int): Boolean = {

}
}
```

Elixir:

```
defmodule Solution do
@spec car_pooling(trips :: [[integer]], capacity :: integer) :: boolean
def car_pooling(trips, capacity) do

end
end
```

Erlang:

```
-spec car_pooling(Trips :: [[integer()]], Capacity :: integer()) ->
boolean().
car_pooling(Trips, Capacity) ->
.
```

Racket:

```
(define/contract (car-pooling trips capacity)
(-> (listof (listof exact-integer?)) exact-integer? boolean?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Car Pooling
* Difficulty: Medium
* Tags: array, sort, queue, heap
*
* 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 carPooling(vector<vector<int>>& trips, int capacity) {

}
};
```

Java Solution:

```
/**
* Problem: Car Pooling
* Difficulty: Medium
* Tags: array, sort, queue, heap
*
* 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 carPooling(int[][] trips, int capacity) {

}
}
```

Python3 Solution:

```
"""
Problem: Car Pooling
Difficulty: Medium
Tags: array, sort, queue, heap

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 carPooling(self, trips: List[List[int]], capacity: int) -> bool:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def carPooling(self, trips, capacity):
"""
:type trips: List[List[int]]
:type capacity: int
:rtype: bool
"""
```

JavaScript Solution:

```
/**
* Problem: Car Pooling
* Difficulty: Medium
* Tags: array, sort, queue, heap
*
* 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[][]} trips
* @param {number} capacity
* @return {boolean}
*/
```

```
var carPooling = function(trips, capacity) {

};
```

TypeScript Solution:

```
/**
* Problem: Car Pooling
* Difficulty: Medium
* Tags: array, sort, queue, heap
*
* 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 carPooling(trips: number[][], capacity: number): boolean {

};
```

C# Solution:

```
/*
* Problem: Car Pooling
* Difficulty: Medium
* Tags: array, sort, queue, heap
*
* 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 CarPooling(int[][] trips, int capacity) {

}
}
```

C Solution:

```
/*
* Problem: Car Pooling
* Difficulty: Medium
* Tags: array, sort, queue, heap
*
* 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 carPooling(int** trips, int tripsSize, int* tripsColSize, int capacity)
{

}
```

Go Solution:

```
// Problem: Car Pooling
// Difficulty: Medium
// Tags: array, sort, queue, heap
//
// 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 carPooling(trips [][]int, capacity int) bool {

}
```

Kotlin Solution:

```
class Solution {
fun carPooling(trips: Array<IntArray>, capacity: Int): Boolean {

}
}
```

Swift Solution:

```
class Solution {
func carPooling(_ trips: [[Int]], _ capacity: Int) -> Bool {
```

```
}
}
```

Rust Solution:

```
// Problem: Car Pooling
// Difficulty: Medium
// Tags: array, sort, queue, heap
//
// 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 car_pooling(trips: Vec<Vec<i32>>, capacity: i32) -> bool {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} trips
# @param {Integer} capacity
# @return {Boolean}
def car_pooling(trips, capacity)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $trips
* @param Integer $capacity
* @return Boolean
*/
function carPooling($trips, $capacity) {

}
}
```

Dart Solution:

```
class Solution {
bool carPooling(List<List<int>> trips, int capacity) {

}
}
```

Scala Solution:

```
object Solution {
def carPooling(trips: Array[Array[Int]], capacity: Int): Boolean = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec car_pooling(trips :: [[integer]], capacity :: integer) :: boolean
def car_pooling(trips, capacity) do

end
end
```

Erlang Solution:

```
-spec car_pooling(Trips :: [[integer()]], Capacity :: integer()) ->
boolean().
car_pooling(Trips, Capacity) ->
.
```

Racket Solution:

```
(define/contract (car-pooling trips capacity)
(-> (listof (listof exact-integer?)) exact-integer? boolean?)
)
```